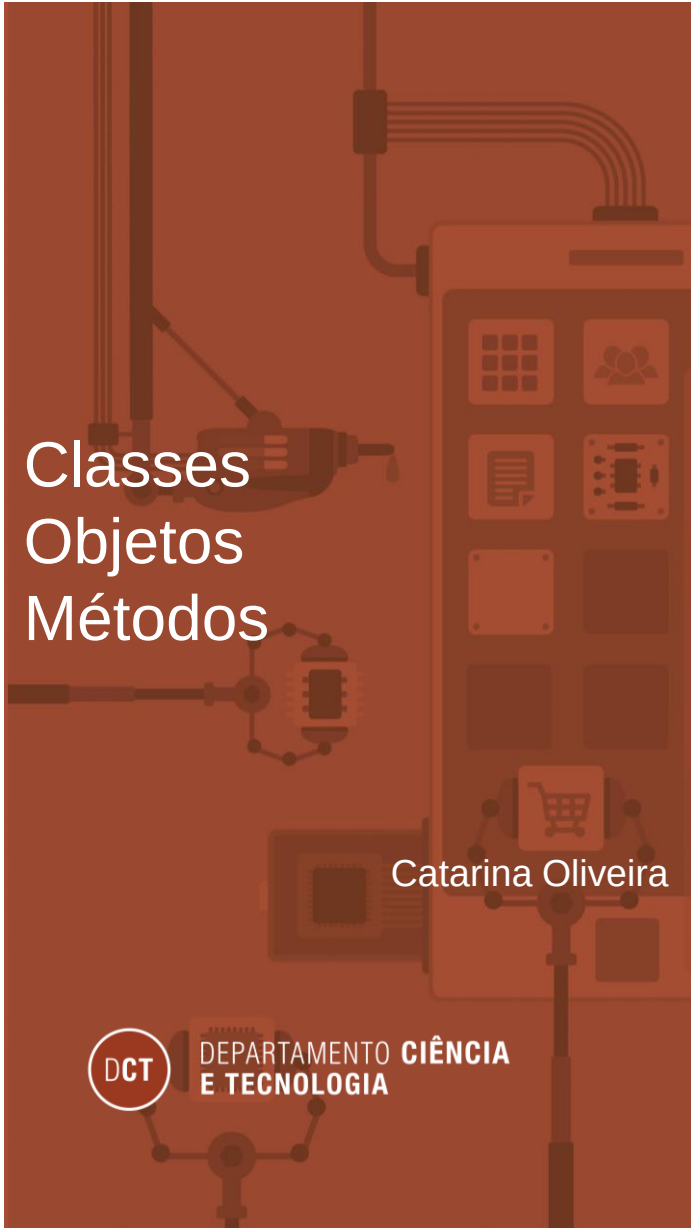


UPT UNIVERSIDADE PORTUCALENSE

CONTEÚDO

1. Classe, _ _init_ _ e self
2. Instância
3. _ _str_ _
4. Variáveis privadas
5. Variáveis de instância e de classe
6. Métodos de instância
7. Métodos de classe
8. Métodos estáticos e Property Decorator @staticmethod
9. Getters e Property Decorator @property
10. Setters e Property Decorator @****.setter
11. Deleters e Property Decorator @****.deleter
12. Herança e super()

Classe, _ _init_ _ e self

Classe

- Estrutura que permite agrupar dados e métodos
- “Molde” de instâncias

- Exemplo: uma empresa pretende representar os seus funcionários recorrendo a uma classe Python. Os dados guardados para cada funcionário são:
  - Primeiro nome
  - Último nome
  - Salário
  - Email

```
class Funcionario:

    def __init__(self, primeiroNome, ultimoNome, salario):
        self.primeiroNome = primeiroNome
        self.ultimoNome = ultimoNome
        self.salario = salario
        self.email = primeiroNome + "." + ultimoNome + "@empresa.pt"
```

`_ _init_ _`

Construtor

método que permite

- Definir a estrutura da classe
- Construir uma instância da classe

`self`

Referência para o próprio (instância)

Instância

Instância: concretização de um objeto de uma classe

- Criação de uma instância da classe funcionário

```
f1 = Funcionario("António","Alves",1000)
```

- Mostrar o conteúdo da instância

```
print(f1) # <__main__.Funcionario object at 0x000001889A097D60>
```

- Mostrar o conteúdo de uma das variáveis

```
print(f1.primeiroNome) # António
```

- Mostrar o conteúdo da instância como um dicionário

```
print(f1.__dict__)
# {'primeiroNome': 'António', 'ultimoNome': 'Alves', 'salario': 1000, 'email': 'António.Alves@empresa.pt'}
```

_ _str_ _

- Definir a descrição textual das instâncias da classe (adicionar dentro da definição da classe o método **_ _str_ _**)

```
def __str__(self):
    return f"{self.primeiroNome} {self.ultimoNome} [{self.email}]: {self.salario}€"
```

- Mostrar o conteúdo da instância

```
print(f1) # António Alves [António.Alves@empresa.pt]: 1000€
```

Variáveis privadas

- Até agora, vimos **variáveis públicas** (acessíveis a partir de fora da classe)

```
def __init__(self, primeiroNome, ultimoNome, salario):
    self.primeiroNome = primeiroNome
    self.ultimoNome = ultimoNome
    self.salario = salario
    self.email = primeiroNome + "." + ultimoNome + "@empresa.pt"
```

- Para utilizar **variáveis privadas** (não acessíveis a partir de fora da classe, para segurança), acrescenta-se _ _ antes do nome

```
def __init__(self, primeiroNome, ultimoNome, salario):
    self.__primeiroNome = primeiroNome
    self.__ultimoNome = ultimoNome
    self.__salario = salario
    self.__email = primeiroNome + "." + ultimoNome + "@empresa.pt"
```

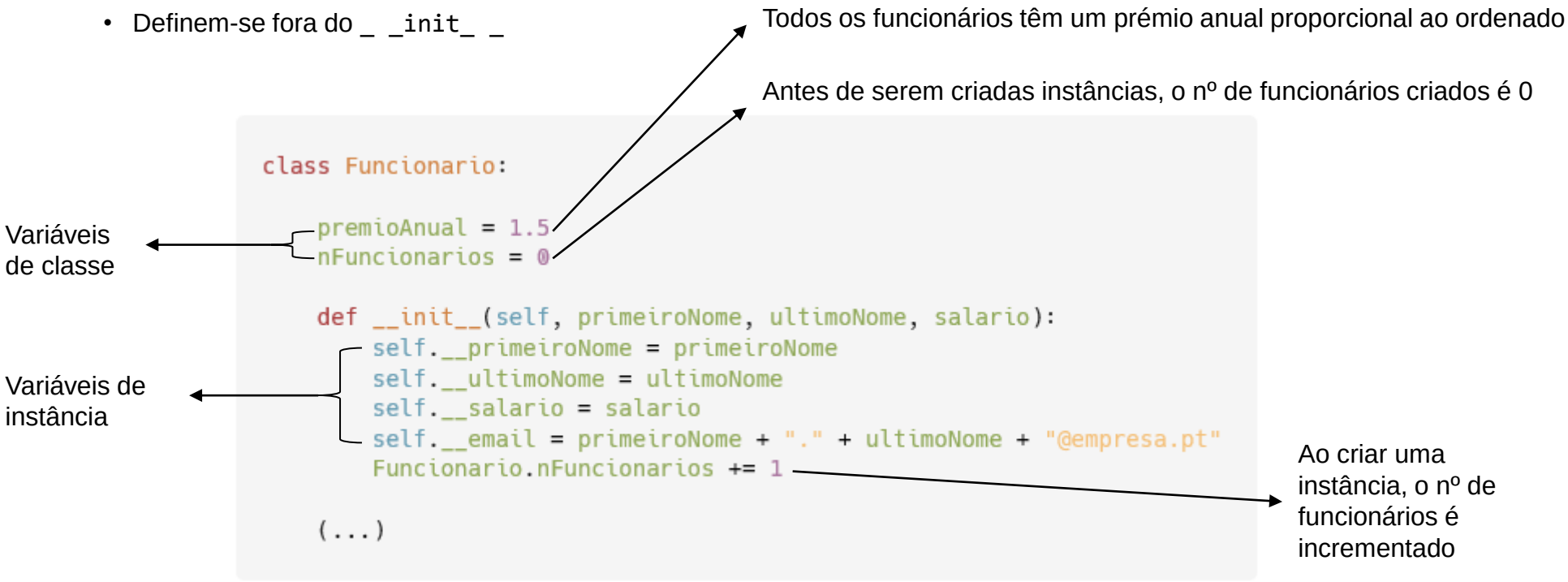
- Assim, já não estão acessíveis a partir de fora da classe

```
print(f1.__primeiroNome) # AttributeError: 'Funcionario' object has no attribute '__primeiroNome'
```

Variáveis de instância e de classe

- As variáveis que vimos até agora são **variáveis de instância**:
  - Podem ter valores distintos para cada instância da classe

- As **variáveis de classe**:
  - Têm o mesmo valor para todas as instâncias da classe
  - Definem-se fora do `_ _init_ _`




```
class Funcionario:

    premioAnual = 1.5
    nFuncionarios = 0

    def __init__(self, primeiroNome, ultimoNome, salario):
        self.__primeiroNome = primeiroNome
        self.__ultimoNome = ultimoNome
        self.__salario = salario
        self.__email = primeiroNome + "." + ultimoNome + "@empresa.pt"
        Funcionario.nFuncionarios += 1

    (...)
```

Métodos de instância

- **Método de instância**:
  - Pode ter outputs diferentes em cada instância
  - Usa variáveis de instância

```
    def nomeCompleto(self):
        return f"{self.__primeiroNome} {self.__ultimoNome}"

    print(f1.nomeCompleto())  # António Alves
```

  - Pode usar variáveis de classe

```
def mostrarProporcaoPremio(self):
    print(Funcionario.premioAnual)

f1.mostrarProporcaoPremio()  # 1.5
```

  - Pode ser usado noutros métodos

```
def __str__(self):
    return f"{self.nomeCompleto()} [{self.__email}]: {self.__salario}€"
```

Métodos de classe

- **Método de classe**:
  - Tem o mesmo output para todas as instâncias
  - Não usa variáveis de instância
  - Pode não usar nenhuma variável

```
def boasVindas(self):
    return "Bem vindo à empresa"
```

  - Pode usar variáveis de classe

```
def mostrarProporcaoPremio(self):
    print(Funcionario.premioAnual)
```

Métodos estáticos e Property Decorator @staticmethod

- Os métodos de classe não utilizam as variáveis de instância. São estáticos.

- Podemos defini-los como tal, deixando de necessitar do parâmetro **`self`**
    - Usa-se a Property Decorator **`@staticmethod`**

```
@staticmethod
def boasVindas():
    return "Bem vindo à empresa"
```

```
@staticmethod
def mostrarProporcaoPremio():
    print(Funcionario.premioAnual)
```

Getters e Property Decorator @property

- Permite a definição de *getters* (usados para obter valores de variáveis)

```
@property
def nomeCompleto(self):
    return f"{self.__primeiroNome} {self.__ultimoNome}"

@property
def email(self):
    return self.__primeiroNome + "." + self.__ultimoNome + "@empresa.pt"
```

- `nomeCompleto` e `email` passam a ser usados como se fossem variáveis (sem parêntesis)

```
print(f1.nomeCompleto) # António Alves
```

- Mesmo quando chamados por outros métodos

```
def __str__(self):
    return f"{self.nomeCompleto} [{self.email}]: {self.__salario}€"
```

Setters e Property Decorator @****.setter

- Podemos definir *setters* (usados para definir valores de variáveis)

```
@nomeCompleto.setter
def nomeCompleto(self, nome):
    primeiro, ultimo = nome.split(" ")
    self.__primeiroNome = primeiro
    self.__ultimoNome = ultimo
```

- Este método permite definir o `primeiroNome` e `ultimoNome` simultaneamente, a partir de um nome completo

```
f1.nomeCompleto = "Bernardo Bento"
print(f1) # Bernardo Bento [Bernardo.Bento@empresa.pt]: 1000€
```

Deleters e Property Decorator @****.deleter

- Podemos definir *deleters* (usados para “esvaziar” / fazer *reset* a valores de variáveis)

```
@nomeCompleto.deleter
def nomeCompleto(self):
    self.__primeiroNome = None
    self.__ultimoNome = None
```

- Este método permite “esvaziar” o `primeiroNome` e `ultimoNome` simultaneamente.
    - Como o `email` é construído a partir desses, fica também a vazio

```
del f1.nomeCompleto

print(f1.__dict__)
# {'_Funcionario__primeiroNome': None, '_Funcionario__ultimoNome': None, '_Funcionario__salario': 1000}
```

Herança e super()

- Em programação, é possível definir que uma classe (**subclasse**) herda a estrutura e métodos de outra classe (**superclasse**)

- Exemplo: há um tipo especial de funcionário na empresa (Programador) que, para além da informação e métodos para o funcionário, guarda também a linguagem de programação com que trabalha

```
class Programador(Funcionario):

    def __init__(self, primeiroNome, ultimoNome, salario, linguagem):
        super().__init__(primeiroNome, ultimoNome, salario)
        self.linguagem = linguagem

    def __str__(self):
        return f"{super().__str__()} => {self.linguagem}"

p1 = Programador("Bernardo","Bento",1000,"Python")
print(p1) # Bernardo Bento [Bernardo.Bento@empresa.pt]: 1000€ => Python
```

`Programador(Funcionario)`

Programador (subclasse) herda de Funcionário (superclasse)

`super().XXX`

Chamar o método XXX da superclasse

Depois só temos de lidar com as variáveis específicas à subclasse

`print(p1)`

Ao mostrar, o Python sabe que tem de executar o método `_ _str_ _` definido para a subclasse

UNIVERSIDADE
PORTUCALENSE

Do conhecimento à prática.