Desafios de Programação

São propostos a seguir três desafios de programação. As respostas devem ser implementadas em Python 3, e serão executadas em um ambiente contendo apenas as bibliotecas padrão da linguagem. Tal ambiente será o resultante da execução dos seguintes comandos:

```
conda create -y -n myenv python=3.9
conda activate myenv
```

Os enunciados abaixo são versões reduzidas de problemas com que o time de Tecnologia da BWGI se deparou no último ano. Todas as respostas serão avaliadas tanto do ponto de vista funcional, com testes automáticos, quanto sob o aspecto da qualidade do código fonte produzido (clareza, eficiência, complexidade, etc.).

reconcile_accounts

Escreva uma função que faça a conciliação (ou batimento) entre dois grupos de transações financeiras.

Sua função, `reconcile_accounts`, deve receber duas listas de listas (representando as linhas dos dados financeiros) e deve devolver cópias dessas duas listas de listas com uma nova coluna acrescentada à direita das demais, que designará se a transação pôde ser encontrada (`FOUND`) na outra lista ou não (`MISSING`).

As listas de listas representarão os dados em quatro colunas:

- Data (em formato `YYYY-MM-DD`)
- Departamento
- Valor
- Beneficiário

Todas as colunas serão representadas como *strings*.

Dados o arquivo `transactions1.csv`:

```
2020-12-04,Tecnologia,16.00,Bitbucket
2020-12-04,Jurídico,60.00,LinkSquares
2020-12-05,Tecnologia,50.00,AWS
```

e o arquivo `transactions2.csv`:

```
2020-12-04,Tecnologia,16.00,Bitbucket
2020-12-05,Tecnologia,49.99,AWS
2020-12-04,Jurídico,60.00,LinkSquares
```

sua função `reconcile_accounts` deve funcionar do seguinte modo:

```
>>> import csv
>>> from pathlib import Path
>>> from pprint import pprint
>>> transactions1 = list(csv.reader(Path('transactions1.csv').open()))
>>> transactions2 = list(csv.reader(Path('transactions2.csv').open()))
>>> out1, out2 = reconcile_accounts(transactions1, transactions2)
>>> pprint(out1)
[['2020-12-04', 'Tecnologia', '16.00', 'Bitbucket', 'FOUND'],
 ['2020-12-04', 'Jurídico', '60.00', 'LinkSquares', 'FOUND'],
 ['2020-12-05', 'Tecnologia', '50.00', 'AWS', 'MISSING']]
>>> pprint(out2)
[['2020-12-04', 'Tecnologia', '16.00', 'Bitbucket', 'FOUND'],
 ['2020-12-05', 'Tecnologia', '49.99', 'AWS', 'MISSING'],
 ['2020-12-04', 'Jurídico', '60.00', 'LinkSquares', 'FOUND']]
```

Sua função deve levar em conta que em cada arquivo pode haver transações duplicadas. Nesse caso, a cada transação de um arquivo deve corresponder uma única outra transação do outro.

Cada transação pode corresponder a outra cuja data seja do dia anterior ou posterior, desde que as demais colunas contenham os mesmos valores. Quando houver mais de uma possibilidade de correspondência para uma dada transação, ela deve ser feita **com a transação que ocorrer mais cedo**. Por exemplo, uma transação na primeira lista com data 2020-12-25 deve corresponder a uma da segunda lista, ainda sem correspondência, de data 2020-12-24 antes de corresponder a outras equivalentes (a menos da data) com datas 2020-12-25 ou 2020-12-26.

last_lines

Escreva uma função, `last_lines`, que devolva as linhas de um dado arquivo de texto em ordem inversa. Caso esteja familiarizado com a linha de comando Unix, sua função deve fazer o mesmo que o comando `tac`. Por exemplo, dado o seguinte arquivo, `my_file.txt`:

```
This is a file
This is line 2
And this is line 3
```

a função `last_lines` deve funcionar do seguinte modo:

```
>>> for line in last_lines('my_file.txt'):
...     print(line, end='')
...
And this is line 3
```

```
This is line 2
This is a file
```

Note que `last_lines` deve incluir o caractere terminador de linha (suponha `\n`) para cada linha.

Sua função deverá devolver um *iterator*. Por exemplo:

```
>>> lines = last_lines('my_file.txt')
>>> next(lines)
'And this is line 3\n'
>>> next(lines)
'This is line 2\n'
>>> next(lines)
'This is a file\n'
```

Sua função deve ler o arquivo fornecido em pedaços não maiores que o tamanho dado por um argumento da função, cujo valor padrão deve ser `io.DEFAULT_BUFFER_SIZE`. Dito de outro modo, `last_lines` não deve ler mais do que um certo número de *bytes* (número esse dado por um argumento facultativo) de cada vez.

Suponha ainda que o arquivo fornecido possa estar codificado em UTF-8, de modo a conter qualquer caractere Unicode válido. Isso significa que, ao ler e decodificar texto do arquivo, deve se certificar de não decodificar um trecho que contenha um caractere pela metade.

computed_property

Na Goldman Sachs, os chamados *"strats"* contam com um banco de dados orientado a objetos, o Securities Database (ou SecDB), em que as relações de dependência entre as entidades são modeladas em um grafo. Isso permite que o valor armazenado em um certo nó seja considerado válido enquanto suas dependências não sofrerem alteração, o que evita recômputos desnecessários. Por outro lado, ao modificar-se qualquer uma de suas dependências, o nó fica invalidado, e deverá ser recomputado na próxima consulta.

O objetivo é recriar esse mecanismo para objetos Python, em memória. Para isso, escreva um *decorator* chamado `computed_property`, análogo ao `property`. O *decorator* `computed_property` deve aceitar múltiplos atributos dos quais ele depende, e *cachear* o valor da *property* enquanto o valor desses atributos permanecer inalterado.

```
>>> from math import sqrt
>>> class Vector:
...     def __init__(self, x, y, z, color=None):
...         self.x, self.y, self.z = x, y, z
...         self.color = color
...     @computed_property('x', 'y', 'z')
...     def magnitude(self):
...         print('computing magnitude')
...         return sqrt(self.x**2 + self.y**2 + self.z**2)
```

```
...
>>> v = Vector(9, 2, 6)
>>> v.magnitude
computing magnitude
11.0
>>> v.color = 'red'
>>> v.magnitude
11.0
>>> v.y = 18
>>> v.magnitude
computing magnitude
21.0
```

Seu *decorator* deve aceitar como dependêcias atributos que não existam no objeto em questão. Tais atributos devem ser ignorados. Em outras palavras, enquanto um atributo estiver faltando, ele deve ser tratado como inalterado.

```
>>> class Circle:
...     def __init__(self, radius=1):
...         self.radius = radius
...     @computed_property('radius', 'area')
...     def diameter(self):
...         return self.radius * 2
...
>>> circle = Circle()
>>> circle.diameter
2
```

Seu *decorator* deve ainda contemplar os métodos `setter` e `deleter`, a exemplo do que ocorre com `property`.

```
>>> class Circle:
...     def __init__(self, radius=1):
...         self.radius = radius
...     @computed_property('radius')
...     def diameter(self):
...         return self.radius * 2
...     @diameter.setter
...     def diameter(self, diameter):
...         self.radius = diameter / 2
...     @diameter.deleter
...     def diameter(self):
...         self.radius = 0
...
>>> circle = Circle()
>>> circle.diameter
2
>>> circle.diameter = 3
>>> circle.radius
1.5
>>> del circle.diameter
>>> circle.radius
0
```

Seu *decorator* também deve tratar corretamente *docstrings*, da mesma forma como é feito por `property`.

```
>>> class Circle:
...     def __init__(self, radius=1):
...         self.radius = radius
...     @computed_property('radius')
...     def diameter(self):
...         """Circle diameter from radius"""
...         print('computing diameter')
...         return self.radius * 2
...
>>> help(Circle)
Help on class Circle in module __main__:

class Circle(builtins.object)
 |  Circle(radius=1)
 |
 |  Methods defined here:
 |
 |  __init__(self, radius=1)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  diameter
 |      Circle diameter from radius
```