CIÊNCIAS EMPRESARIAIS

ESCOLA SUPERIOR POLITÉCNICO SETÚBAL

Fundamentos de Programação

Aula 7

Mestrado em Ciência de Dados para Empresas

David Simões

Tuplos

O QUE É UM TUPLO?

- Um **tuplo** é uma forma de gerir uma *coleção ordenada* de itens
  - Semelhante a uma lista, mas imutável (não pode ser alterada no lugar)
  - Coleção: um tuplo pode conter múltiplos elementos
  - Ordenados: podemos referir os elementos pela sua posição

- Normalmente usados para gerir dados conceptualmente relacionados, como
  - Coordenadas de um *ponto*: (x, y)
  - Valores RGB para uma *cor*: (red, green, blue)
  - Elementos de uma *morada*: (rua, cidade, cp)

- Podem ser usados para retornar múltiplos valores de uma função

MOSTREM-ME OS TUPLOS!

- Criar tuplos
  - Os tuplos começam/acabam com parêntesis. Os elementos separam-se com vírgulas.

```
meu_tuplo = (1, 2, 3)
ponto = (4.7, -6.0)
strs = ('strings', 'no', 'tuplo')
morada = ('Campus do IPS', 'Estefanilha', 2914)
tuplo_vazio = ()
```

- Tuplo com um elemento tem uma vírgula (para denotar tuplo)
  - Podemos experimentar na consola

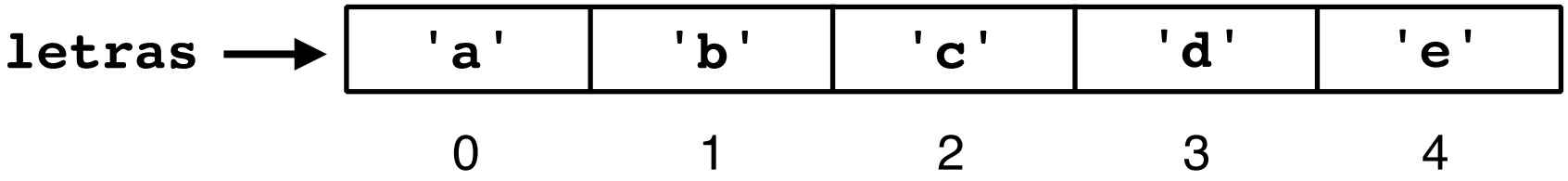
```
>>> tuplo_um = (1,)
>>> um = 1
>>> tuplo_um == um
False
```

ACEDER AOS ELEMENTOS DE UM TUPLO

- Consideremos o seguinte tuplo:

```
letras = ('a', 'b', 'c', 'd', 'e')
```

- Acedemos aos elementos de um tuplo tal como uma lista:
  - Os índices começam de 0



- Aceder a elementos individuais:

`letras[0]` é `'a'`

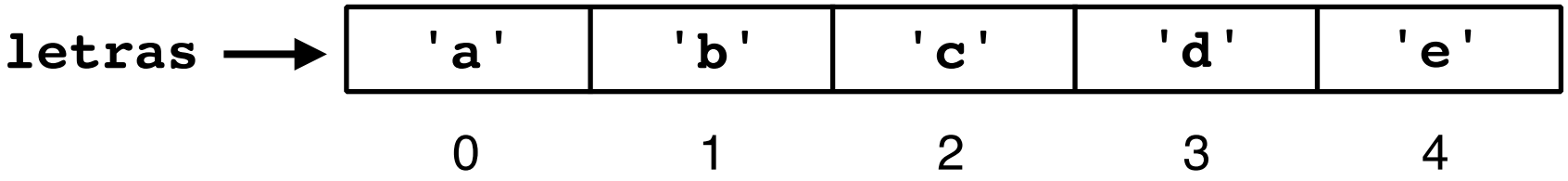
`letras[4]` é `'e'`

ACEDER AOS ELEMENTOS DE UM TUPLO

- Consideremos o seguinte tuplo:

```
letras = ('a', 'b', 'c', 'd', 'e')
```

- Acedemos aos elementos de um tuplo tal como uma lista:
  - Os índices começam de 0



- **Não podemos** atribuir aos elementos individuais:
  - Os tuplos são **imutáveis**

```
letras[0] = 'x'
```

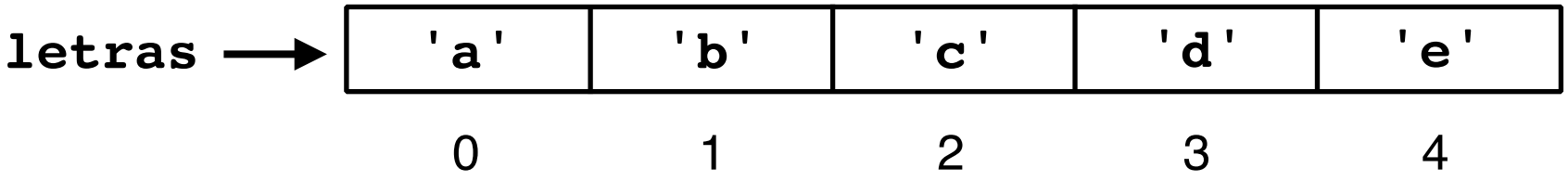
```
TypeError: 'tuple' object does not support
item assignment
```

ACEDER AOS ELEMENTOS DE UM TUPLO

- Consideremos o seguinte tuplo:

```
letras = ('a', 'b', 'c', 'd', 'e')
```

- Acedemos aos elementos de um tuplo tal como uma lista:
  - Os índices começam de 0



- **Não podemos** atribuir aos elementos individuais:
  - Os tuplos são **imutáveis**
  - Não há funções **`append`**/**`pop`** para tuplos
  - Os tuplos não podem ser alterados no lugar
  - Para alterar, necessário criar novo tuplo e escrever por cima da variável

OBTER O COMPRIMENTO DE UM TUPLO

- Consideremos o seguinte tuplo:

```
letras = ('a', 'b', 'c', 'd', 'e')
```

- Podemos obter o comprimento de um tuplo com a função **`len`**:

```
len(letras) é 5
```

  - Os elementos de um tuplo são indexados desde 0 até length - 1

- Usar o **`len`** para iterar ao longo de um tuplo:

```
for i in range(len(letras)):
    print(i, "->", letras[i])
```

```
0 -> a
1 -> b
2 -> c
3 -> d
4 -> e
```

ÍNDICES E SLICES

- Consideremos o seguinte tuplo:

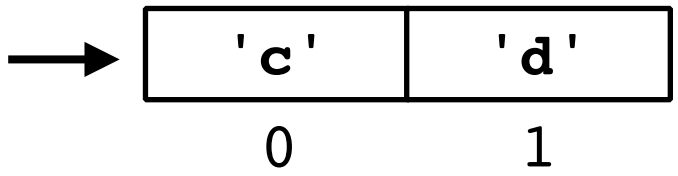
```
letras = ('a', 'b', 'c', 'd', 'e')
```

- Índices negativos num tuplo funcionam tal como nas listas
  - Contar do fim do tuplo para trás
  - Exemplo

    `letras[-1]` é **`'e'`**

- As slices funcionam nos tuplos como nas listas:

```
>>> slice = letras[2: 4]
>>> slice
('c', 'd')
```



GOOD TIMES COM TUPLOS

- Mais exemplos de tuplos:

```
turquesa_rgb = (64, 224, 208)
torre_belem = ('Av. de Brasília', 1400, 'Lisboa')
```

- Imprimir tuplos:

```
>>> print(turquesa_rgb)
(64, 224, 208)
>>> print(torre_belem)
('Av. de Brasília', 1400, 'Lisboa')
```

- Verificar se um tuplo está vazio (um tuplo vazio é como “False”)

```
if torre_belem:
    print("torre_belem não está vazio")
else:
    print("torre_belem está vazio”)
```

MAIS GOOD TIMES COM TUPLOS

- Mais exemplos de tuplos:

```
turquesa_rgb = (64, 224, 208)
torre_belem = ('Av. de Brasília', 1400, 'Lisboa')
```

- Verificar se um tuplo contém um elemento:

```
cidade = 'Lisboa'
if cidade in torre_belem:
    # fazer qualquer coisa
```

- Forma geral do teste (avalia para um Booleano)

elemento* `in` *tuplo

  - Retorna **`True`** se *elemento* é um valor no *tuplo*, **`Falso`** caso contrário
  - Também podemos testar se o elemento não está no tuplo com **`not in`**

ALGUMAS FUNÇÕES COM TUPLOS

```
turquesa_rgb = (64, 224, 208)
```

- Função: **`max(turquesa_rgb)`**
  - Retorna o valor máximo no tuplo

```
>>> max(turquesa_rgb)
224
```

- Função: **`min(turquesa_rgb)`**
  - Retorna o valor mínimo no tuplo

```
>>> min(turquesa_rgb)
64
```

- Função: **`sum(turquesa_rgb)`**
  - Retorna a soma dos valores no tuplo

```
>>> sum(turquesa_rgb)
496
```

CICLOS NOS ELEMENTOS DE UM TUPLO

```
torre_belem = ('Av. de Brasília', 1400, 'Lisboa')
```

- Ciclo **`for`** utilizando o **`range`**:

```
for i in range(len(torre_belem)):
    elem = torre_belem[i]
    print(elem)
```

- Ciclo "for-each"

```
for elem in torre_belem:
    print(elem)
```

Output:

```
Av. de Brasília
1400
Lisboa
```

- Ambos os ciclos iteram sobre todos os elementos do tuplo
  - A variável **`elem`** assume cada um dos valores do tuplo (por ordem)
  - Funciona tal como os ciclos em listas

TUPLOS COMO PARÂMETROS I

- Quando passamos um tuplo como parâmetro, é como se estivéssemos a passar um inteiro ou uma string
  - Os tuplos são imutáveis, por isso alterações numa função não persistem!

```
def remove_vermelho(tuplo_rgb)
    tuplo_rgb = (0, tuplo_rgb[1], tuplo_rgb[2])
    print("Dentro de remove_vermelho: " + str(tuplo_rgb))

def main()
    turquesa_rgb = (64, 224, 208)
    remove_vermelho(turquesa_rgb)
    print("Dentro do main: " + str(turquesa_rgb)
```

Output:

```
Dentro de remove_vermelho: (0, 224, 208)
Dentro de main: (64, 224, 208)
```

ATRIBUIÇÃO COM TUPLOS

- Podemos usar tuplos para atribuir a múltiplas variáveis de uma só vez:
  - O número de variáveis do lado esquerdo tem de ser igual ao tamanho do tuplo à direita

```
>>> (x, y) = (3, 4)
>>> x
3
>>> y
4
```

RETORNAR TUPLOS DE FUNÇÕES

- Podemos usar tuplos para devolver múltiplos valores de uma função:
  - Ponto estilístico: os valores retornados devem fazer sentido quando tomados em conjunto - ex: coordenadas (x, y)

```
def obter_data():
    dia = int(input("Dia (DD): "))
    mes = int(input("Mês (MM): "))
    ano = int(input("Ano (YYYY): "))
    return dia, mes, ano

def main():
    (dd, mm, yyyy) = obter_data()
    print(str(mm) + "/" + str(dd) + "/" + str(yyyy)
```

Terminal:

```
Dia (DD): 24
Mês (MM): 2
Ano (YYYY): 2023
24/2/2023
```

RETORNAR TUPLOS DE FUNÇÕES

- Podemos usar tuplos para devolver múltiplos valores de uma função:
  - Ponto estilístico: os valores retornados devem faze sentido quando tomados em conjunto - ex: coordenadas (x, y)

```
def obter_data():
    dia = int(input("Dia (DD): "))
    mes = int(input("Mês (MM): "))
    ano = int(input("Ano (YYYY): "))
    return dia, mes, ano

def main():
    (dd, mm, yyyy) = obter_data()
    print(str(mm) + "/" + str(dd) + "/" + str(yyyy)
```

  - Todos os caminhos da função devem retornar um tuplo com o mesmo tamanho, ou o programa pode crashar
  - Para as funções que retornam tuplos, deve existir um comentário a especificar o número de valores de retorno (e os seus tipos)

TUPLOS E LISTAS

- Podemos criar listas a partir de tuplos com a função **`list`**:

```
>>> meu_tuplo = (10, 20, 30, 40, 50)
>>> minha_lista = list(meu_tuplo)
>>> minha_lista
[10, 20, 30, 40, 50]
```

- Podemos criar tuplos a partir de listas com a função **`tuple`**:

```
>>> uma_lista = ['Adoro', 'o', 5]
>>> um_tuplo = tuple(uma_lista)
>>> um_tuplo
('Adoro', 'o', 5)
```

TUPLOS E DICIONÁRIOS

- Podemos obter pares chave/valor dos dicionários em forma de tuplos usando a função **`items`**:

```
>>> dic = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>>  list(dic.items())
[('a', 1), ('b', 2), ('c', 3), ('d', 4)]
```

- Podemos iterar pelos pares chave/valor como tuplos:

```
for chave, valor in dic.items():
    print(chave + '->' + str(valor))
```

Output:

```
a -> 1
b -> 2
c -> 3
d -> 4
```

TUPLOS EM DICIONÁRIOS

- Podemos usar tuplos como chaves em dicionários:

```
>>> dic = {('a', 1): 10, ('b', 1): 20, ('a', 2): 30}
>>> list(dic.keys())
[('a', 1), ('b', 1), ('a', 2)]
>>> list(dic.values())
[10, 20, 30]
```

- Podemos usar tuplos como valores em dicionários:

```
>>> cores = {'laranja': (255, 165, 0),
             'amarelo': (255, 255, 0),
             'aqua':    (0, 128, 128)  }
>>> list(cores.values())
[(255, 165, 0), (255, 255, 0), (0, 128, 128)]
>>> list(cores.keys())
['laranja', 'amarelo', 'aqua']
```

Object-Oriented Programming (OOP)

OBJECT-ORIENTED PROGRAMMING

- Existem diferentes *paradigmas* em programação
- Até agora, aprendemos programação *imperativa*
  - Fornecer um conjunto de comandos diretos para a execução do programa
  - Os comandos alteram o *estado* do programa
- Programação orientada a objetos
  - Definir *objetos* que contêm dados e comportamento (funções)
  - O programa é (na sua maior parte) uma interação entre objetos
  - Chamamos funções dos objetos (chamados “métodos”)
- O Python suporta a programação nos dois paradigmas
  - Existem mais paradigmas, mas não vamos falar deles em FP

O QUE SÃO CLASSES E OBJETOS?

- Classes são como um molde, ou um modelo pormenorizado
  - Fornecem a especificação para um tipo de objeto
  - Definem um novo **tipo**
  - Ex: “Pessoa” seria uma classe
    - Em geral têm dois braços, duas pernas, respiram ar, etc.

- Objetos são *instâncias* de classes
  - Podemos ter múltiplos objetos da mesma classe
  - Ex: qualquer um de nós é uma instância da classe Pessoa
    - Portanto, temos as propriedades da nossa classe (Pessoa)

EXEMPLO DE UMA CLASSE EM PYTHON

- Vamos criar uma classe Contador
  - Podemos pedir o “próximo” número de senha
  - Temos de controlar qual é o próximo número de senha
  - Sem função main() (uma classe **não** é um programa)

dois underscores - “dunder”

```
class Contador:

    # Construtor
    def __init__(self):
        self.num_senha = 0 # variável “de instância”

    # Método (função) que retorna o próximo valor da senha
    def proximo_valor(self):
        self.num_senha += 1
        return self.num_senha
```

counter.py

OS OBJETOS SÃO MUTÁVEIS

- Quando passamos um objeto como parâmetro, as alterações ao objeto dentro da função persistem depois da função terminar

```
from contador import Contador  # importar a Classe

def contar_duas_vezes(contador):
    for i in range(2):
        print(contador.proximo_valor())

def main():
    contador1 = Contador()
    contador2 = Contador()

    print('Contador1: ')
    contar_duas_vezes(contador1)

    print('Contador2: ')
    contar_duas_vezes(contador2)

    print('Contador1: ')
    contar_duas_vezes(contador1)
```

Output:

```
Contador 1:
1
2
Contador 2:
1
2
Contador 1:
3
4
```

FORMA GERAL PARA ESCREVER UMA CLASSE

- O nome de ficheiro para uma classe é normalmente ***nomeclasse*.py**

```
class NomeClasse:

    # Construtor
    def __init__(self, parâmetros adicionais):
        corpo
        self.nome_variavel = valor # variável de instância exemplo

    # Método
    def nome_metodo(self, parâmetros adicionais):
        corpo
```

CONSTRUTOR DE UMA CLASSE

- Sintaxe:

```
def __init__(self, parâmetros adicionais):
    corpo
```

- Chamado quando um novo objeto é criado
  - Não define explicitamente um valor de retorno
  - Um novo objeto é criado e retornado
    - Podemos pensar num construtor como uma “fábrica” que cria novos objetos
  - Responsável pela inicialização do objeto (definir valores iniciais)
  - Normalmente o sítio onde são criadas as variáveis de instância (com **self**)

```
self.nome_variavel = valor  # criar variável de instância
```

VARIÁVEIS DE INSTÂNCIA

- As variáveis de instância são variáveis associadas aos objetos
  - Cada objeto tem **o seu próprio conjunto** de variáveis de instância
  - As variáveis de instância são inicializadas no construtor da classe
  - Acedidas usando o **`self`**

    `self.`*`nome_variavel`*` = `*`valor`
  - O self refere-se ao próprio objeto de onde o método é chamado

```
def main():
    contador1 = Contador()
    contador2 = Contador()
    x = contador1.proximo_valor()
    y = contador2.proximo_valor()
```

VARIÁVEIS DE INSTÂNCIA

- As variáveis de instância são variáveis associadas aos objetos
  - Cada objeto tem **o seu próprio conjunto** de variáveis de instância
  - As variáveis de instância são inicializadas no construtor da classe
  - Acedidas usando o **`self`**

    `self.`*`nome_variavel`*` = `*`valor`
  - O self refere-se ao próprio objeto de onde o método é chamado

```
def main():
    contador1 = Contador()
    contador2 = Contador()
    x = contador1.proximo_valor()
    y = contador2.proximo_valor()
```

```
def __init__(self):
    self.num_senha = 0
```

VARIÁVEIS DE INSTÂNCIA

- As variáveis de instância são variáveis associadas aos objetos
  - Cada objeto tem **o seu próprio conjunto** de variáveis de instância
  - As variáveis de instância são inicializadas no construtor da classe
  - Acedidas usando o **`self`**

    `self.`*`nome_variavel`*` = `*`valor`
  - O self refere-se ao próprio objeto de onde o método é chamado

```
def main():
    contador1 = Contador()
    contador2 = Contador()
    x = contador1.proximo_valor()
    y = contador2.proximo_valor()
```

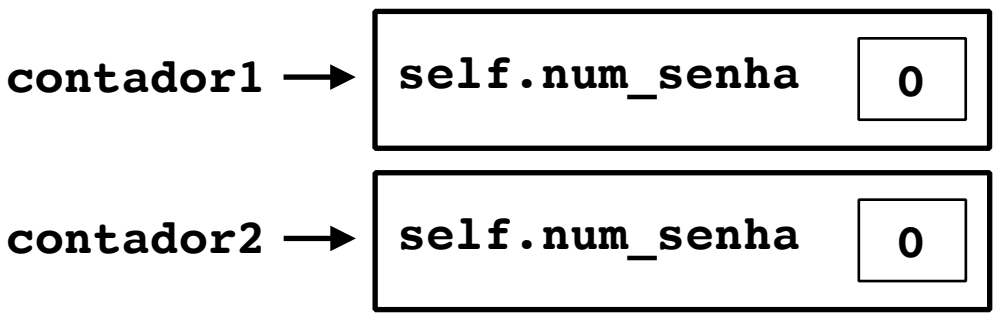
VARIÁVEIS DE INSTÂNCIA

- As variáveis de instância são variáveis associadas aos objetos
  - Cada objeto tem **o seu próprio conjunto** de variáveis de instância
  - As variáveis de instância são inicializadas no construtor da classe
  - Acedidas usando o **`self`**

    `self.`*`nome_variavel`*` = `*`valor`
  - O self refere-se ao próprio objeto de onde o método é chamado

```
def main():
    contador1 = Contador()
    contador2 = Contador()
    x = contador1.proximo_valor()
    y = contador2.proximo_valor()
```

```
def __init__(self):
    self.num_senha = 0
```

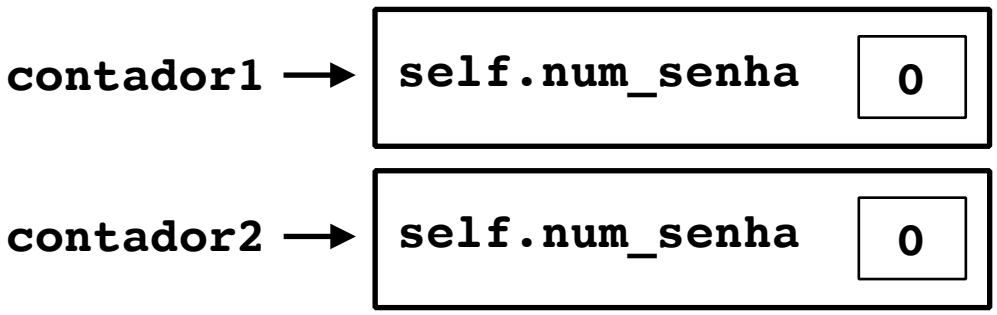


VARIÁVEIS DE INSTÂNCIA

- As variáveis de instância são variáveis associadas aos objetos
  - Cada objeto tem **o seu próprio conjunto** de variáveis de instância
  - As variáveis de instância são inicializadas no construtor da classe
  - Acedidas usando o **`self`**

    `self.`*`nome_variavel`*` = `*`valor`
  - O self refere-se ao próprio objeto de onde o método é chamado

```
def main():
    contador1 = Contador()
    contador2 = Contador()
    x = contador1.proximo_valor()
    y = contador2.proximo_valor()
```



VARIÁVEIS DE INSTÂNCIA

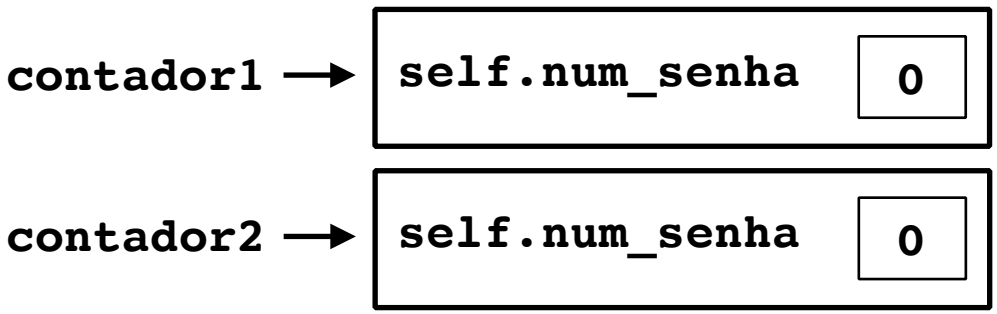
- As variáveis de instância são variáveis associadas aos objetos
  - Cada objeto tem **o seu próprio conjunto** de variáveis de instância
  - As variáveis de instância são inicializadas no construtor da classe
  - Acedidas usando o **`self`**

    `self.`*`nome_variavel`*` = `*`valor`
  - O self refere-se ao próprio objeto de onde o método é chamado

```
def main():
    contador1 = Contador()
    contador2 = Contador()
    x = contador1.proximo_valor()
    y = contador2.proximo_valor()
```

```
def proximo_valor(self):
    self.num_senha += 1
    return self.num_senha
```

contador1



VARIÁVEIS DE INSTÂNCIA

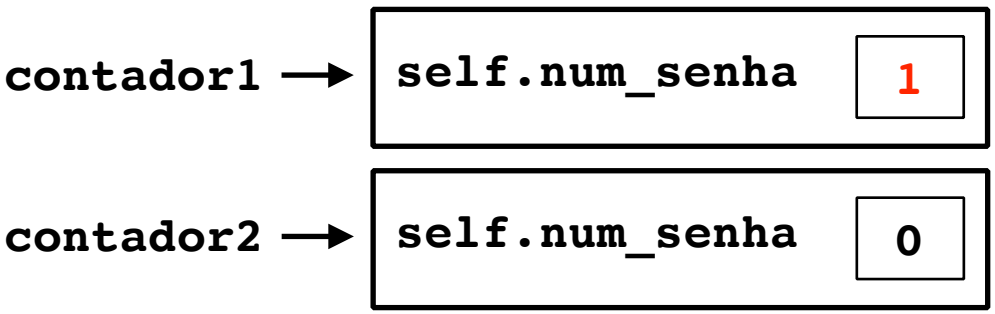
- As variáveis de instância são variáveis associadas aos objetos
  - Cada objeto tem **o seu próprio conjunto** de variáveis de instância
  - As variáveis de instância são inicializadas no construtor da classe
  - Acedidas usando o **`self`**

    `self.`*`nome_variavel`*` = `*`valor`
  - O self refere-se ao próprio objeto de onde o método é chamado

```
def main():
    contador1 = Contador()
    contador2 = Contador()
    x = contador1.proximo_valor()
    y = contador2.proximo_valor()
```

```
def proximo_valor(self):
    self.num_senha += 1
    return self.num_senha
```

contador1

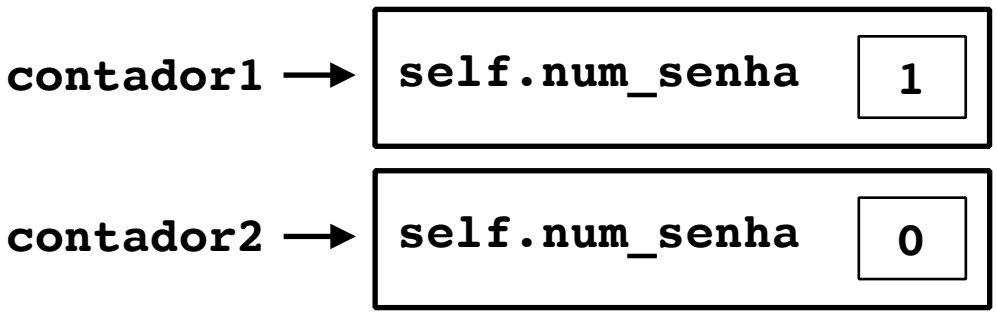


VARIÁVEIS DE INSTÂNCIA

- As variáveis de instância são variáveis associadas aos objetos
  - Cada objeto tem **o seu próprio conjunto** de variáveis de instância
  - As variáveis de instância são inicializadas no construtor da classe
  - Acedidas usando o **`self`**

    `self.`*`nome_variavel`*` = `*`valor`
  - O self refere-se ao próprio objeto de onde o método é chamado

```
def main():
    contador1 = Contador()
    contador2 = Contador()
    x = contador1.proximo_valor()
    y = contador2.proximo_valor()
```



VARIÁVEIS DE INSTÂNCIA

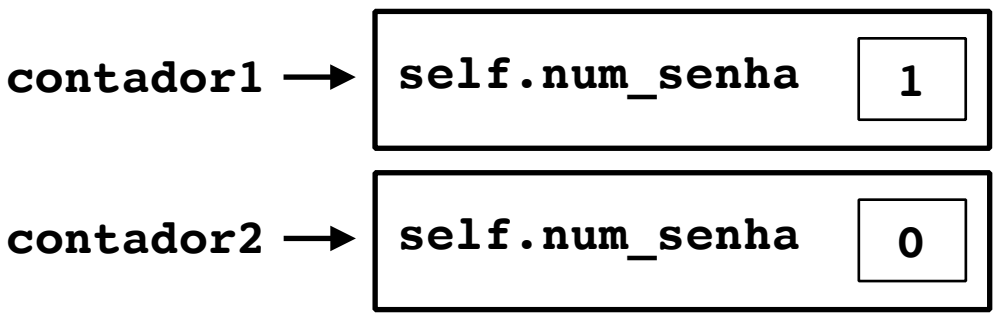
- As variáveis de instância são variáveis associadas aos objetos
  - Cada objeto tem **o seu próprio conjunto** de variáveis de instância
  - As variáveis de instância são inicializadas no construtor da classe
  - Acedidas usando o **`self`**

    `self.`*`nome_variavel`*` = `*`valor`
  - O self refere-se ao próprio objeto de onde o método é chamado

`contador2`

```
def main():
    contador1 = Contador()
    contador2 = Contador()
    x = contador1.proximo_valor()
    y = contador2.proximo_valor()
```

```
def proximo_valor(self):
    self.num_senha += 1
    return self.num_senha
```



VARIÁVEIS DE INSTÂNCIA

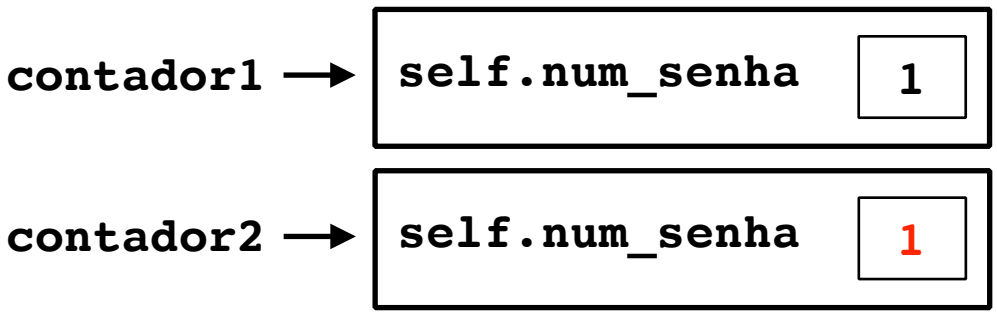
- As variáveis de instância são variáveis associadas aos objetos
  - Cada objeto tem **o seu próprio conjunto** de variáveis de instância
  - As variáveis de instância são inicializadas no construtor da classe
  - Acedidas usando o **`self`**

    `self.`*`nome_variavel`*` = `*`valor`
  - O self refere-se ao próprio objeto de onde o método é chamado

`contador2`

```
def main():
    contador1 = Contador()
    contador2 = Contador()
    x = contador1.proximo_valor()
    y = contador2.proximo_valor()
```

```
def proximo_valor(self):
    self.num_senha += 1
    return self.num_senha
```

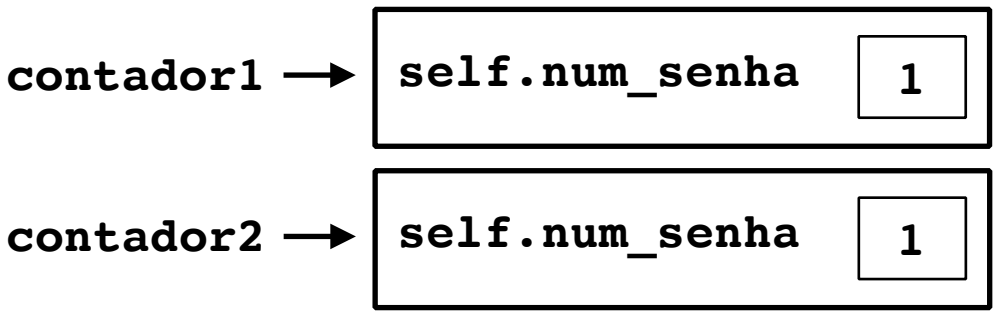


VARIÁVEIS DE INSTÂNCIA

- As variáveis de instância são variáveis associadas aos objetos
  - Cada objeto tem **o seu próprio conjunto** de variáveis de instância
  - As variáveis de instância são inicializadas no construtor da classe
  - Acedidas usando o **`self`**

    `self.`*`nome_variavel`*` = `*`valor`
  - O self refere-se ao próprio objeto de onde o método é chamado

```
def main():
    contador1 = Contador()
    contador2 = Contador()
    x = contador1.proximo_valor()
    y = contador2.proximo_valor()
```



MÉTODOS (FUNÇÕES) NUMA CLASSE

- Sintaxe:

```
def nome_metodo(self, parâmetros adicionais):
    corpo
```

- Funciona como uma função normal em Python
  - Pode retornar valores (como uma função normal)
  - Tem acesso às variáveis de instância (através do **self**)

    self.*nome_variavel* = *valor*
  - Chamados através de um objeto:

    nome_objeto.nome_metodo(parâmetros adicionais)
  - Não esquecer, o parâmetro self é automaticamente definido pelo Python como o objeto do qual este método é chamado
    - Nós escrevemos: **numero = contador1.proximo_valor()**
    - O Python trata-o como **numero = proximo_valor(contador1)**

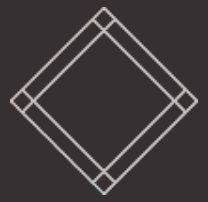
OUTRO EXEMPLO: ESTUDANTES

- Queremos uma classe para gerir informação sobre Estudantes
  - Cada estudante tem informação:
    - Nome
    - Número de estudante
    - Créditos ECTS adquiridos
  - Queremos especificar um nome e um número ao criar um objeto estudante
    - No início, créditos inicializados a 0
  - Número de ECTS do estudante pode ser atualizado ao longo do tempo
  - Também queremos poder verificar se um estudante completou o curso
    - Estudante necessita de ter um número de créditos >= ECTS_PARA_TERMINAR

Mostrem-me os estudantes!
estudante.py

OBJETIVOS DE APRENDIZAGEM

1. Aprender tuplos em Python
2. Aprender sobre programação orientada a objetos
3. Escrever código em Python com Classes e Objetos



REFERÊNCIAS

- Slides adaptados de Piech and Sahami, CS106A, Stanford University
- PyCharm - Jetbrains, https://www.jetbrains.com/pycharm/

FP@moodle

https://moodle.ips.pt/2223/course/view.php?id=1100