INSTITUTO UNIVERSITÁRIO DE LISBOA

2019/2020

Licenciatura em Ciência de Dados – PL

UC Estruturas de Dados e Algoritmos

1º Ano 2º Semestre

TRABALHO I - A

CONSEQUÊNCIAS DE SE FAZER REFERÊNCIAS MÚLTIPLAS A OBJETOS MUTÁVEIS EM PYTHON:

Exploração de um Exemplo

Trabalho realizado por:

João Diogo Mendes Martins, n.º 93259

PARTE A) "Qual a consequência de fazer várias referências a um objeto mutável ou uma referência a um objeto imutável, na linguagem Python?" (Ver enunciado – Anexo I)

Os nomes simbólicos são no fundo designações que damos a qualquer objeto por nós criado em Python, sendo metaforicamente um "contentor" que armazena uma referência de memória que informa onde está armazenado o valor associado ao objeto. Por exemplo, na atribuição `temperatura = 20`, o nome simbólico `temperatura` é usado para designar o inteiro `20`. Podemos usar diferentes nomes simbólicos para um mesmo objeto (fazer múltiplas referências) – por exemplo, se além da atribuição anterior, fizermos também `temp_max = 20`. Nestes exemplos, tanto `temperatura` como `temp_max` passam a referir-se ao mesmo objeto: o inteiro `20`. A cada nome simbólico referente à mesma entidade, designa-se de pseudónimo (*alias*, no Inglês) (Martins, 2015).

As estruturas de dados em Python podem ser categorizadas em objetos mutáveis ou imutáveis. Como objetos mutáveis temos os tipos de objeto lista, conjunto (*set*), e dicionário. Nos imutáveis, temos os tipos: inteiro; número de ponto flutuante (*float*); sequência de texto (*string*); tuplos; booleano (*bool*); e *frozenset* (Goodrich *et al.*, 2013) Ora, consoante esta característica de mutabilidade, o comportamento dos objetos no processo de atribuição será distinto. Vejamos o que acontece esquematicamente, usando duas referências para manipular objetos mutáveis (neste caso, listas):

```
lista1 = [1, 2, 3]    #Figura 1
lista2 = lista1       #Figura 1
lista2[1] = 0         #Figura 2
```

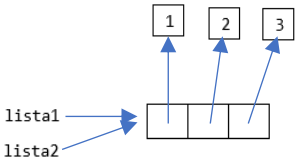


Figura 1

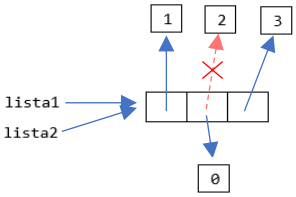


Figura 2

Fazer múltiplas referências a um mesmo objeto que é mutável, ou seja, fazendo o *aliasing*, significa que conseguimos alterar o valor original do objeto através de qualquer um dos *alias* (Martins, 2015). Isto é observável no comportamento das listas: se criar `lista1 = [1, 2, 3]` e `lista2 = lista1`, e se fizer `lista2[1] = 0`, alteraremos indiretamente o valor também em `lista1`, no índice 1.

Nos objetos imutáveis isto já não acontece, porque simplesmente não conseguimos alterar o valor original por natureza. Vejamos o exemplo:

```
temperatura = 20              #Figura 3
temp_max = temperatura        #Figura 3
temperatura = 24              #Figura 4
```

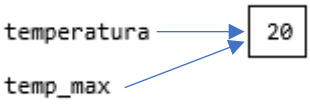


Figura 3

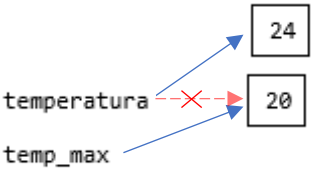


Figura 4

Neste caso atribuem-se os pseudónimos `temperatura` e `temp_max` ao objeto inteiro `20` (Figura 3). Contudo, ao se fazer de seguida a atribuição `temperatura = 24`, o objeto inteiro não é alterado (pois é imutável); em vez disto, `temperatura` passa a apontar para o sítio de memória onde existe o objeto inteiro 24. Com isto, as duas variáveis passam a ser entidades diferentes, uma vez que deixam de estar associadas ao mesmo objeto original (Figura 4).

PARTE B) "Faça a depuração (*debug*) do código (…) e explique o que se passou. (…) Tente descobrir e explicar o que não está a funcionar e porquê, e indique, num relatório, um código correto." (Ver enunciado – Anexo I)

Problema: *programa não gera o output esperado.*

Ao fazer correr o programa, tal como está disponível no enunciado, verificamos que nenhum *output* é gerado. Ora, com base nas instruções presentes da linha 51 à linha 57, seria expectável que o programa retornasse os objetos da classe `Empregado`, assim como o respetivo departamento (objeto da classe `Dept`) a que cada empregado pertence ou já pertenceu. "Traduzindo", temos no último bloco de código relativo aos testes:

```
51  se o módulo correr como programa principal:
52      # testes
53      cria a variável dept como sendo objeto da classe Dept, com os argumentos ('dept1', 'div1')
54      cria 20 objetos da classe Empregado, com a nomenclatura emp+i
55      cria um segundo departamento (dept2), invocando o método divide_dept()
56      print dos objetos da classe Empregado pertencentes ao dept1, assim como os respetivos departamentos
57      print dos objetos da classe Empregado pertencentes ao dept2, assim como os respetivos departamentos
```

E sabe-se já que nada é retornado. Existe então um problema de funcionalidade.

O método/programa começa por definir uma classe `Dept`, com os atributos `self.emps`, `self.nome` e `self.divisao`, ou seja, variáveis que armazenarão o conjunto de objetos da classe Empregado, o nome do departamento, e a divisão a que pertence, respetivamente. A estrutura de dados escolhida para armazenar o conjunto dos empregados é um *set*. Em linhas posteriores, encontramos um método dentro da classe que permite adicionar novos empregados ao departamento, adicionando o objeto da classe `Empregado` ao *set* `self.emps`, e adicionando o objeto da classe Dept à lista `emp.dept` (atributo da classe `Empregado`).

O enunciado mostra tentativas de correção na linha 19, em que temos `novo_dept = self.` Nas linhas anteriores a esta, não se evidencia nenhuma causa possível para o problema, apesar de na linha 17 a função `divide_dept` ter um parâmetro que não é utilizado no seu corpo (o parâmetro `novo`). Este ponto, que não está relacionado com o problema, será solucionado na correção do código presente no fim do exercício.

O problema é encontrado na função `divide_dept`. Esta tem como propósitos: criar uma variável `novo_dept` que é teoricamente uma cópia de `self` (o departamento original); eliminar metade dos empregados que compõem o *set* `self.emps` e criar uma lista `restantes` com os elementos que ficam no *set*; por cada empregado que se encontra em `restantes`, eliminar a sua respetiva cópia do *set* `novo_dept.emps`; em cada empregado que fica em `novo_dept`, invoca-se um método da classe `Empregado` (muda_dept) para acrescentar o novo departamento à lista `emp.dept`. Teoricamente, com o `return` deverá ser devolvido um objeto `novo_dept` que contém um *set* `novo_dept.emps` com metade dos empregados que compunham o *set* `self.emps`. Contudo, o que acontece realmente é que este método retorna o objeto `novo_dept` cujo set `novo_dept.emps` se encontra vazio. Aliás, tanto `novo_dept.emps` como `self.emps` acabarão por ficar vazios com a execução do método `divide_dept`.

Observação: *com* `novo_dept = self` *assume-se que* `novo_dept` *e* `self` *serão entidades independentes, apesar de cópias uma da outra, mas a realidade é diferente.*

Ao fazermos a atribuição `novo_dept = self`, estamos no fundo a "afirmar" que `novo_dept` terá a mesma referência de memória que `self`, ou seja, ambos os objetos serão compostos pelos mesmos apontadores de memória. Para esta verificação, basta introduzir na linha 20:

```
20          print("referência de self: ", self "\nreferência de novo_dept: ", novo_dept)
```

E temos como *output*:

```
referência de self:  <__main__.Dept object at 0x000001574736B208>
referência de novo_dept:  <__main__.Dept object at 0x000001574736B208>
```

Confirma-se assim que ambos os objetos têm na sua origem o mesmo sítio na memória (`0x000001574736B208`). Poder-se-á dizer que são o mesmo objeto, mas com duas designações possíveis. Na linha 25 do *script* original (Anexo I), verificamos a aplicação do método `pop()` ao conjunto de empregados `self.emps`, com o objetivo que este fique composto somente por metade dos elementos originais. Contudo, ao fazermos isto, significa que todos aqueles elementos do *set* que são alvo do método acabam por ser perdidos, uma vez que não são atribuídos a nenhum outro objeto. Erradamente, está-se a assumir que como se fez anteriormente a atribuição `novo_dept = self`, `novo_dept.emps` contém o conjunto de empregados original, e que por isto poderemos retirar metade dos elementos a `self.emps` sem afectar `novo_dept`. Mas a realidade é que, por terem ponteiros para o mesmo sítio de memória, ao alterarmos o conteúdo de um estamos indiretamente a alterar também o conteúdo do outro.

Na tabela abaixo são apresentadas as transformações que cada variável sofre ao longo do método `divide_dept`, <u>após</u> a ação de cada linha de código da primeira coluna. São atribuídos identificadores únicos aos objetos, utilizando o método `id()`.

Tabela 1

Código	print(id(self.emps))	if len(self.emps) != 0: [print(id(emp)) for emp in self.emps] else: print("self.emps is empty")	print(id(novo_dept.emps))	if len(novo_dept.emps) != 0: [print(id(emp)) for emp in novo_dept.emps] else: print("novo_dept.emps is empty")	print(id(restantes))	if len(restantes) != 0: [print(id(emp)) for emp in restantes] else: print("restantes is empty")
19 novo_dept = self	1899810826280	1899811172424 1899811262536 1899811262664 1899811275080 1899811262792 1899811262920 1899811263048 1899811175112 1899811263176 1899811175240 1899811277704 1899811175368 1899811275784 1899811175560 1899811175944 1899811173960 1899811174088 1899811278600 1899811176264 1899811174216	1899810826280	1899811172424 1899811262536 1899811262664 1899811275080 1899811262792 1899811262920 1899811263048 1899811175112 1899811263176 1899811175240 1899811277704 1899811175368 1899811275784 1899811175560 1899811175944 1899811173960 1899811174088 1899811278600 1899811176264 1899811174216	Não Disponível	Não Disponível

21	`for i in range(int(len(self.emps)/2)): self.emps.pop()`	1899810826280	1899811277704 1899811175368 1899811275784 1899811175560 1899811175944 1899811173960 1899811174088 1899811278600 1899811176264 1899811174216	1899810826280	1899811277704 1899811175368 1899811275784 1899811175560 1899811175944 1899811173960 1899811174088 1899811278600 1899811176264 1899811174216	Não Disponível	Não Disponível
24	`restantes = [emp for emp in self.emps]`	1899810826280	1899811277704 1899811175368 1899811275784 1899811175560 1899811175944 1899811173960 1899811174088 1899811278600 1899811176264 1899811174216	1899810826280	1899811277704 1899811175368 1899811275784 1899811175560 1899811175944 1899811173960 1899811174088 1899811278600 1899811176264 1899811174216	1899811172552	1899811277704 1899811175368 1899811275784 1899811175560 1899811175944 1899811173960 1899811174088 1899811278600 1899811176264 1899811174216
27	`for emp in restantes: novo_dept.emps.remove(emp)`	1899810826280	self.emps is empty	1899810826280	novo_dept.emps is empty	1899811172552	1899811277704 1899811175368 1899811275784 1899811175560 1899811175944 1899811173960 1899811174088 1899811278600 1899811176264 1899811174216

Através da observação da tabela é possível concluir:

```
19 novo_dept = self
```

 # self.emps e novo_dept.emps são o mesmo objecto (id: 1899810826280) e cujo conteúdo, por sua vez, se refere ao mesmo conjunto de emps (Nota: como são sets, os seus elementos não estão organizados, ao contrário do representado abaixo; não dá para prever a ordem dos elementos dentro do set) [Figura 5]

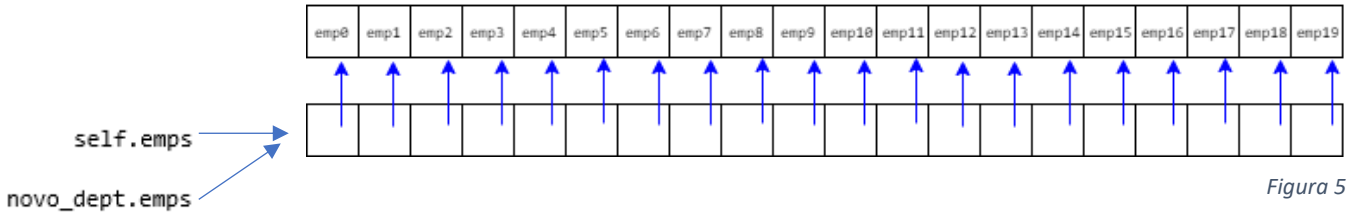


Figura 5

```
21 for i in range(int(len(self.emps)/2)):
22
23     self.emps.pop()
```

 # ao se eliminar metade dos elementos de self.emps, elimina-se indiretamente metade dos elementos de novo_dept.emps [Figura 6]

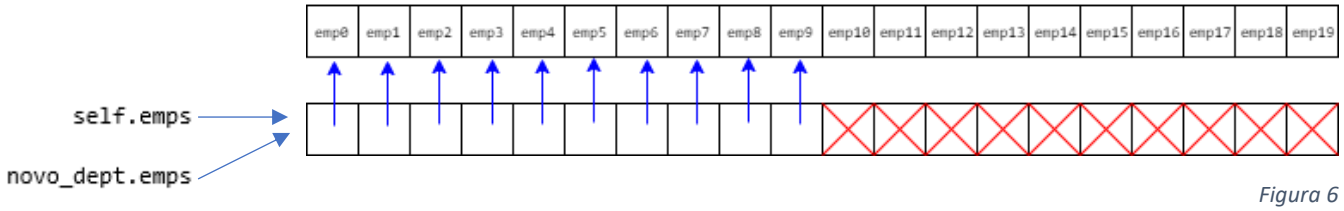


Figura 6

```
24 restantes = [emp for emp in self.emps]
25
26
```

 # apesar de ser criado um objeto distinto (restantes), com id diferente dos outros, o seu conteúdo são os mesmos objetos que compõem self.emps e novo_dept.emps, ou seja, esta variável é uma lista composta por ponteiros para os sítios de memória em que encontramos os elementos de self.emps ou novo_dept.emps [Figura 7]

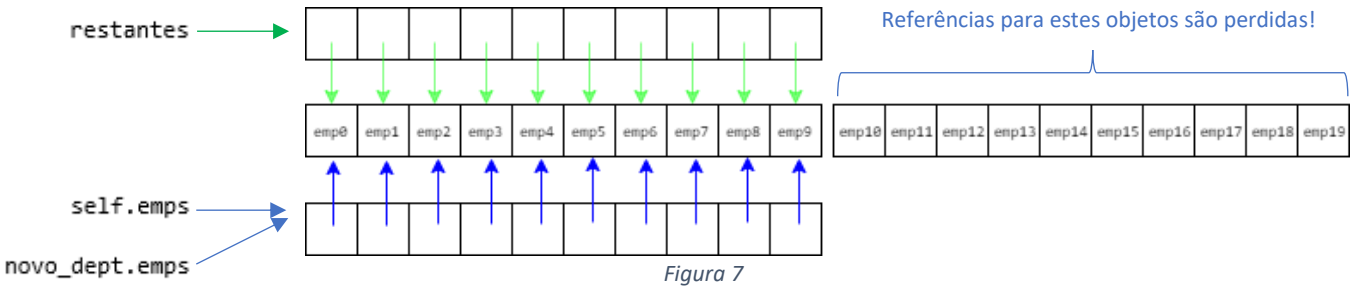


Figura 7

```
27 for emp in restantes:
28
29     novo_dept.emps.remove(emp)
```

```
# ao ser instruída a remoção dos elementos em novo_dept.emps que também compõem restantes, todos eles aqueles acabam removidos; e da mesma forma que acontece na linha 21, elimina-se também indiretamente os elementos de self.emps [Figura 8]
```

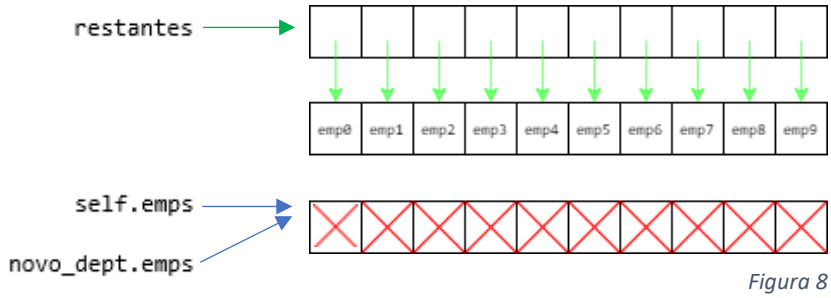


Figura 8

O problema no código é que se está a fazer uma múltipla referência a objetos mutáveis, com a atribuição `novo_dept = self`. Implica termos dois nomes simbólicos para o mesmo *set* que contém as referências para os elementos `emp`. Por isso, tal como já tinha sido observado em exemplos no exercício da parte A, neste caso estaremos inadvertidamente a manipular sempre o mesmo conjunto de objetos, não fazendo diferença se o manipulamos usando `self.emps` ou `novo_dept.emps`.

Problema: *ao se alterar a linha 19 para `novo_dept = self.copy()`, como sugerido no enunciado, é gerado um erro.*

Substituindo a linha 19 por `novo_dept = self.copy()`, continuamos sem sucesso no *output* pretendido. Aliás, substituindo de forma literal, o resultado obtido é um erro (Anexo II). Significa que tentámos invocar um método da classe `Dept` que não existe: `copy()`. Uma forma correta seria invocar o método `copy()` do módulo `copy`, ou seja: `novo_dept = copy.copy(self)`.

Problema: *com `novo_dept = copy.copy(self)` também não é gerado o output esperado.*

Mesmo apesar da correção aplicada acima continua a não surgir nenhum *output*.

A explicação para isto é em tudo semelhante à causa do problema inicial: não estamos a lidar com verdadeiras cópias dos elementos que compõem as estruturas de dados originais. De acordo com o que encontramos na documentação oficial da linguagem Python, a interface `copy.copy(x)` cria uma "cópia oca" (*shallow copy*, no original), "construindo um novo objecto composto e (dentro do possível) insere nele referências para os objectos encontrados no original" (2020). Ou seja, `self` e `novo_dept` são objetos distintos desta vez (com identificadores diferentes, se se aplicar o método `id()`). Contudo, na sua composição, os atributos `self.emps` e `novo_dept.emps` continuarão a partilhar as mesmas referências de memória e os elementos de ambos continuarão a ser o mesmo conjunto de objetos da classe Empregado. Consequentemente, ao se retirar elementos de `self.emps`, continua-se a retirar também de forma indireta elementos de `novo_dept.emps`, e vice-versa. E o "filtro" `restantes` — chamemos-lhe assim — contribuirá na mesma para que todos os elementos de `novo_dept.emps` sejam eliminados, sobrando um *set* vazio.

Problema: *com `novo_dept = copy.deepcopy(self)`, como feito no enunciado, é gerado erro.*

Por fim, a última alteração sugerida pelo enunciado implica substituir a linha 19 por `novo_dept = copy.deepcopy(self)`. Mas obtemos novo erro (Anexo III).

Aqui, é possível perceber que o ciclo `for` que inicia na linha 30 (linha 27 no *script* original) gera o erro porque os elementos `emp` de `restantes` são agora entidades distintas dos elementos `emp` de `novo_dept.emps`. Logo, quando na teoria queremos que "por cada elemento presente em `restantes` se remova esse elemento em `novo_dept.emps`", na linha 32 `novo_dept.emps.remove(emp)` não pode ser executado porque o elemento não existe no *set*.

```
30          for emp in restantes:
31          # retira os que ficaram do novo departamento
32              novo_dept.emps.remove(emp)
```

Objetos distintos!

Solução:

Para garantir que o programa funcione como intencionado, teremos de garantir que o método `divide_dept` da classe `Dept` crie de facto um novo objeto departamento independente do original (algo que foi conseguido no passo anterior) e que, de forma eficaz, mantenha metade dos empregados no departamento original e adicione a outra metade no novo departamento. A correção encontrada passa por criar uma cópia do departamento original, esvaziar o *set* `novo_dept.emps`, e então fazer uma migração de metade dos empregados no departamento original (`self.emps`) para este novo departamento:

```
1       import copy
…
20      def divide_dept(self, novo):
21          """ divide o departamento"""
22          novo_dept = copy.deepcopy(self)
23          novo_dept.nome = novo
24          novo_dept.emps = set()
25          for i in range(int(len(self.emps)/2)):
26              novo_dept.emps.add(self.emps.pop())
27          for emp in novo_dept.emps:
28              # adiciona o novo departamento na lista
29              # dos departamentos já trabalhados
30              emp.muda_dept(novo_dept)
31          return novo_dept
```

#utilizando o método **deepcopy** garantimos que é criada uma cópia de **self**: **novo_dept** – um objeto composto independente do original; desta forma assegura-se que qualquer modificação feita a este novo objeto não se repercute naquele que lhe deu origem

cada elemento retirado de **self.emps** serve de argumento para ser adicionado ao novo departamento; garantimos assim num único ciclo que metade dos elementos do departamento original sejam migrados para **novo_dept**, sem recorrer à lista **restantes**

Como output (Anexo IV), temos então o pretendido – todos os colaboradores que foram criados assim como os departamentos a que pertencem ou já pertenceram.

Na linha 19, poderíamos ter optado por não criar uma cópia e sim criar um novo departamento – `novo_dept = Dept(novo, self.divisao)` – e eliminar as linhas 23 e 24. Contudo, caso os objetos da classe `Dept` vissem mais atributos acrescentados numa futura edição do código, os respetivos valores não seriam mantidos no novo departamento criado. Assim, conclui-se que a forma mais eficaz é criar de facto uma cópia e editar apenas os atributos necessários (conjunto de empregados e nome do departamento), mantendo tudo o resto (consultar programa completo e corrigido no Anexo V).

Conclusão:

Como vimos, para que o programa funcione como intencionado, temos de garantir que o método `divide_dept` da classe Dept crie uma cópia do objeto departamento já existente, com uma referência distinta, e cujos objetos que o compõem tenham também referências diferentes daquelas do departamento original. Desta forma, previnem-se problemas de funcionalidade causados pela múltipla referenciação a um objeto mutável. Teremos assim a composição de dois departamentos que são, de facto, entidades distintas.

REFERÊNCIAS BIBLIOGRÁFICAS:

Copy - Shallow and deep copy operations. (28 de Março 2020). Retirado em Março 28, 2020, de https://docs.python.org/3/library/copy.html

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). Data structures and algorithms in Python. Hoboken: John Wiley & Sons.

Martins, J. P. (2015). Programação em Python: Introdução à Programação Utilizando Múltiplos Paradigmas. Lisboa: IST - Instituto Superior Técnico.

ANEXOS

ANEXO I:

(Enunciado fornecido)

iscte
INSTITUTO UNIVERSITÁRIO DE LISBOA

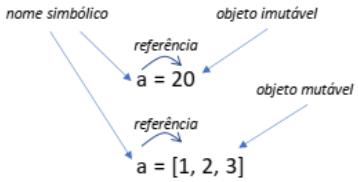
Estruturas de Dados e Algoritmos **Ano letivo 2019/2020**

Trabalho 1 - A

A) Consulte a referências bibliográficas [1] ou [2] ou pesquise em sítios **fidedignos** na Internet, e perceba o que são entre nomes simbólicos e referências (references) e a consequência de fazer várias referências a um objeto mutável ou uma referência a um objeto imutável, na linguagem Python.

[1] **Data Structures & Algorithms in Python, M. Goodrich, R. Tamassia e M. Goldwasser, Wiley, 2013.**

[2] **Problem Solving with Algorithms and Data Structures using Python by Bradley N. Miller, David L. Ranum Release 3.0, 2013.**



B) Faça a depuração (*debug*) do código (na página seguinte) e explique o que se passou, recorrendo a diagramas de apontadores. Se entender que o ajuda a diagnosticar e a explicar os erros cometidos, pode socorrer-se da função "*id(obj)*", que atribui ao objeto dado como argumento um identificador único.

O código define dois tipos de objetos, Departamentos e Trabalhadores, tais que,

- Um departamento tem informação associada (omissa no código abaixo indicado) para além de uma lista dos trabalhadores do departamento.
- Um trabalhador, tem informação própria associada (nome, etc.) e, ainda, uma lista de departamentos onde já trabalhou, sendo o fim da lista o departamento atual.

Os departamentos por vezes crescem demasiado, altura em que são divididos ao meio, criando um novo departamento, com toda a informação original, mas com novo nome e metade dos empregados originais.

Quem tentou codificar as respetivas classes, para efetuar esta divisão, criou um método *divide_dept*, para a divisão dos departamentos. No entanto, este método não está a mostrar o comportamento pretendido. O programador tentou alterar a instrução na linha 19 do código:

```
novo_dept = self
```

para

```
novo_dept = self.copy()
```

mas sem sucesso.

Tentou ainda, depois de importar o módulo *copy*,

```
novo_dept = copy.deepcopy(self)
```

e o programa cancelou em erro.

É necessário que ajude a diagnosticar o que se está a passar, pelo que pedimos a sua ajuda. Tente descobrir e explicar o que não está a funcionar e porquê, e indique, num relatório, um código correto. **Escreva um relatório com o código corrigido, as suas conclusões** (a explicar porque é que as anteriores tentativas de resolução não funcionaram) **e as fontes consultadas.** (Pode usar um notebook)

(Programa fornecido no enunciado)

```
class Dept:
    def __init__(self, nome, divisao):
        self.emps = set()
        # conjunto de empregados
        self.nome = nome
        # nome do departamento
        self.divisao = divisao
        # nome da divisão
        # outrs atributos...

    def add_emp(self, emp):
        """ adiciona empregado a este departamento e atualiza
        a lista de departamentos onde já trabalhou """
        self.emps.add(emp)
        emp.dept.append(self)

    def divide_dept(self, novo):
        """ divide o departamento"""
        novo_dept = self
        # copia o departamento antigo para o novo
        for i in range(int(len(self.emps)/2)):
        # retira metade dos empregados do departamento original
            self.emps.pop()
        restantes = [emp for emp in self.emps]
        # estes são os que ficaram

        for emp in restantes:
        # retira os que ficaram do novo departamento
            novo_dept.emps.remove(emp)

        for emp in novo_dept.emps:
            # adiciona o novo departamento na lista dos
            # departamentos já trabalhados
            emp.muda_dept(novo_dept)
        return novo_dept

    def imprime_emps(self):
        [print(emp.nome, dep.nome)
        for emp in self.emps for dep in emp.dept]


class Empregado:
    def __init__(self, nome):
        self.nome = nome
        self.dept = []

    def muda_dept(self, novo_dept):
        self.dept.append(novo_dept)


if __name__ == '__main__':
    # testes
    dept = Dept('dept1', 'div1')
    [dept.add_emp(Empregado('emp' + str(i))) for i in range(20)]
    dept2 = dept.divide_dept('dept2')
    dept.imprime_emps()
    dept2.imprime_emps()
```

ANEXO II:

(Erro obtido com alteração da linha 19 para `novo_dept = self.copy()`)

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-1-63332d823c40> in <module>
     53     dept = Dept('dept1', 'div1')
     54     [dept.add_emp(Empregado('emp' + str(i))) for i in range(20)]
---> 55     dept2 = dept.divide_dept('dept2')
     56     dept.imprime_emps()
     57     dept2.imprime_emps()

<ipython-input-1-63332d823c40> in divide_dept(self, novo)
     17     def divide_dept(self, novo):
     18         """ divide o departamento"""
---> 19         novo_dept = self.copy()
     20         # copia o departamento antigo para o novo
     21         for i in range(int(len(self.emps)/2)):

AttributeError: 'Dept' object has no attribute 'copy'
```

ANEXO III:

(Erro obtido com alteração da linha 19 para `novo_dept = copy.deepcopy(self)`)

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-2-770079cf6de5> in <module>
     56     dept = Dept('dept1', 'div1')
     57     [dept.add_emp(Empregado('emp' + str(i))) for i in range(20)]
---> 58     dept2 = dept.divide_dept('dept2')
     59     dept.imprime_emps()
     60     dept2.imprime_emps()

<ipython-input-2-770079cf6de5> in divide_dept(self, novo)
     30         for emp in restantes:
     31         # retira os que ficaram do novo departamento
---> 32             novo_dept.emps.remove(emp)
     33
     34         for emp in novo_dept.emps:

KeyError: <__main__.Empregado object at 0x0000021FA81874C8>
```

ANEXO IV:

(Output correto do programa)

```
emp2 dept1
emp18 dept1
emp3 dept1
emp4 dept1
emp19 dept1
emp9 dept1
emp5 dept1
emp6 dept1
emp7 dept1
emp8 dept1
emp14 dept1
emp14 dept2
emp10 dept1
emp10 dept2
emp11 dept1
emp11 dept2
emp15 dept1
emp15 dept2
emp0 dept1
emp0 dept2
emp17 dept1
emp17 dept2
emp12 dept1
emp12 dept2
emp1 dept1
emp1 dept2
emp16 dept1
emp16 dept2
emp13 dept1
emp13 dept2
```

ANEXO V:

(Programa completo e corrigido)

```
import copy


class Dept:
    def __init__(self, nome, divisao):
         self.emps = set()
         # conjunto de empregados
         self.nome = nome
         # nome do departamento
         self.divisao = divisao
         # nome da divisão
         # outrs atributos...

    def add_emp(self, emp):
         """ adiciona empregado a este departamento e atualiza
         a lista de departamentos onde já trabalhou """
         self.emps.add(emp)
         emp.dept.append(self)

    def divide_dept(self, novo):
         """ divide o departamento"""
         novo_dept = copy.deepcopy(self) #cria uma cópia do departamento original
         novo_dept.nome = novo
         novo_dept.emps = set()   # esvazia o conjunto de empregados da cópia do departamento
         for i in range(int(len(self.emps)/2)):
             novo_dept.emps.add(self.emps.pop())  # migra metade dos elementos de self.emps
                                                  # para novo_dept.emps
         for emp in novo_dept.emps:
             # adiciona o novo departamento na lista
             # dos departamentos já trabalhados
             emp.muda_dept(novo_dept)
         return novo_dept

    def imprime_emps(self):
         [print(emp.nome, dep.nome)
         for emp in self.emps for dep in emp.dept]


class Empregado:
    def __init__(self, nome):
         self.nome = nome
         self.dept = []

    def muda_dept(self, novo_dept):
         self.dept.append(novo_dept)


if __name__ == '__main__':
    # testes
    dept = Dept('dept1', 'div1')
    [dept.add_emp(Empregado('emp' + str(i))) for i in range(20)]
    dept2 = dept.divide_dept('dept2')
    dept.imprime_emps()
    dept2.imprime_emps()
```