

# LINGUAGENS DE PROGRAMAÇÃO

Engenharia Informática

# Polimorfismo

- Define a possibilidade de manusear objetos distintos dependendo do seu tipo ou classe.
- Há um contrato que é definido e que tem de ser respeitado na interface das classes
- Polimorfismo com uma função
- Polimorfismo com classes abstratas

# Polimorfismo – com função

```
class Bear:  
    def sound(self):  
        print "Groarr"
```

```
class Dog:  
    def sound(self):  
        print "Woof woof!"
```

```
def makeSound(animalType):  
    animalType.sound()
```

animalType tem de implementar:  
sound()

```
bearObj = Bear()  
dogObj = Dog()
```

```
makeSound(bearObj)  
makeSound(dogObj)
```

# Herança - Inheritance

- Permite que se crie primeiro uma classe geral (superclasse) que depois é “extendida” para uma mais específica (subclasse).
- A subclasse herda o acesso aos atributos e métodos e ainda permite adicionar atributos e métodos.
  - *Apenas os não privados.*

# Herança - Inheritance

```
class Veiculo: ←———— Classe Super  
    pass
```

```
class Carro(Veiculo): ←———— Sub Classe  
    pass
```

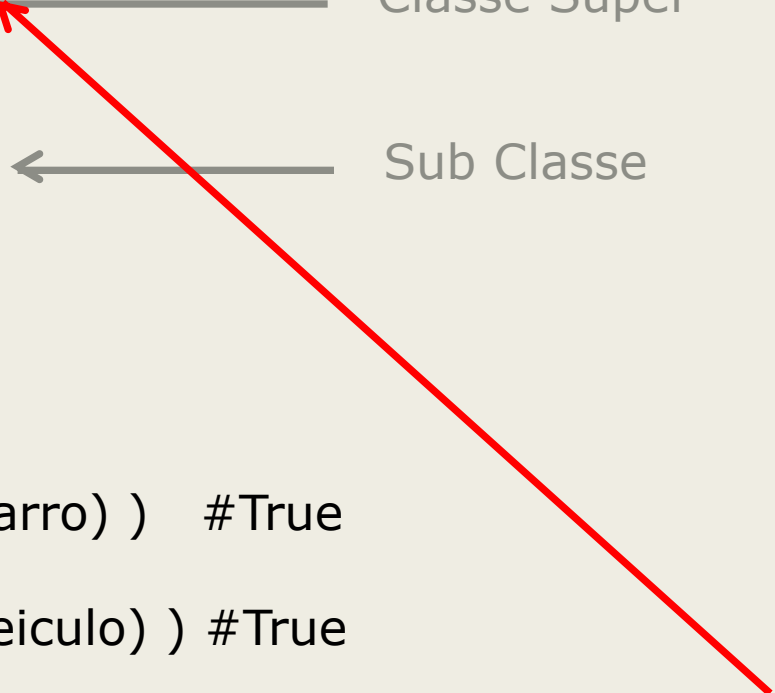
```
c = Carro()
```

```
print( isinstance(c, Carro) ) #True
```

```
print( isinstance(c, Veiculo) ) #True
```

```
print( isinstance(c, object) ) #True
```

```
class Veiculo(object):  
    pass
```



super() – refere-se à classe Pai

Métodos herdados  
Públicos

```
class Carro(Veiculo):
```

```
    def __init__(self, nome, cor, modelo):  
        super().__init__(nome, cor)  
        self.__modelo = modelo
```

```
    def getDescricao(self):  
        return self.getNome() + " " + self.__modelo + " " + self.getCor()
```

```
c = Carro("Ford Mustang", "Vermelho", "GT350")
```

```
print(c.getDescricao())  
print(c.getNome())
```

```
class Veiculo:
```

```
    def __init__(self, nome, cor):  
        self.__nome = nome  
        self.__cor = cor
```

```
    def getCor(self):  
        return self.__cor
```

```
    def getNome(self):  
        return self.__nome
```

```
    def anda(self):  
        print("Veiculo a andar...")
```

# Override

```
class Carro(Veiculo):
```

```
    def __init__(self, nome, cor, modelo):  
        super().__init__(nome, cor)  
        self.__modelo = modelo
```

```
    def anda(self): #Overriding  
        super().anda()  
        print("Vruuuuum...")
```

```
c = Carro("Ford Mustang", "Vermelho", "GT350")  
c.anda()
```

```
class Veiculo:
```

```
    def __init__(self, nome, cor):  
        self.__nome = nome  
        self.__cor = cor
```

```
    def getCor(self):  
        return self.__cor
```

```
    def getNome(self):  
        return self.__nome
```

```
    def anda(self):  
        print("Veiculo a andar...")
```

# Override – object methods

- O object é a classe raiz em python
- Mas já possuí alguns métodos nativos

```
class Veiculo:  
    pass
```

```
v=Veiculo()
```

```
print(v)
```

```
#Imprime: <__main__.Veiculo object at 0x10f6923c8>
```

```
print (dir(v))
```

```
#Imprime: ['__doc__', '__str__', '__eq__', '__init__', ... ]
```



```
class Veiculo:
```

```
    #Override de __str__()
```

```
    def __str__(self):
```

```
        return "Sou um veiculo da classe Veiculo"
```

```
v=Veiculo()
```

```
print(v)
```

```
#Imprime: Sou um veiculo da classe Veiculo
```

A função print recorre ao `__str__()`  
Para imprimir o conteúdo do objecto

A diagram consisting of two red arrows. The first arrow starts at the `print(v)` line and points to the `def __str__(self):` line. The second arrow starts at the `return` statement inside the `__str__` method and points back to the `print(v)` line, indicating the return value being passed to the print function.

# Polimorfismo - Classes Abstract

```
class VeiculoAbstrato:
```

```
    def __init__(self, nome, cor):  
        self.__nome = nome  
        self.__cor = cor
```

```
    def getCor(self):  
        return self.__cor
```

```
    def getNome(self):  
        return self.__nome
```

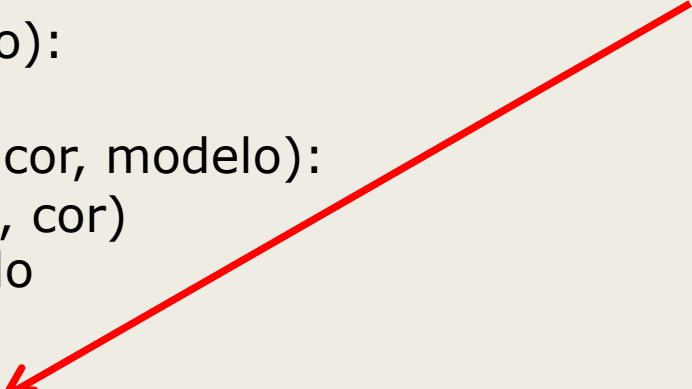
```
    def __str__(self):  
        return self.getDescricao()
```

Pode estar, ou não.

```
    def getDescricao(self):  
        raise NotImplementedError("Subclass tem de implementar getDescricao()")
```

# Classes Abstract

Implementação do método  
Abstrato



```
class Carro(VeiculoAbstrato):
```

```
    def __init__(self, nome, cor, modelo):  
        super().__init__(nome, cor)  
        self.__modelo = modelo
```

```
    def getDescricao(self):  
        return self.getNome() + " " + self.__modelo + " " + self.getCor()
```

```
c = Carro("Ford Mustang", "Vermelho", "GT350")  
print(c)  
#Imprime Ford Mustang GT350 Vermelho
```

# Classes Abstract

- Python não possui classes abstratas nativamente.
- O que vimos foi uma forma de imitar o funcionamento abstrato.
- Há ferramentas mais avançadas para gerir as classes abstratas em Python:

```
from abc import ABC, abstractmethod
```

```
class AbstractClass(ABC):  
    @abstractmethod  
    def foo(self):  
        pass
```

```
class SubClasse(AbstractClass):  
    pass
```

```
t=SubClasse()
```



Imprime:  
TypeError: Can't instantiate abstract class  
SubClasse with abstract methods foo

# Fim

- Questões?