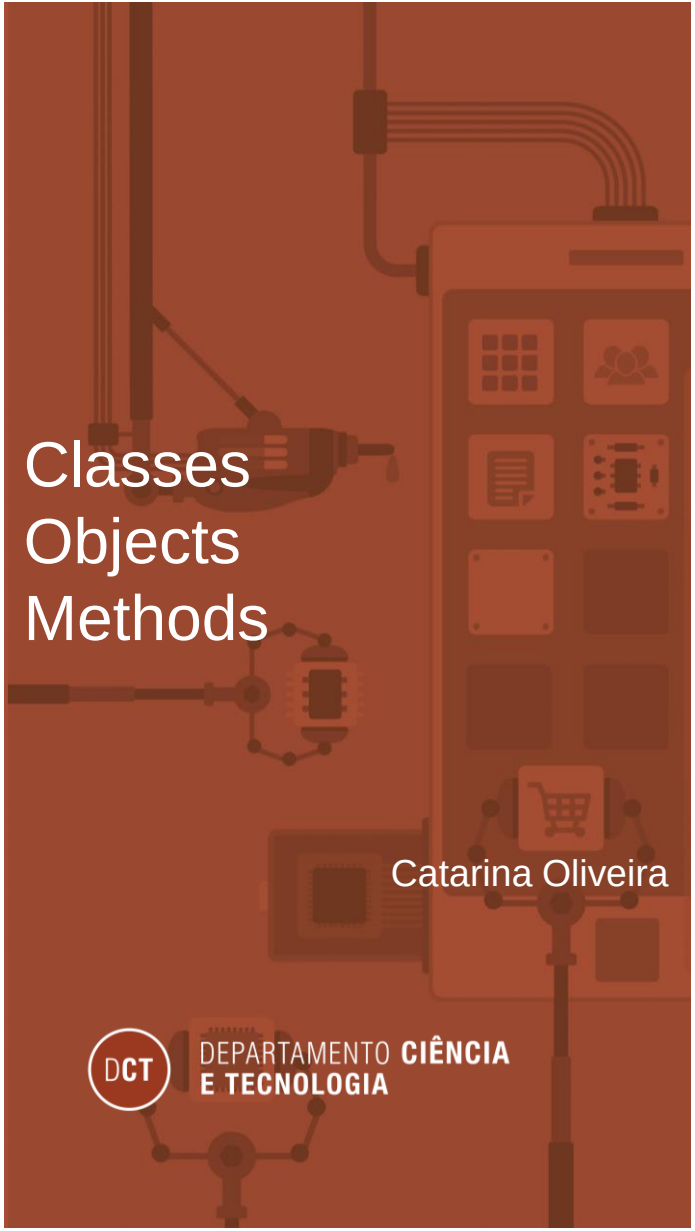


UNIVERSIDADE PORTUCALENSE

CONTENT

1. Class, _ _ init _ _ and self
2. Instance
3. _ _ str _ _
4. private variables
5. Instance and class variables
6. Instance Methods
7. class methods
8. Static Methods and Property Decorator @staticmethod
9. Getters and Property Decorator @property
10. Setters and Property Decorator @****.setter
11. Deleters and Property Decorator @****.deleter
12. Inheritance and super ()

Class, _ _ init _ _ and self

Class

- Structure that allows you to group data and methods
- “Mold” of instances

- Example: a company wants to represent its employees using a Python class. The data saved for each employee is:
  - First name
  - last name
  - Wage
  - 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 _ _`

Constructor

method that allows

- Define the structure of the class
- Construct an instance of the class

`self`

Reference to itself (instance)

Instance

Instance : embodiment of an object of a class

- Creating an instance of the employee class

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

- Show instance content

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

- Show the contents of one of the variables

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

- Show instance contents as a dictionary

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

_ _ str _ _

- **_ _ str _ _** method inside the class definition)

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

- Show instance content

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

private variables

- So far, we've looked at **public variables** (accessible from outside the class)

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

- To use **private variables** (not accessible from outside the class, for security), add _ _ before the name

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

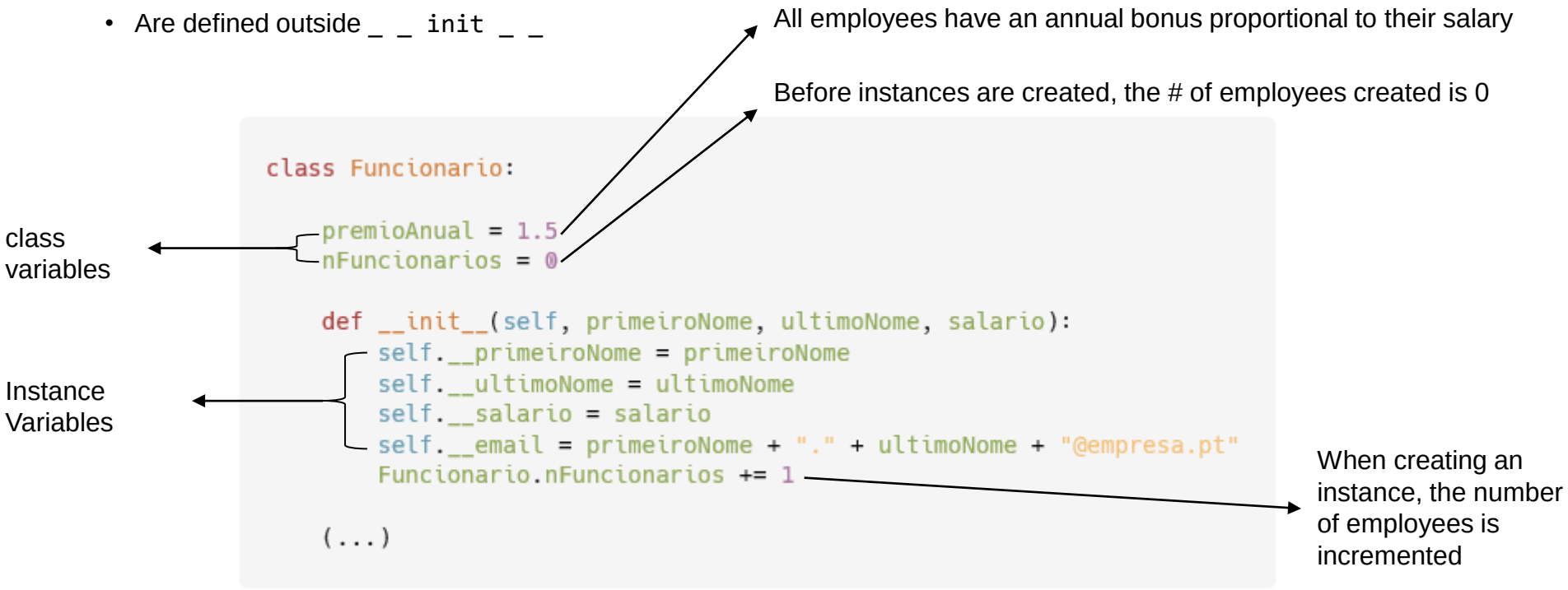
- Thus, they are no longer accessible from outside the class.

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

Instance and class variables

- The variables we've seen so far are **instance variables** :
  - Can have different values for each instance of the class

- **Class** variables :
  - Have the same value for all instances of the class
  - Are defined outside `_ _ 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

    (...)
```

Instance Methods

- **Instance method** :
  - Can have different outputs in each instance
  - Use instance variables

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

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

  - Can use class variables

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

f1.mostrarProporcaoPremio()  # 1.5
```

  - Can be used in other methods

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

class methods

- **Class method** :
  - Has the same output for all instances
  - Does not use instance variables
  - Can not use any variables

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

  - Can use class variables

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

Static Methods and Property Decorator @staticmethod

- Class methods do not use instance variables. They are static.

- We can define them as such, no longer needing the **`self parameter.`**
    - Property is used decorator **`@staticmethod`**

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

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

Getters and Property Decorator @property

- Allows the definition of *getters* (used to obtain variable values)

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

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

- `fullname` and `email` are now used as if they were variables (without parentheses)

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

- Even when called by other methods

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

Setters and Property Decorator @****.setter

- We can define *setters* (used to set variable values)

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

- This method allows defining the `firstName` and `lastName` simultaneously, starting from a complete name

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

Deleters and Property Decorator @****.deleter

- We can define *deleters* (used to “empty” / reset *variable* values)

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

- This method allows you to “empty” `firstName` and `lastName` simultaneously.
  - As the `email` is constructed from these, it is also empty.

```
del f1.nomeCompleto

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

Inheritance and super ()

- In programming, it is possible to define that a class (**subclass**) inherits the structure and methods of another class (**superclass**)

- Example: there is a special type of employee in the company (Programmer) who, in addition to information and methods for the

```
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
```

`Programmer ( Employee )`

Programmer (subclass) inherits from Employee (superclass)

`super ().xxx`

Call the superclass's XXX method

Then we just have to deal with the variables specific to the subclass

`print(p1)`

By showing, Python knows it has to execute the `_ _ str _ _ method` defined for the subclass

UNIVERSIDADE
PORTUCALENSE

Do conhecimento à prática.