

LINGUAGENS DE PROGRAMAÇÃO

Engenharia Informática

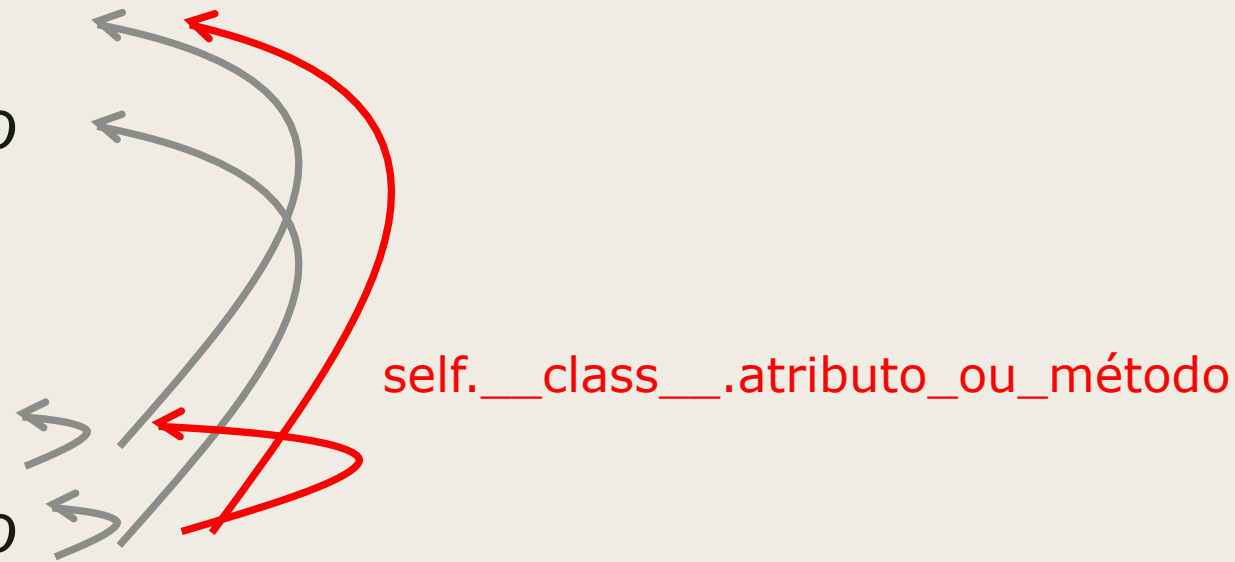
... na última aula.

■ Atributos

- *da Classe*
- *do Objecto*

■ Métodos

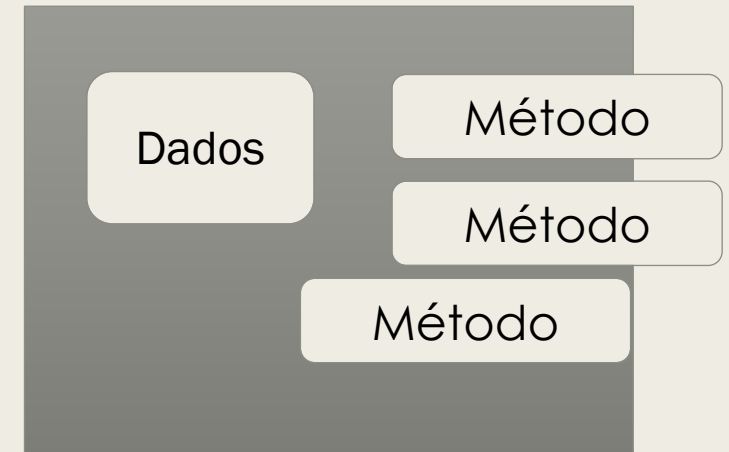
- *da Classe*
- *do Objecto*
- *Estáticos (como uma função externa à classe)*



Encapsulamento

- Vamos encapsular, enclausurar, blindar, proteger... o Objecto.
- Esconder os detalhes e criar abstração
 - *Podemos querer que o utilizador do objecto não tenha acesso aos detalhes;*
 - *permite alterações futuros sem que a interface disponibilizada seja alterada;*
 - por outras palavras, o código que utiliza o objecto mantém-se igual.

```
class Pizza():  
    def __init__(self, nome, lista):  
        self.nome = nome  
        self.lista = lista  
  
    def upperize(self):  
        return self.nome.upper()
```



- Não podemos dizer que os atributos/dados estejam protegidos.

```
margarita = Pizza("Margarita", ["queijo", "tomate"])  
margarita.nome = "Estou a mudar o nome."
```

- Podem existir atributos que não queremos disponibilizados fora do encapsulamento.
- O mesmo se aplica a métodos.

como proteger/esconder

■ Em Java:

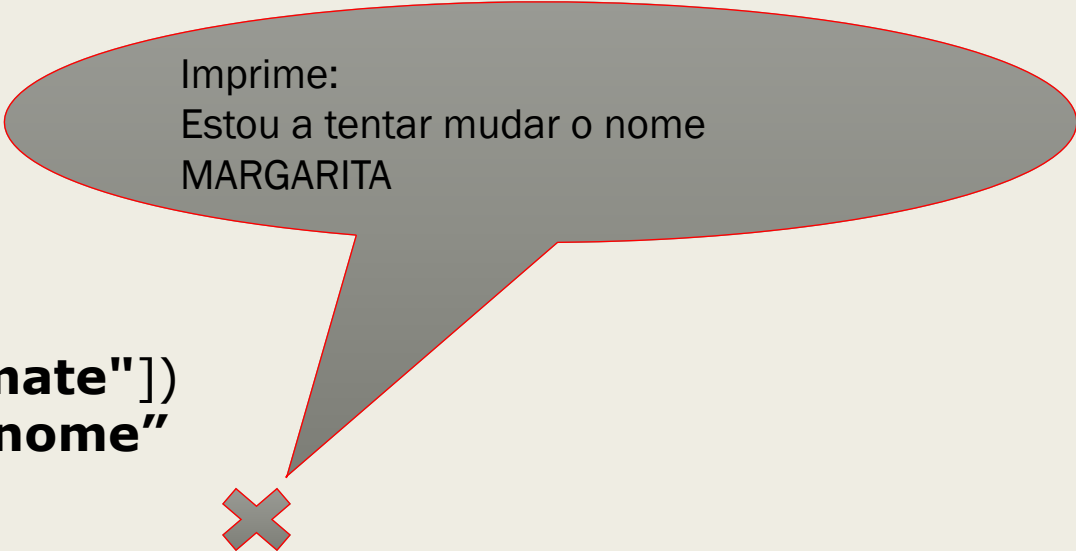
– *public* | *private* | *protected*

■ Em Python

– *public* | *private* (mais ou menos... já lá vamos!)

```
class Pizza():  
    def __init__(self, nome, lista):  
        self.__nome = nome  
        self.__lista = lista  
  
    def upperize(self):  
        return self.__nome.upper()
```

```
margarita = Pizza("Margarita", ["queijo", "tomate"])  
margarita.__nome = "Estou a tentar mudar o nome"  
print(margarita.__nome)  
print(margarita.upperize())
```



Imprime:
Estou a tentar mudar o nome
MARGARITA

- Quando se usa o prefixo `__` (2 x underscore) num atributo ou método o python esconde o acesso ao atributo/método a partir de fora.
 - No exemplo anterior, em:
margarita.__nome = "Estou a tentar mudar o nome"
O que estamos a fazer mesmo é criar um novo atributo chamado `__nome`.
 - O verdadeiro `self.__nome` está escondido – privado.



Confuso?

Como é que o python esconde?

- Atributos e métodos com prefixo `__` são renomeados juntanto o prefixo com o nome da classe onde está o atributo_NomeDaClasse
 - Ex: Se “`__nome`” é um atributo privado na classes “`Pizza`”, então é renomeado para “`_Pizza__nome`”

```
margarita = Pizza("Margarita", ["queijo", "tomate"])
margarita._Pizza__nome = "Estou a mudar o nome"
print(margarita.upperize())
```



Daniel Stori {turnoff.us}

- Resumindo...

- Em Python:

- *Os conteúdos privados protegem acessos inadvertidos, mas não protegem acessos intencionais.*

Getter and Setter



- Métodos que permitem ler (getter) ou escrever (setter) num atributo privado.

```
class Pizza():  
    def __init__(self, nome, lista):  
        self.__nome = nome  
        self.__lista = lista  
  
    def setNome(self, nome):  
        self.__nome = nome  
  
    def getNome(self):  
        return self.__nome
```

- Setter e Getter permitem definir se queremos dar acesso de leitura ou de escrita ou ambos.
 - *Assumindo que em Python atributos private são mesmo protegidos.*
- Ajudam a definir os interface de interação com o objecto.
 - *Acção!!! Verbos... métodos!*
 - *Há um método para escrever o nome.*
 - *Há um método para ler o nome.*
- Permitem programação defensiva com verificações/validações
 - **def** setNome(self, nome):
 if len(nome)>2:
 self.__nome = nome

Atribuição com pré-validação

Property

- Os setters e getters ajudam a definir a fronteira de como manipular os dados privados da classe.
- Assim, passamos de:
 - *margarita.nome*
margarita.get_nome() 
 - *margarita.nome="margherita"*
margarita.set_nome("margherita") 
 - *Not the smartest pythonic move, though!*

Property

- E se conseguíssemos manter a forma de acesso anterior (`margarita.nome`) juntamente com as validações dos setters e getters?
- O property permite criar variáveis “virtuais” que utilizam funções de set e get.

Property

De fora, não há acesso
Direto a `__nome`

O Property "nome" permite
acesso de fora e acede
indiretamente a `__nome`

```
class Pizza():
    def __init__(self, name, lista):
        self.__nome = name
        self.__lista = lista

    def setNome(self, name):
        if len(name) > 2:
            self.__nome = name

    def getNome(self):
        return self.__nome

    nome = property(getNome, setNome)

margarita = Pizza("Margarita", [])
print(margarita.nome)

margarita.nome = "Margherita"
print(margarita.nome)
```

@property

- Outra forma de definir o mesmo property:

```
class Pizza():  
    def __init__(self, name, lista):  
        self.__nome = name  
        self.__lista = lista
```

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

```
@nome.setter  
def nome(self, name):  
    if len(name) > 2:  
        self.__nome = name
```

■ Questões?