

Computação II

MAB 225

OO

Brunno Goldstein

bfgoldstein@cos.ufrj.br

www.cos.ufrj.br/~bfgoldstein

Ementa

- Programação Orientada a Objetos
- Tratamento de Exceções
- Módulos
- Manipulação de Arquivos
- Interface Gráfica (Tkinter)
- Biblioteca Numérica (Numpy)

Ementa

- Programação Orientada a Objetos
- Tratamento de Exceções
- Módulos
- Manipulação de Arquivos
- Interface Gráfica (Tkinter)
- Biblioteca Numérica (Numpy)

Relembrando Computação 1

- Programação estruturada;
 - C, Pascal, Fortran, Python, etc.
- Programa é definido através de uma sequência de instruções e chamadas de funções que manipulam os dados;
- Ótima opção para códigos pequenos e de rápida implementação.

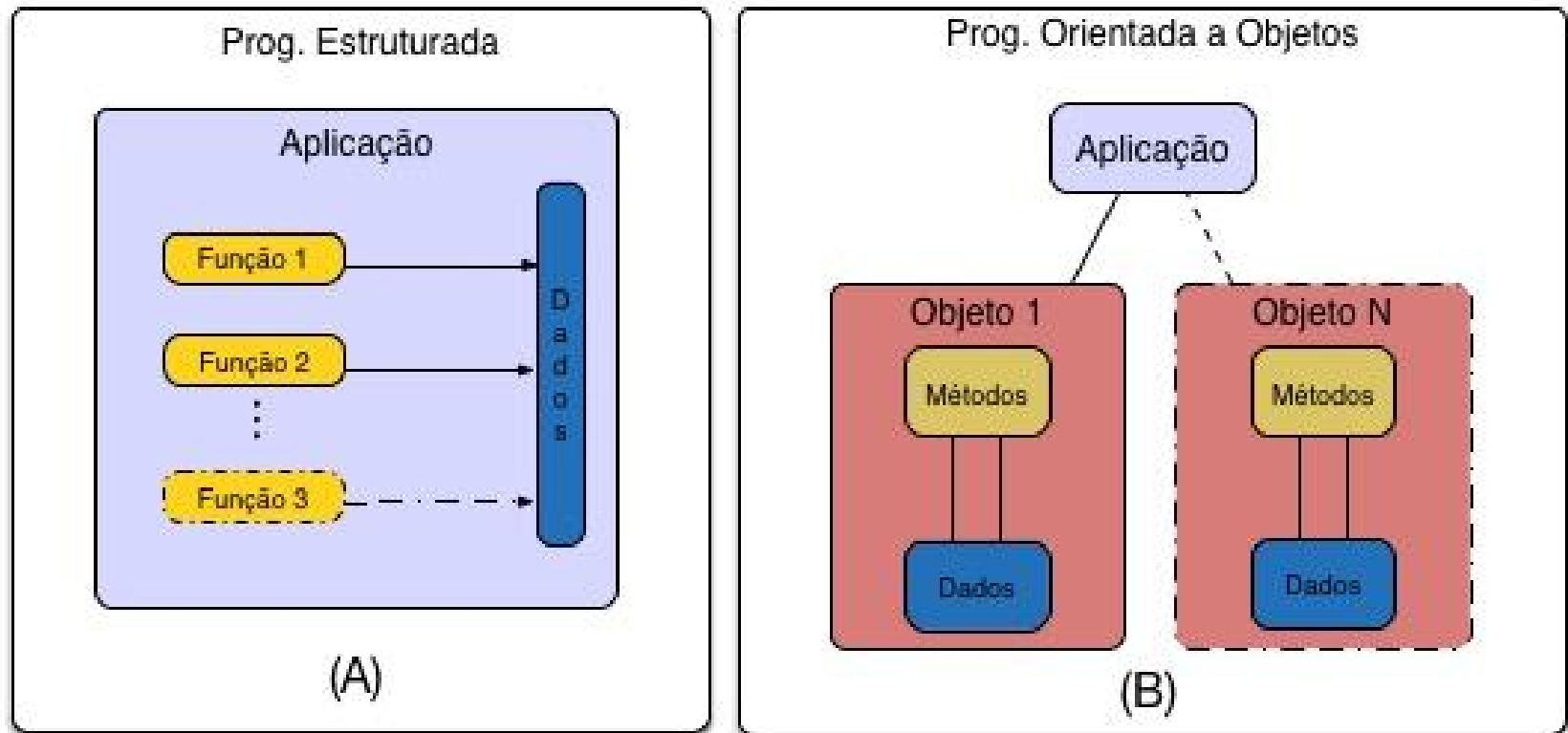
Relembrando Computação 1

```
# -*- coding: utf-8 -*-  
  
#função soma  
def soma(x, y):  
    res = x + y  
    return res  
  
#Início do programa  
x = 5  
y = 10  
resultado = soma(5,10)  
#imprime o resultado  
print "%d + %d = %d" % (x, y, resultado)
```

Programação Orientada a Objetos

- Diferente da programação estruturada;
- Modelo de programação que reflete melhor o mundo real;
- Mais fácil de compreender e modelar o problema no código;

Programação Orientada a Objetos



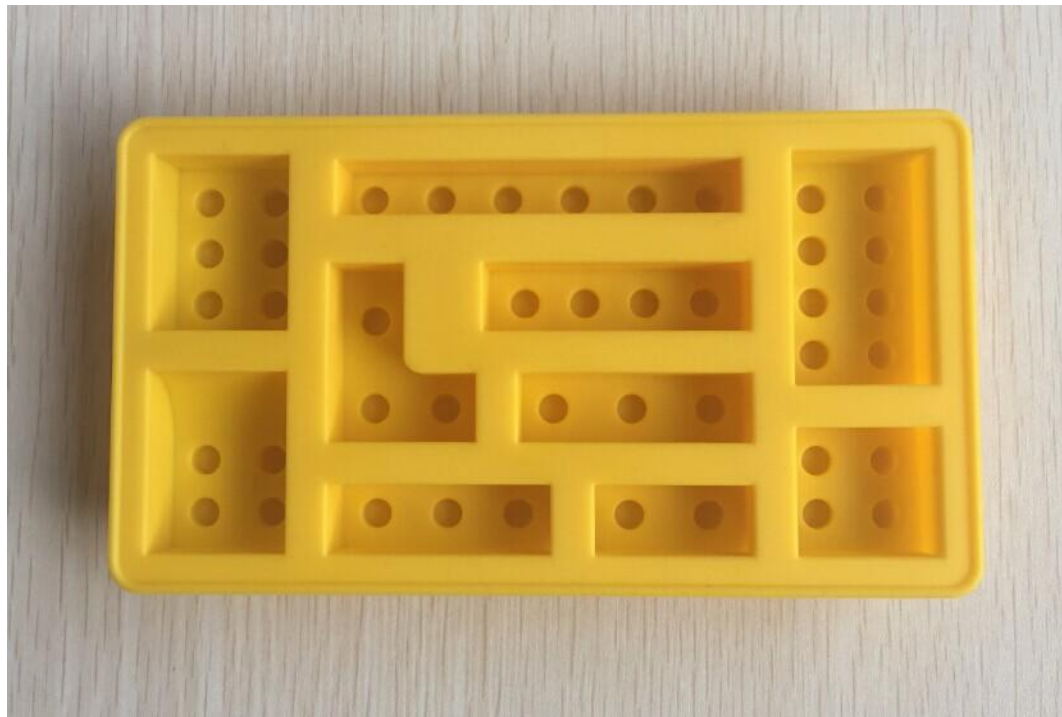
Estruturada vs Orientada a Objetos

Programação Orientada a Objetos

- Conceitos importantes:
 - Classe
 - Objeto
 - Método
 - Atributo

Programação Orientada a Objetos

- Classe é um molde de objetos;
- Possui as informações/variáveis (atributos) e as funções (métodos) que os objetos vão poder exercer.



Programação Orientada a Objetos

- Objetos são gerados a partir das classes;
- Essa "criação" é chamada de instanciação do objeto;
- E aquele objeto passa a ser uma instância da classe.



Programação Orientada a Objetos

Classe X



Instanciação

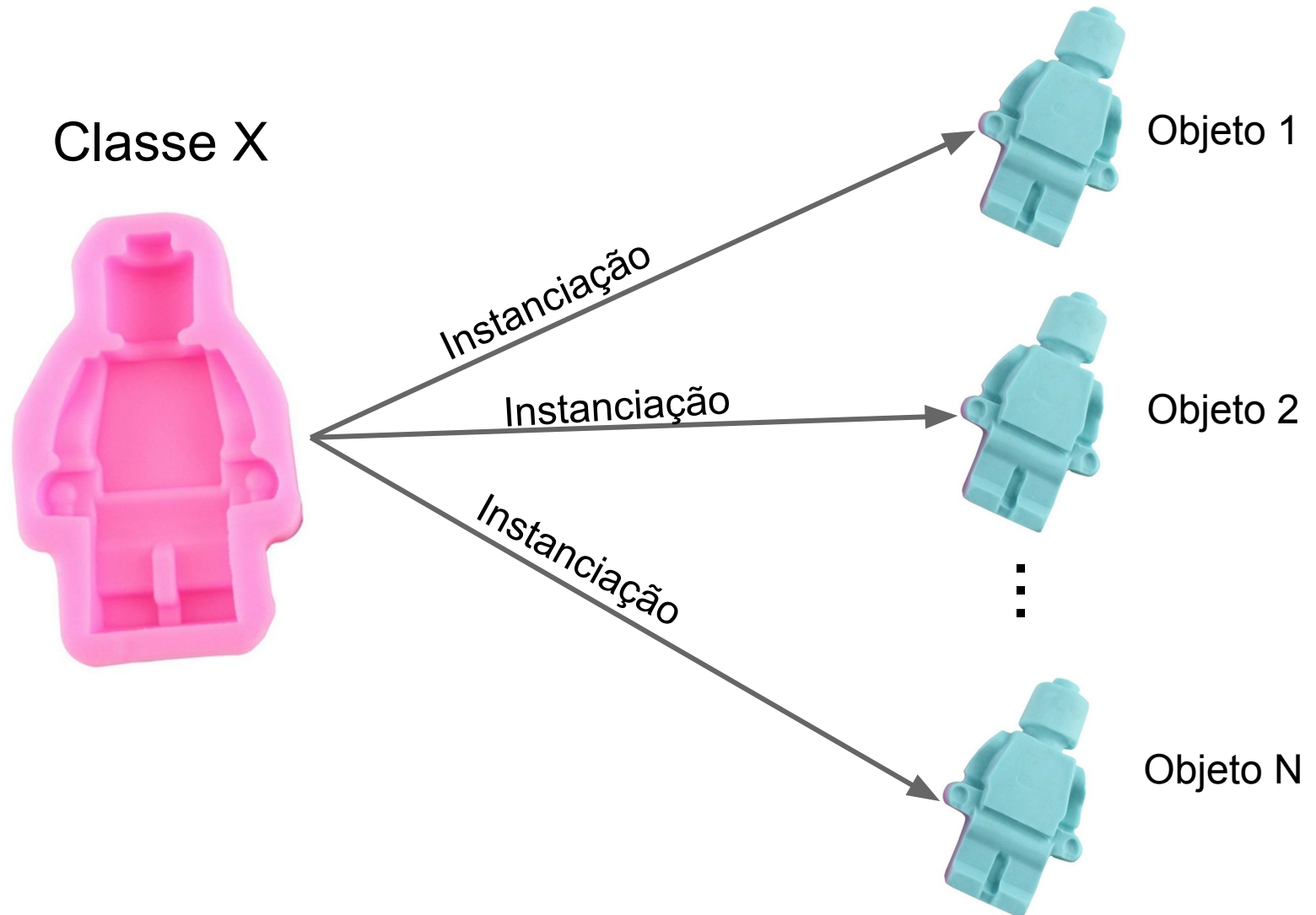


Objeto 1



Dizemos que Objeto 1 é uma instância da Classe X.

Programação Orientada a Objetos



Programação Orientada a Objetos

- O grupo de objetos criados forma então a sua aplicação/programa.



Programação Orientada a Objetos



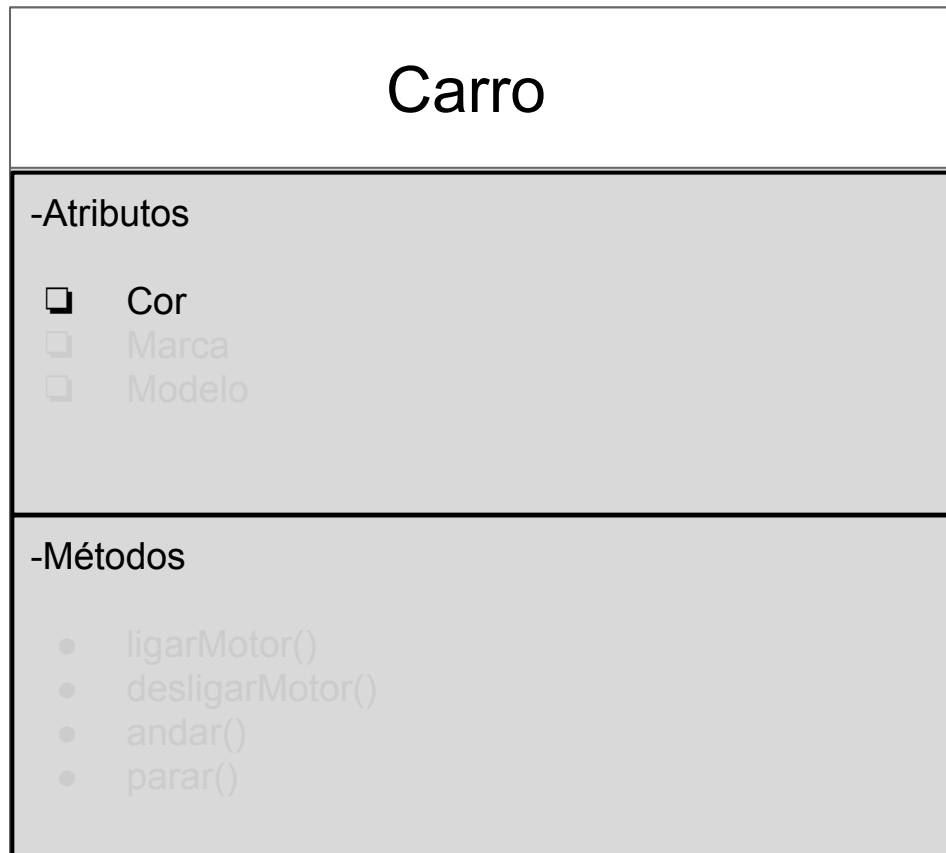
- A porta, por exemplo, é um objeto da classe Porta;
- Possui um atributo cor cujo valor é 'amarelo';
- Possui dois métodos (ações): abrir e fechar;

Vamos por partes...

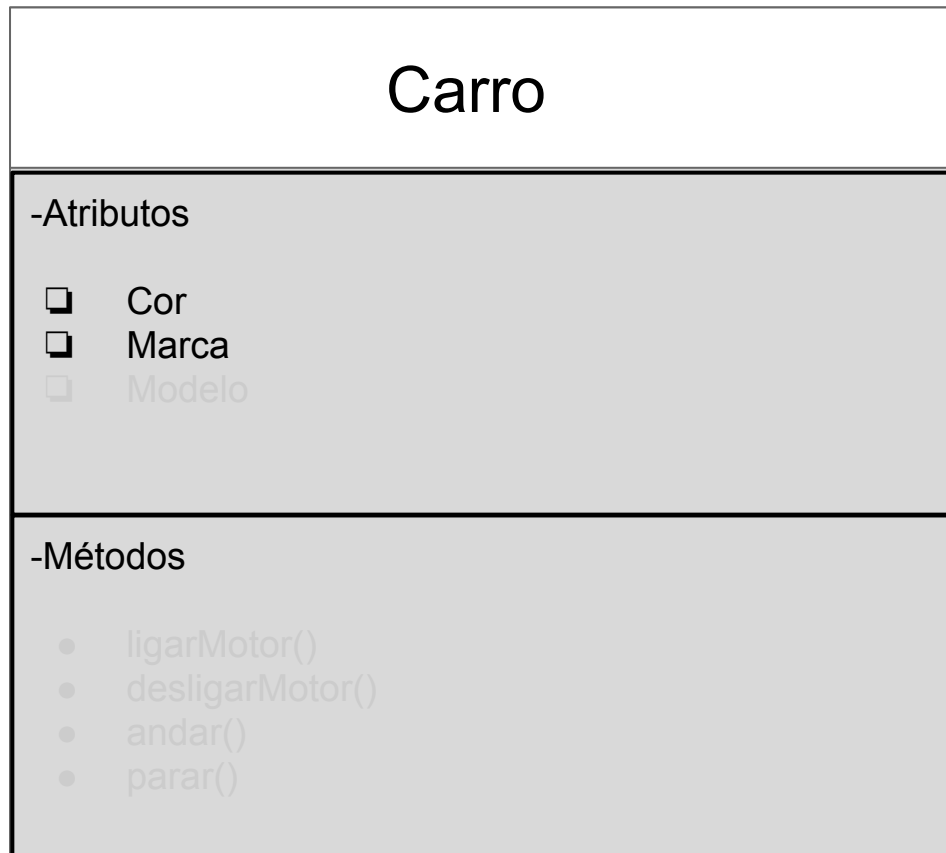
1. A primeira coisa a se fazer é definir sua classe:
 - a. Quais atributos os objetos dessa classe devem possuir?
 - b. Quais métodos (ações) eles devem ser capazes de fazer?

Vamos criar a classe carro!

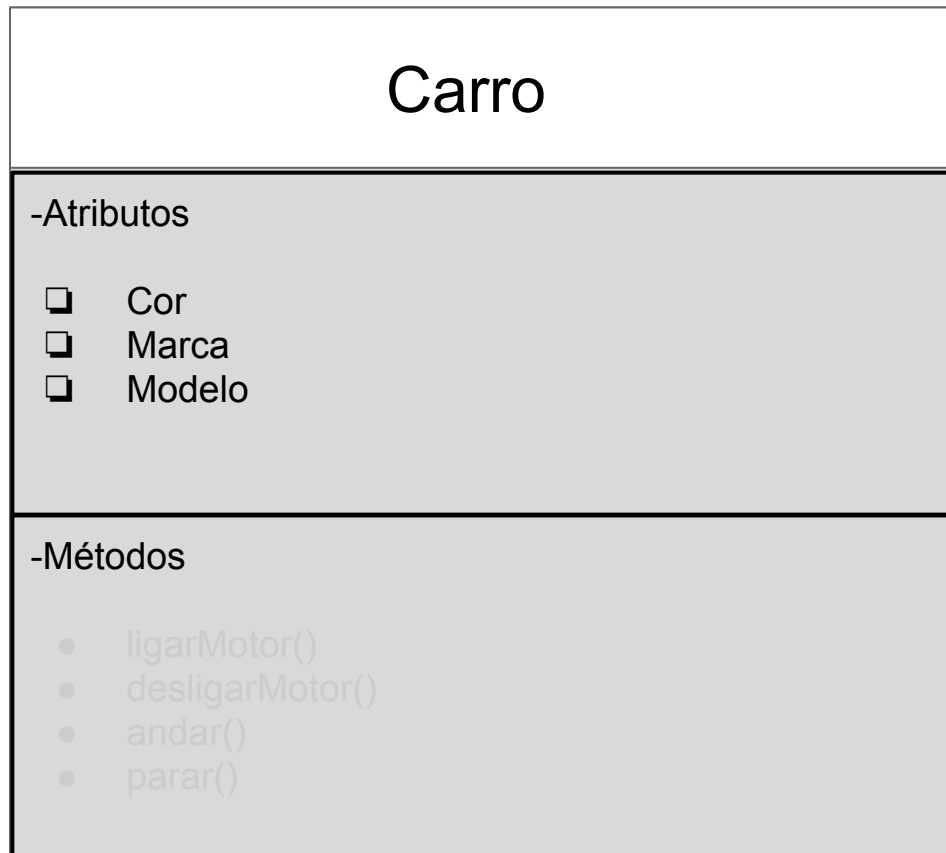
Classe Carro



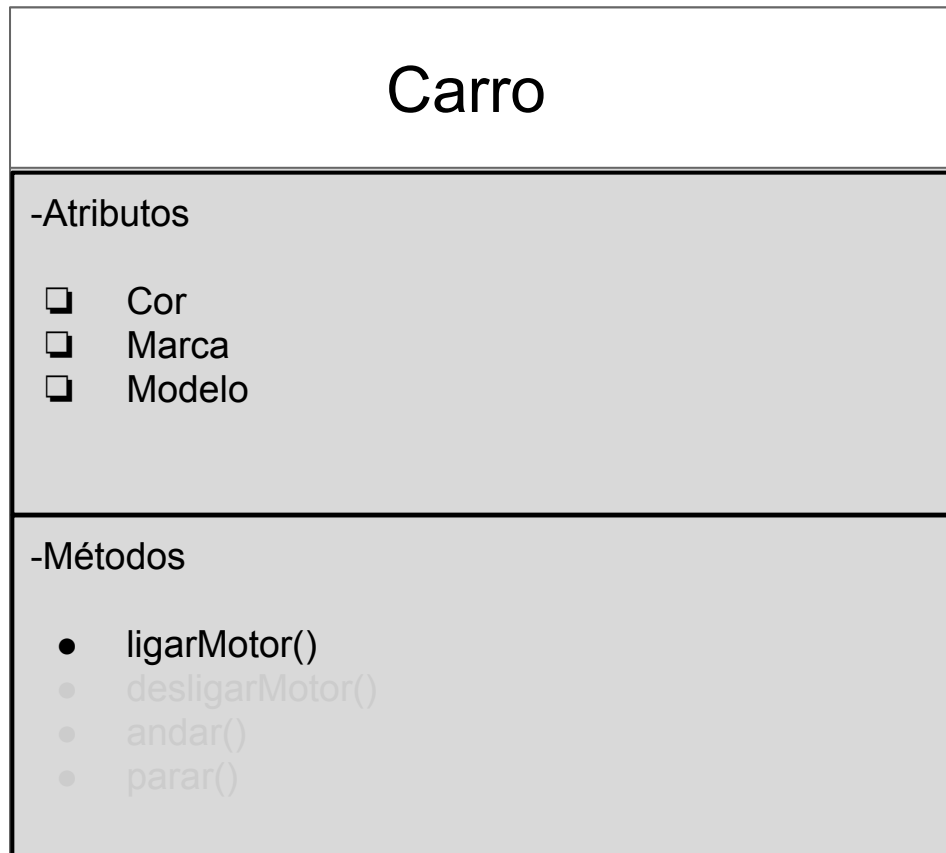
Classe Carro



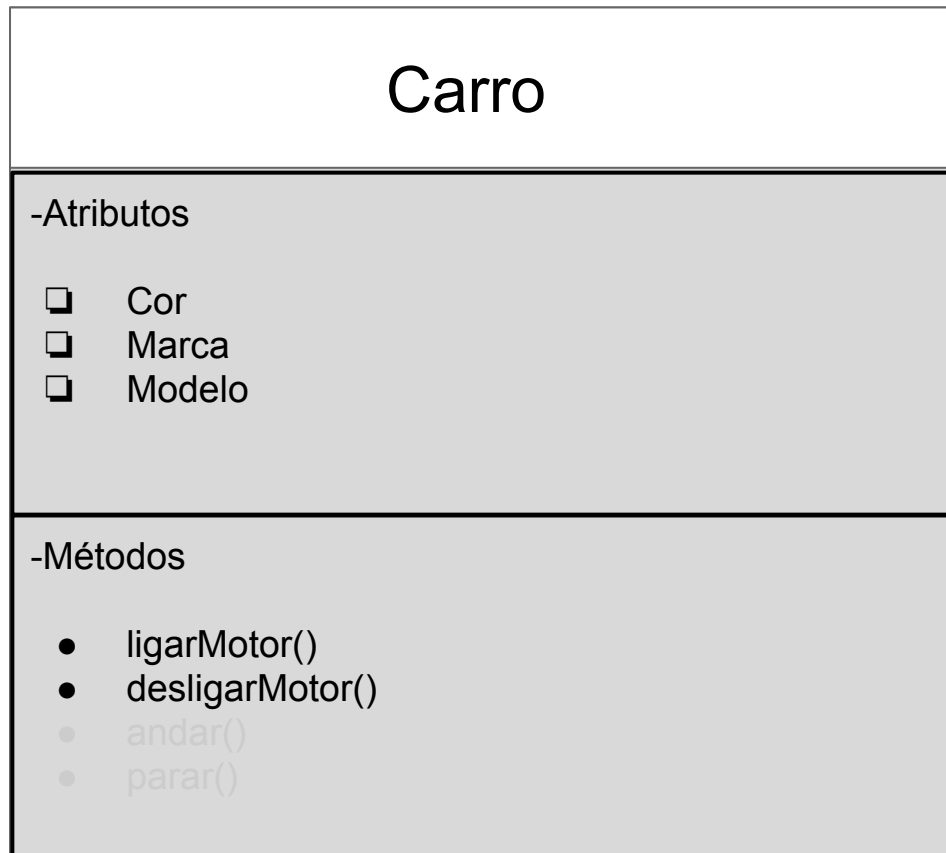
Classe Carro



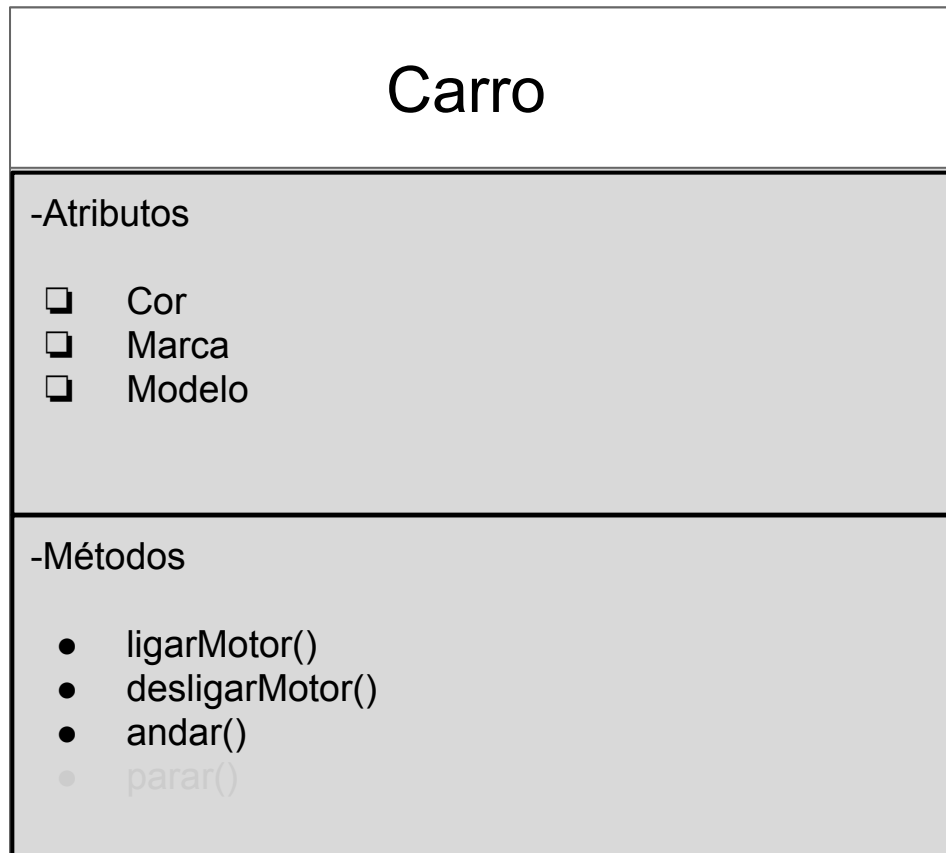
Classe Carro



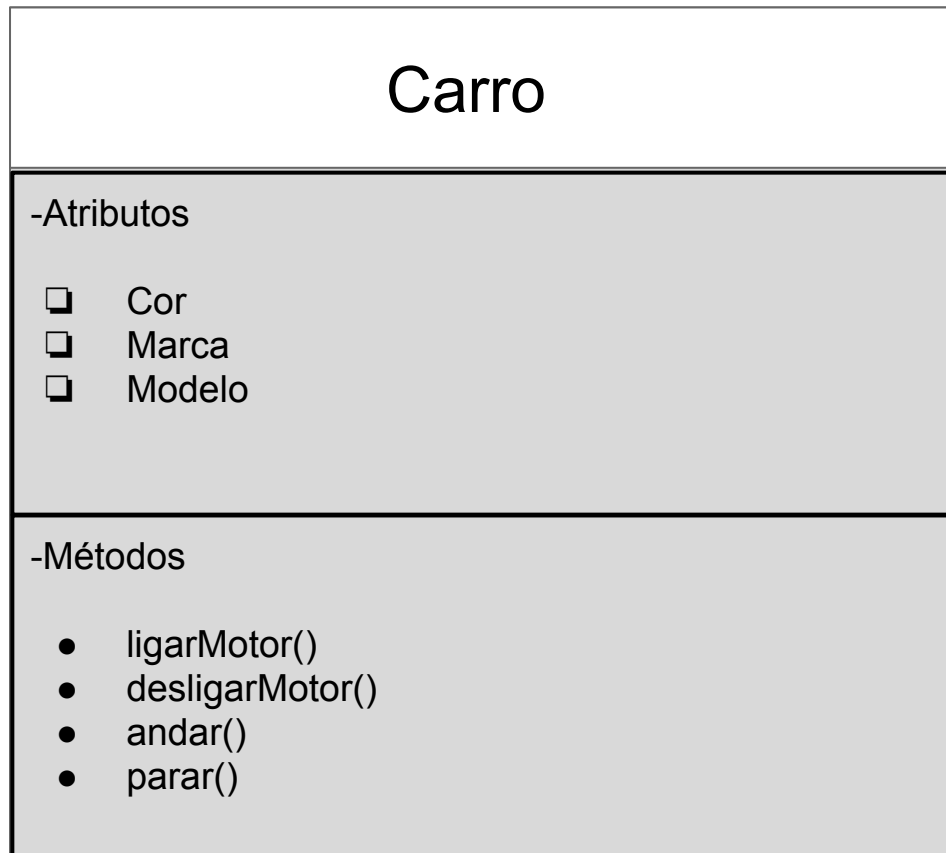
Classe Carro



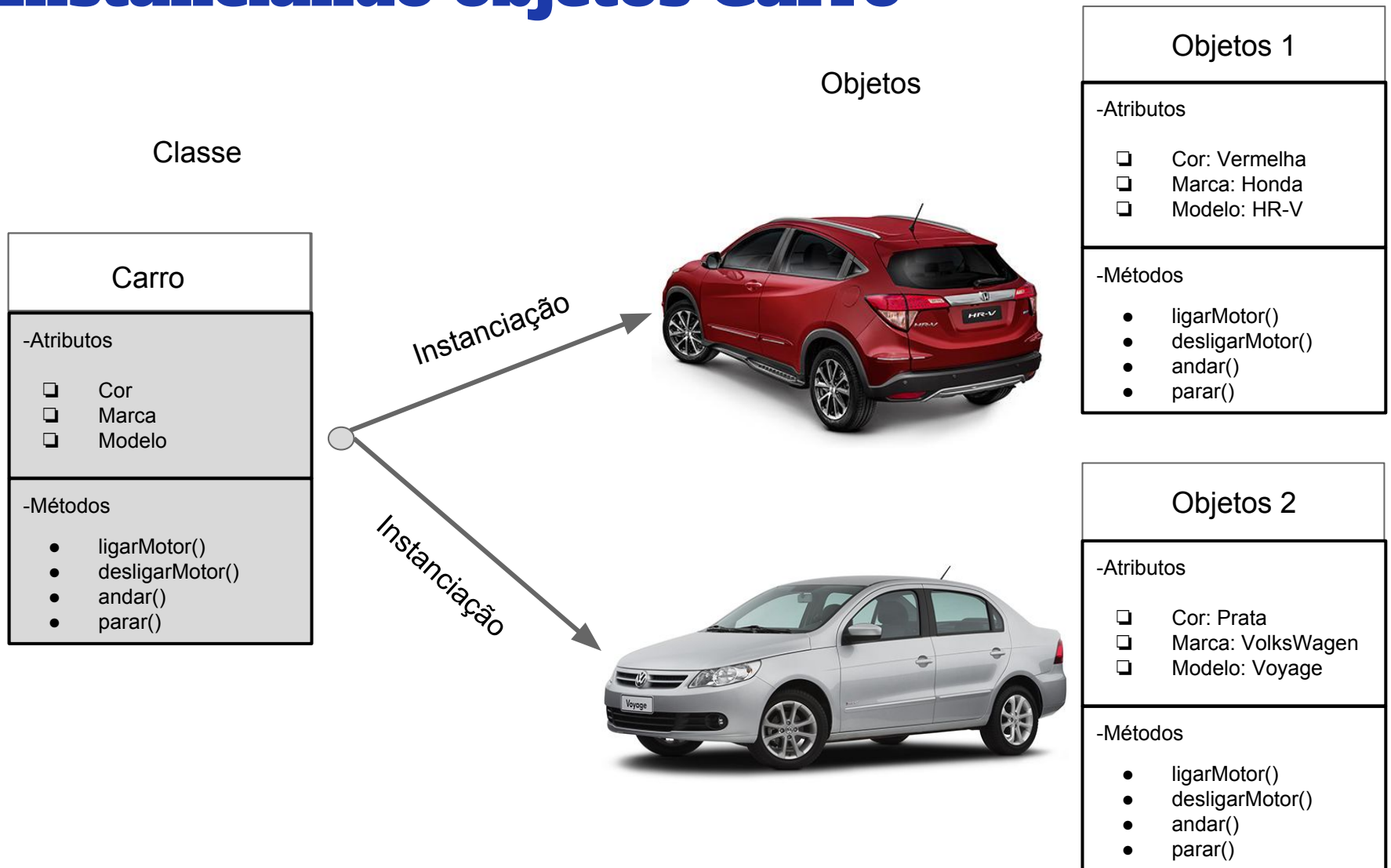
Classe Carro



Classe Carro



Instanciando objetos Carro



Classes em Python

- Como definir uma classe em Python?

```
class Nome:  
    atributo_1 = valor_1  
    atributo_2 = valor_2  
    atributo_3 = valor_3  
  
    def metodo_1(self):  
        #faz algo aqui
```

- Uma nova instância da classe pode ser criada a partir da chamada `var = Nome();`
- 'var' irá armazenar a instância criada. Ou seja, o objeto será salvo em var;

Classes em Python

- Nossa classe Carro em Python:

```
class Carro:
    cor = 'sem cor'
    marca = 'sem marca'
    modelo = 'sem modelo'
    ano = 2010
    km_rodados = 0

    def detalhes(self):
        print 'cor:', self.cor
        print 'marca:', self.marca
        print 'modelo:', self.modelo
        print 'ano:', self.ano
        print 'km rodados:', self.km_rodados
```

Criando objetos em Python

```
>>>car_1 = Carro() #Instância o objeto da classe Carro na variável 'car_1'
>>>car_1.cor = 'Vermelho'
>>>car_1.marca = 'Honda'
>>>car_1.modelo = 'HR-V'
>>>car_1.ano = 2016
>>>car_1.detalhes() #Chama o método 'detalhes' implementado na classe Carro
```

cor: Vermelho

marca: Honda

modelo: HR-V

ano: 2016

km_rodados: 0

Criando objetos em Python

```
>>>car_2 = Carro() #Instância o objeto da classe Carro na variável 'car_2'  
>>>car_2.cor = 'Prata'  
>>>car_2.marca = 'Volkswagen'  
>>>car_2.modelo = 'Voyage'  
>>>car_2.ano = 2014  
>>>car_2.km_rodados = 3000  
>>>car_2.detalhes() #Chama o método 'detalhes' implementado na classe Carro
```

cor: Prata

marca: Volkswagen

modelo: Voyage

ano: 2014

km rodados: 3000

Criando objetos em Python

```
>>>Carro.detalhes(car_1)
```

```
cor: Vermelho
```

```
marca: Honda
```

```
modelo: HR-V
```

```
ano: 2016
```

```
km rodados: 0
```

```
>>>Carro.detalhes(car_2)
```

```
cor: Prata
```

```
marca: Volkswagen
```

```
modelo: Voyage
```

```
ano: 2014
```

```
km rodados: 3000
```

Criando novos Métodos

```
class Carro:
    cor = 'sem cor'
    marca = 'sem marca'
    modelo = 'sem modelo'
    ano = 2010
    km_rodados = 0

    def detalhes(self):
        print 'cor:', self.cor
        print 'marca:', self.marca
        print 'modelo:', self.modelo
        print 'ano:', self.ano
        print 'km rodados:', self.km_rodados

    def adiciona_km_rodados(self, km):
        self.km_rodados = self.km_rodados + km
```

Criando novos Métodos

```
>>>car_1 = Carro() #Instância o objeto da classe Carro na variável 'car_1'  
>>>car_1.cor = 'Vermelho'  
>>>car_1.marca = 'Honda'  
>>>car_1.modelo = 'HR-V'  
>>>car_1.ano = 2016  
>>>car_1.detalhes() #Chama o método 'detalhes' implementado na classe Carro
```

cor: Vermelho

marca: Honda

modelo: HR-V

ano: 2016

km_rodados: 0

Criando novos Métodos

```
>>>car_1.adiciona_km_rodados(450)
```

```
>>>car_1.detalhes()
```

```
cor: Vermelho
```

```
marca: Honda
```

```
modelo: HR-V
```

```
ano: 2016
```

```
km_rodados: 450
```

Quando usar self

- **self** é uma variável que referencia um determinado objeto da classe;
- Todo método de uma classe recebe **self** como primeiro parâmetro;
- Tal parâmetro indica qual objeto está executando aquele método;
- **self.** deve preceder um atributo da classe dentro de métodos;

Quando usar self

- **self.** deve preceder os atributos da classe dentro de **métodos**;
- Variáveis (de métodos) que não possuem **self.** são consideradas locais e deixam de existir após a execução do método.

Quando usar self

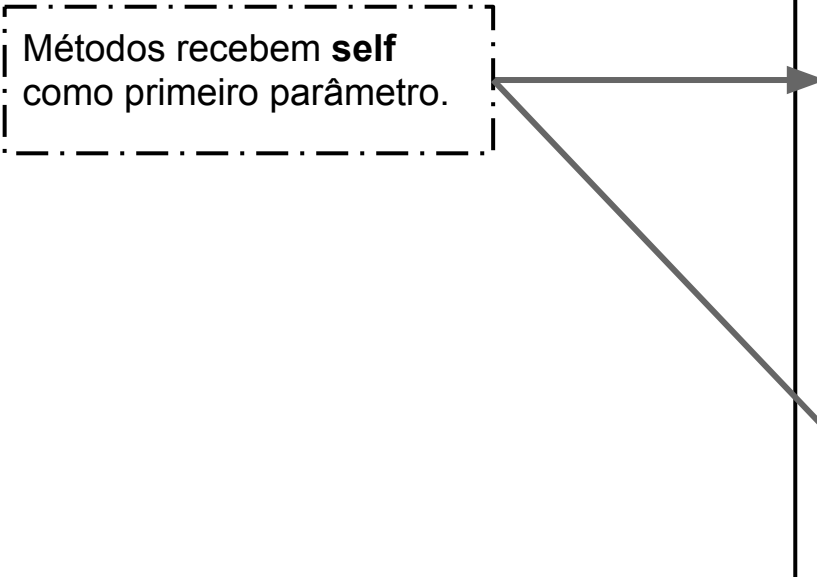
```
class Carro:
    cor = 'sem cor'
    marca = 'sem marca'
    modelo = 'sem modelo'
    ano = 2010
    km_rodados = 0

    def detalhes(self):
        print 'cor:', self.cor
        print 'marca:', self.marca
        print 'modelo:', self.modelo
        print 'ano:', self.ano
        print 'km rodados:', self.km_rodados

    def adiciona_km_rodados(self, km):
        self.km_rodados = self.km_rodados + km
```

Quando usar self

Métodos recebem **self**
como primeiro parâmetro.



```
class Carro:
    cor = 'sem cor'
    marca = 'sem marca'
    modelo = 'sem modelo'
    ano = 2010
    km_rodados = 0

    def detalhes(self):
        print 'cor:', self.cor
        print 'marca:', self.marca
        print 'modelo:', self.modelo
        print 'ano:', self.ano
        print 'km rodados:', self.km_rodados

    def adiciona_km_rodados(self, km):
        self.km_rodados = self.km_rodados + km
```

Quando usar self

- Atributos da classe.
- Fora dos métodos.

Não possuem **self** e todos os objetos oriundos dessa classe possuem o mesmo valor.

```
class Carro:
```

```
    cor = 'sem cor'  
    marca = 'sem marca'  
    modelo = 'sem modelo'  
    ano = 2010  
    km_rodados = 0
```

```
    def detalhes(self):
```

```
        print 'cor:' self.cor  
        print 'marca:' self.marca  
        print 'modelo:' self.modelo  
        print 'ano:' self.ano  
        print 'km rodados:' self.km_rodados
```

```
    def adiciona_km_rodados(self, km):
```

```
        self.km_rodados = self.km_rodados + km
```

- Atributos da classe.
- Dentro de métodos.

Possuem **self** e alteram ou carregam os valores dos atributos criados fora dos métodos.

Quando usar self

- Variável local do método **detalhes**;
- Não possui **self.**;
- Só existe durante a execução do método **detalhes**;

```
class Carro:  
    cor = 'sem cor'  
    marca = 'sem marca'  
    modelo = 'sem modelo'  
    ano = 2010  
    km_rodados = 0  
  
    def detalhes(self):  
        print 'cor:', self.cor  
        print 'marca:', self.marca  
        print 'modelo:', self.modelo  
        print 'ano:', self.ano  
        print 'km rodados:', self.km_rodados  
        passageiro = True  
  
    def adiciona_km_rodados(self, km):  
        self.km_rodados = self.km_rodados + km
```

Quando usar self

```
class Carro:
    cor = 'sem cor'
    marca = 'sem marca'
    modelo = 'sem modelo'
    ano = 2010
    km_rodados = 0

    def detalhes(self):
        print 'cor:', self.cor
        print 'marca:', self.marca
        print 'modelo:', self.modelo
        print 'ano:', self.ano
        print 'km rodados:', self.km_rodados

    def adiciona_km_rodados(self, km):
        self.km_rodados = self.km_rodados + km
```

OO em Python

- Em Python, uma Classe é um tipo de dado;
- Todo valor pertence a alguma classe;
- Exemplos:
 - 4 pertence à classe 'int'
 - 2.5 pertence à classe 'float'
 - 'huehuehuebr' pertence à classe 'str' (string)

Classes em Python

```
>>> help(int)
```

Help on class int in module `__builtin__`:

```
class int(object)
```

```
| int(x=0) -> int or long
```

```
| int(x, base=10) -> int or long
```

```
|
```

```
| Convert a number or string to an integer, or return 0 if no arguments  
| are given. If x is floating point, the conversion truncates towards zero.  
| If x is outside the integer range, the function returns a long instead.
```

```
| Methods defined here:
```

```
|
```

```
| __abs__(...)
```

```
| x.__abs__() <==> abs(x)
```

```
|
```

```
| __add__(...)
```

```
| x.__add__(y) <==> x+y
```

...

Classes em Python

```
>>> help(float)
```

Help on class float in module `__builtin__`:

```
class float(object)
```

```
| float(x) -> floating point number
```

```
|
```

```
| Convert a string or number to a floating point number, if possible.
```

```
|
```

```
| Methods defined here:
```

```
|
```

```
| __abs__(...)
```

```
| x.__abs__() <==> abs(x)
```

```
|
```

```
| __add__(...)
```

```
| x.__add__(y) <==> x+y
```

...

Exercício

1. Implementar os métodos abaixo para a classe Carro:
 - a. ligarMotor
 - b. desligarMotor
 - c. andar
 - d. parar

2. Criar atributos para:
 - a. Status do motor (ligado/desligado)
 - b. Status do movimento do carro (andando/parado)

3. Criar métodos para informar (exibir na tela) o status acima.

Construtores

- Métodos importantes em Classes;
- São executados assim que o Objeto é instanciado;
- Em Python, possui a seguinte estrutura:

```
def __init__(self):
```

- Comumente utilizados para inicialização de atributos.

Construtores

- Nossa classe Carro em Python **SEM** construtor:

```
class Carro:
    cor = 'sem cor'
    marca = 'sem marca'
    modelo = 'sem modelo'
    ano = 2010
    km_rodados = 0

    def detalhes(self):
        print 'cor:', self.cor
        print 'marca:', self.marca
        print 'modelo:', self.modelo
        print 'ano:', self.ano
        print 'km rodados:', self.km_rodados
```

Construtores

```
>>>car_1 = Carro() #Instância o objeto da classe Carro na variável 'car_1'
>>>car_1.cor = 'Vermelho'
>>>car_1.marca = 'Honda'
>>>car_1.modelo = 'HR-V'
>>>car_1.ano = 2016
>>>car_1.detalhes() #Chama o método 'detalhes' implementado na classe Carro
```

```
cor: Vermelho
```

```
marca: Honda
```

```
modelo: HR-V
```

```
ano: 2016
```

```
km_rodados: 0
```

Construtores

- Nossa classe Carro em Python **COM** construtor:

```
class Carro:
    def __init__(self, cor, marca, modelo, ano, km_rodados):
        self.cor = cor
        self.marca = marca
        self.modelo = modelo
        self.ano = ano
        self.km_rodados = km_rodados

    def detalhes(self):
        print 'cor:', self.cor
        print 'marca:', self.marca
        print 'modelo:', self.modelo
        print 'ano:', self.ano
        print 'km rodados:', self.km_rodados
```

Construtores

```
>>> from Carro_constructor import Carro
>>> car = Carro('azul', 'Honda', 'HR-V', 2016, 2000)
>>> car.detalhes()
```

cor: azul

marca: Honda

modelo: HR-V

ano: 2016

km rodados: 2000

Documentação

- Documentar Classes e Métodos
 - Necessário para você e outros que irão utilizar o código
- Função `help()` ensinada em Comp I
 - Exibe documentação de um método/classe;

```
>>> help(math.cos)
Help on built-in function cos in module math:
cos(...)
    cos(x)
        Return the cosine of x (measured in
radians).
```


Documentação

- Documentação em Python: **docstrings**

```
class Carro:  
    '''  
    Classe que representa um carro.  
    Cada carro possui:  
        -cor  
        -marca  
        -modelo  
        -ano  
        -km_rodados  
        -statusMotor  
        -statusMovimento  
    '''
```

Documentação

```
def andar(self):  
    '''Método que coloca o carro em movimento  
    Verifica antes se o carro está ligado ou desligado'''  
  
    if(self.statusMotor == True):  
        if(self.statusMovimento == True):  
            print 'O carro já está em movimento!'  
        else:  
            self.statusMovimento = True  
            print 'Carro em movimento!'  
    else:  
        print 'Necessário ligar o motor!'
```

Documentação

```
class Carro
```

```
| Classe que representa um carro.
```

```
| Cada carro possui:
```

```
|     -cor
```

```
|     -marca
```

```
|     -modelo
```

```
|     -ano
```

```
|     -km_rodados
```

```
|     -statusMotor
```

```
|     -statusMovimento
```

```
| Methods defined here:
```

```
| andar(self)
```

```
|     Método que coloca o carro em movimento
```

```
|     Verifica antes se o carro está ligado ou desligado
```

Encapsulamento

- Encapsulamento de dados é a proteção dos atributos e métodos de uma Classe;
- Seu objetivo é restringir o acesso direto à informação;
- Existem dois tipos de atributos em OO - Python:
 - Público
 - "Privado"

Encapsulamento

- Atributos públicos:
 - Podem ser acessados diretamente;
 - Não existe restrição quanto a escrita e leitura deles.

- Atributos privados:
 - São acessados via métodos;
 - Restrição de leitura e escrita aos dados de forma direta;

Obs.: Em Python os atributos não são realmente privados. Tal opção aparece apenas para informar ao programador que aquele determinado atributo não deve ser acessado diretamente.

Encapsulamento

```
class Carro:  
    #Atributo público  
    cor = 'azul'  
  
    #Atributo privado  
    __nomeProprietario = 'Brunno Goldstein'
```

Encapsulamento

- Se o atributo é privado, como acessar?
- Via métodos:

```
class Carro:
    #Atributo público
    cor = 'azul'
    #Atributo privado
    __nomeProprietario = 'Brunno Goldstein'

    def getNomeProprietario(self):
        return self.__nomeProprietario

    def setNomeProprietario(self, novoProprietario):
        self.__nomeProprietario = novoProprietario
```

Encapsulamento

- Métodos GET/SET:
 - Utilizados para acesso de leitura (GET) e escrita (SET) de atributos privados;
- Métodos definidos da seguinte forma:
 - `getNomeDoAtributo(self)`
 - `setNomeDoAtributo(self, novoNomeDoAtributo)`

Herança

- Técnica de OO para especialização;
- Classes passam a ser especializações das outras classes;
- Atributos e Métodos podem ser herdados de outra classe sem a necessidade de reimplementação;
- Ajuda a simplificar o código através do reuso.

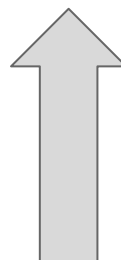
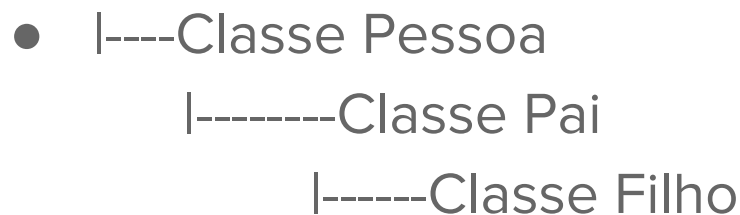
Herança

```
class Pessoa:
    def __init__(self, nome, idade, cpf, rg, endereco):
        self.nome = nome
        self.idade = idade
        self.__cpf = cpf
        self.__rg = rg
        self.__endereco = endereco
    ...

class Pai(Pessoa):
    ...
```

Herança - Construtores

- Python sobe na hierarquia até encontrar o primeiro construtor;



Sobe até encontrar o primeiro construtor

- Caso necessário, construtores podem ser chamados em cadeia.

Herança e Construtores

```
class Pessoa(object):
    def __init__(self, nome, tipo, endereco):
        self.nome = nome
        self.tipo = tipo
        self.endereco = endereco

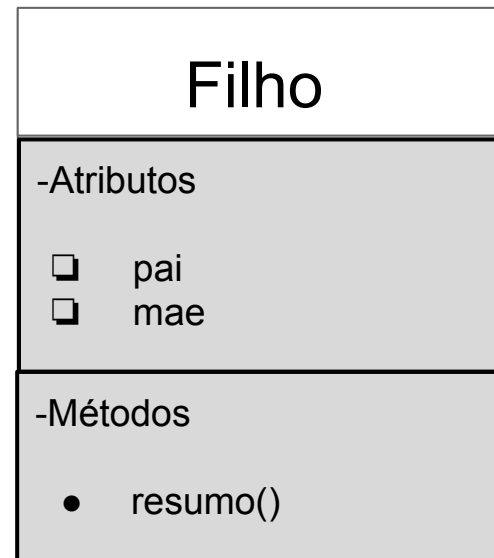
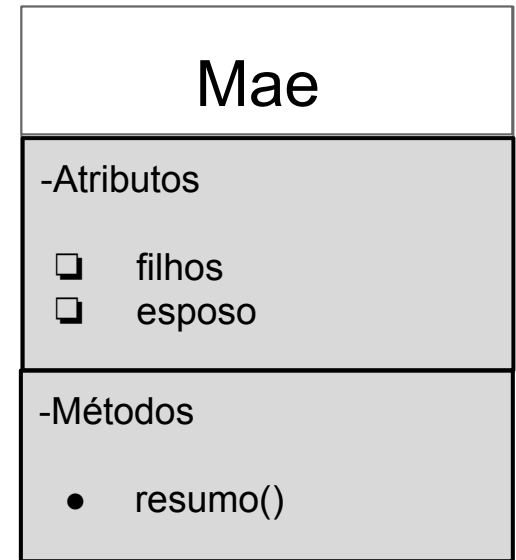
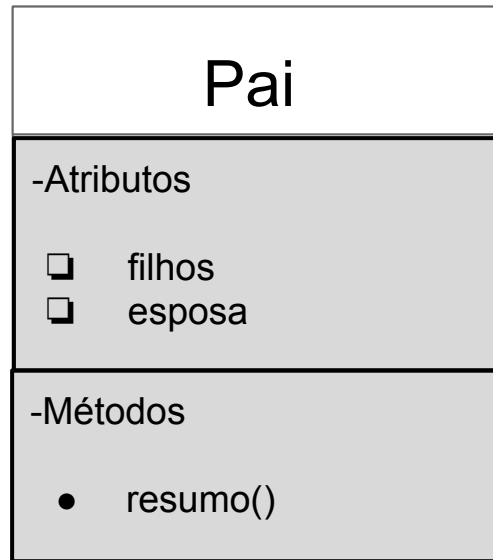
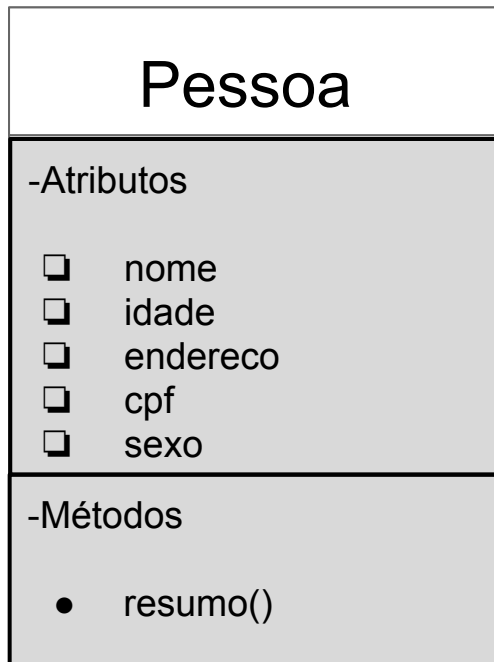
class Fisica(Pessoa):
    def __init__(self, nome, cpf, endereco):
        Pessoa.__init__(self, nome, 'fisica', endereco)
        self.__cpf = cpf

class Juridica(Pessoa):
    def __init__(self, nome, cnpj, endereco):
        Pessoa.__init__(self, nome, 'juridica', endereco)
        self.__cnpj = cnpj
```

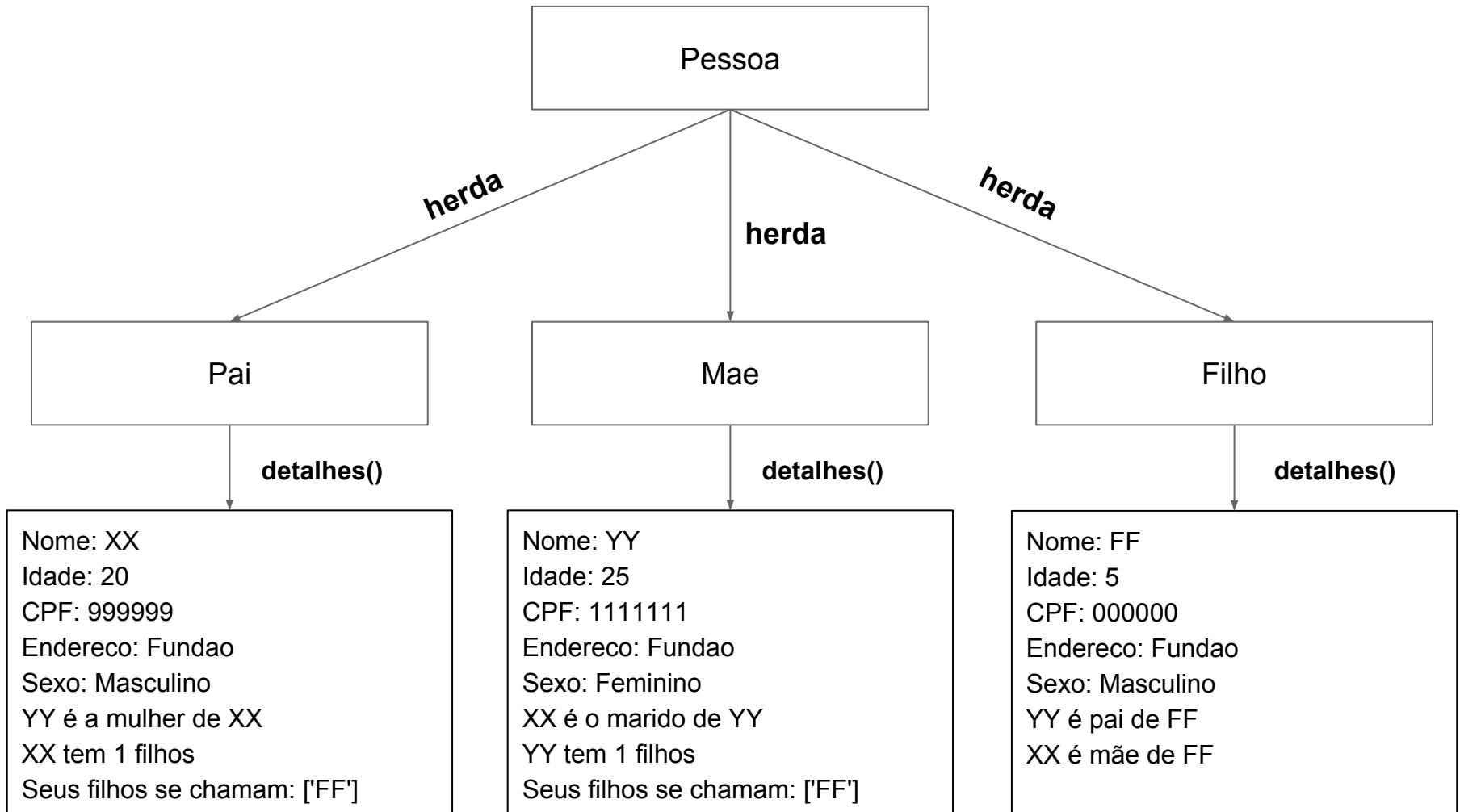
Exercício

1. Vamos modelar uma família com Pai, Mãe, Filhos;
2. Todos devem herdar da classe Pessoa;
3. Na classe Pai e Mãe, crie um método que irá adicionar objetos da classe Filhos;
4. Na classe Filho, crie um método que irá adicionar objetos Pai e Mãe;
5. Crie um método Resumo genérico na classe Pessoa que listará atributos do objeto;
6. Crie um método Resumo especializado para cada Classe: Pai, Mãe e Filhos.

Exercício



Exercício



Encoding

- Computadores só compreendem bits:
 - 0 ou 1
 - Ligado ou Desligado
 - Verdadeiro ou Falso
- Letras são traduzidas para uma sequência de bits;
- Cada letra é mapeada em uma tabela indicando qual sequência ela representa;
- Tal processo pode ser chamada de codificação (encoding);

```
01100010  01101001  01110100  01110011  
b          i          t          s
```


Encoding

- Diversas línguas + diversos caracteres = Várias tabelas;
- Foi criado então o Unicode;
- Tabela padrão que consegue traduzir todos os caracteres ;
- Unicodes são comumente chamados de UTF's
- UTF-8, UTF-16, UTF-32
- Para que o interpretador de Python consiga compreender caracteres especiais (ex.: acentuação). Adicionar o comentário abaixo na primeira linha do código:

```
# -*- coding: utf-8 -*-
```

Herança e classe **Object**

- Classe **object** foi introduzida na versão 2.2 do Python;
- Introduziu um novo estilo de classes para a linguagem;
- **Object** é uma classe molde;
- Classes pai devem herdar de **object** para ter acesso a métodos específicos;
- Na versão 3.3+, toda classe faz parte do novo estilo, não sendo necessário herdar de **object**.

Property - Get/Set

- Criados através do uso de anotações;
- Anotações são palavras restritas iniciadas com @;

```
@property
def nomeVariavelPrivada(self):
    return self.__nomeVariavelPrivada

@nomeVariavelPrivada.setter
def nomeVariavelPrivada(self, valor):
    self.__nomeVariavelPrivada = valor
```

Property - Get/Set

```
self.__cpf = cpf

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

@cpf.setter
def cpf(self, valor):
    self.__cpf = valor
```

OBS: A classe "pai" deve herdar de object para utilizar o @property !

Sobrecarga de Operadores

- Chamados também de métodos mágicos;
- Métodos são chamados usando operadores sobre os objetos ao invés do nome;
 - Ex.:
obj_1 + obj_2
obj_4 - obj_3
- Método mágico que já vimos: `__init()`

Sobrecarga de Operadores

❖ Lista de alguns métodos numéricos que podem ser sobrecarregados:

- `__add__`: Adição
 - $A+B$
- `__sub__`: Subtração
 - $A-B$
- `__mul__`: Multiplicação
 - $A*B$

- `__div__`: Divisão
 - A/B
- `__mod__`: Resto da divisão
 - $A\%B$
- `__abs__`: Absoluto
 - $|A|$

Sobrecarga de Operadores

❖ Lista de alguns métodos não numéricos que podem ser sobrecarregados:

- **__repr__**: Representação
 - Chamado quando o objeto é impresso

- **__str__**: Conversão para String
 - Chamado quando o objeto é impresso

Método **__repr__** é chamado se **__str__** não for especificado.

- **__repr__** utilizado por desenvolvedores
- **__str__** utilizado por usuários

Sobrecarga de Operadores

```
class Racional:
    def __init__(self, divisor, dividendo):
        self.divisor = divisor
        self.dividendo = dividendo

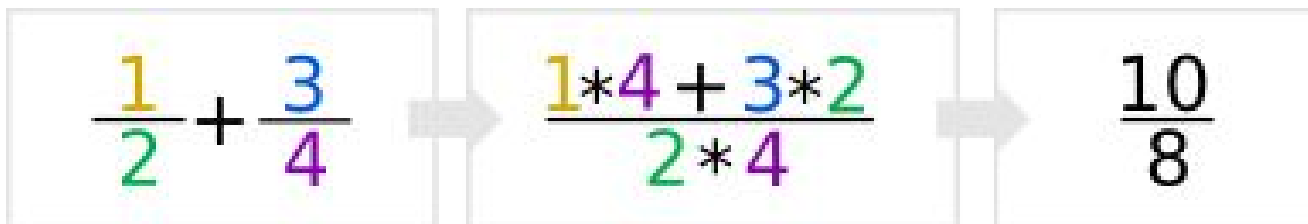
    def __str__(self):
        return str(self.divisor) + '/' + str(self.dividendo)

    def __mul__(self, outro):
        divisor = self.divisor*outro.divisor
        dividendo = self.dividendo*outro.dividendo
        return Racional(divisor, dividendo)

    def __add__(self, outro):
        divisor = self.divisor * outro.dividendo + outro.divisor * self.dividendo
        dividendo = self.dividendo * outro.dividendo
        return Racional(divisor, dividendo)
```


Sobrecarga de Operadores

```
>>> from Racional import *  
>>> a = Racional(1,2)  
>>> b = Racional(3,4)  
>>> c = a+b  
>>> print c  
10/8
```



Fonte: <http://eupodiatamatando.com/2007/04/09/sobrecarga-de-operadores-em-python/>

Exercício

1. Utilizando a funcionalidade **property**, implementar as classes Pessoa, Física e Jurídica.
2. Criar classe Matriz com os seguintes atributos:
 - a. dimensao
 - b. data
 - c. tipo
3. Implementar método mágico para somar objetos Matriz;
4. Implementar método mágico para multiplicar objetos Matriz;
5. Implementar método mágico `__str__` para exibir os dados da matriz;

Obs.: Os métodos deverão verificar as dimensões das matrizes antes de realizar as operações.

Herança Múltipla

- Classes podem herdar de uma ou mais classes:
 - Ex.: `class Elefante(Animal, Mamifero)`
- Classe derivada herda todos os atributos e métodos de ambas as classes;
- Se ambas as classes base possuem um atributo/método com mesmo nome, aquela citada primeiro prevalece :
 - No exemplo acima, se `Animal` e `Mamífero` possuem um atributo `nome`, então `Elefante.nome` se refere ao que foi herdado de `Animal`.

Polimorfismo

- *"Qualidade ou estado de ser capaz de assumir diferentes formas";*
- Habilidade de um objeto de adaptar seu código ao tipo de objeto que está processando.
 - Método **len(obj)**:

"Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set)."

Polimorfismo

```
class Room:
    def __init__(self, door):
        self.door = door

    def open(self):
        self.door.open()

    def close(self):
        self.door.close()

    def is_open(self):
        return self.door.is_open()
```

Polimorfismo

```
class Door:
    def __init__(self):
        self.status = "closed"

    def open(self):
        self.status = "open"

    def close(self):
        self.status = "closed"

    def is_open(self):
        return self.status == "open"
```

```
class BooleanDoor:
    def __init__(self):
        self.status = True

    def open(self):
        self.status = True

    def close(self):
        self.status = False

    def is_open(self):
        return self.status
```

Polimorfismo

```
>>> door = Door()
>>> bool_door = BooleanDoor()
>>> room = Room(door)
>>> bool_room = Room(bool_door)

>>> room.open()
>>> room.is_open()
True
>>> room.close()
>>> room.is_open()
False

>>> bool_room.open()
>>> bool_room.is_open()
True
>>> bool_room.close()
>>> bool_room.is_open()
False
```

Polimorfismo

- Ambas as classes representam uma porta que pode estar aberta ou fechada;
- Porém, representam utilizando "tipos" diferentes: Strings e Booleanos;
- Desconsiderando o fato anterior, as duas classes funcionam da mesma forma;
- Portanto, as duas podem ser utilizadas para implementar uma porta.