Tarefa de Laboratório 07 - Listas e Strings

Criando um plano infalível e indescobrível

O nosso querido Bairro do Limoeiro sempre foi palco de muita criatividade de todos seus moradores, em especial os de Cebolinha para conseguir afanar Sansão, o icônico coelho azul de pelúcia da Mônica. No entanto, para a felicidade de uns e infelicidade de outros, suas engenhocas aparentemente *infalíveis* acabam sendo bem *falíveis*, seja por descuido na hora de execução, ou um planejamento que não previu certos aspectos.

Pois bem, como todos no Bairro sabem que estamos em um momento que não podemos nos reunir presencialmente, eles precisam fazer isso com auxílio da tecnologia. No entanto, como a imaginação infantil é bem fértil, após ouvir de Cascão um *"E se estiverem nos escutando ou lendo nossas mensagens, Cebolinha?"*, o pequeno estrategista entrou numa introspecção profunda, tentando encontrar uma solução... que acabou tornando-se um de seus novos planos.

Após pesquisar um pouco sobre criptografia, Cebolinha percebeu que muitos dos métodos mais recentes dariam muito trabalho para manusear, então acabou recorrendo à codificações únicas e diretas pra cada caracter, concluindo que, *se fosse algo não ortodoxo, aumentaria as chances de ninguém decifrar, pois jamais pensariam nisso*. Com isso, um conjunto de regras foram criadas e o menino pediu ajuda ao Franjinha para construir um programa que codificasse e decodificasse mensagens, pois *"Se um plano infalível já ela bom, tolnú-lo infalível e indescoblível selia melhor ainda!"*

Franjinha, impressionado ao saber de seus trabalhos nos Laboratórios Repsol e na MAGAMI, pede sua ajuda para a realização deste programa.

O conjunto de regras é o seguinte:

1. As mensagens, codificadas ou não, são sempre lidas de cima pra baixo.
2. Para a codificação, cada caracter da mensagem é substituído pelo seu **valor ASCII associado em Python** convertido para **hexadecimal ou octal**, a depender se o número da mensagem é ímpar ou par:

   2.1. As mensagens de número **ímpar** (linhas 1, 3, 5, etc. da mensagem original) são codificadas *da esquerda pra direita*, substituindo os caracteres pelo valores ASCII em hexadecimal **sem o** `0X`. As letras de `A` até `F` devem ser **maiúsculas**.

   2.2. Já as mensagens de número **par** (linhas 2, 4, 6, etc. da mensagem original) são codificadas de uma forma ligeiramente diferente: primeiro a **mensagem original é invertida** (ou seja, o primeiro caracter vira o último, o segundo vira o penúltimo, etc.), para depois ser codificada *da esquerda pra direita*, substituindo os caracteres pelos valores ASCII em octal **sem o** `0o`.
3. A decodificação segue regras análogas à codificação, obedecendo a numeração das mensagens.

Descrição da entrada

A entrada é composta por duas partes que são dependentes do modo que seu programa deve operar:

- A primeira linha contém três inteiros separados por um espaço em branco: o primeiro diz qual o modo (**M**) que seu programa deve operar, sendo `1` para **codificar** e `2` para **decodificar**. O segundo é o exnerto (**E**), que é o **número fixo de quantos caracteres codificados devem representar um mesmo caracter original** (ver a explicação da razão disso mais adiante). Já o terceiro diz quantas linhas (**L**) a mensagem a ser codificada/decodificada possui.
- As próximas **L** linhas são compostas por mensagens que devem ser codificadas ou decodificadas pelo seu programa, de acordo com o valor de **M** dado. Uma mensagem é um conjunto de caracteres que termina com uma quebra de linha. Você pode assumir que caracteres especiais como letras acentuadas **NÃO** aparecerão nas mensagens, codificadas ou não. Considere também que a contagem das linhas **sempre começa em 1**, ou seja, a primeira linha é de número 1 e não 0.

Um exemplo de entrada seria (EX1):

```
1 3 2
Reuniao hoje as 16h
Nos vemos no Zoom!
```

Outro exemplo seria (EX2):

```
2 4 2
0045006E00740065006E00640069006400F002E
00410141015400400163015701550145016201410164016301 05
```

Descrição da saída

A saída é composta por **L** linhas, com as mensagens codificadas ou decodificadas **mantendo a mesma ordem da entrada**. O caracter de enxerto utilizado tanto na codificação quanto na decodificação **será sempre o número 0**.

A saída do EX1 seria:

```
0520650750 6E0690 6106F02006806F06A0650200610730200310360 68
0411551571571320401571560401631571551451660401631571 16
```

Já a saída do EX2 seria:

```
Entendido.
Estaremos la!
```

Importante: *Por que esse "enxerto" é necessário?*

A resposta curta é: para que seu programa saiba identificar exatamente o que está tentando decodificar. Agora vamos ao porquê disso:

Existem um total de 128 caracteres ASCII codificados no Python que você pode acessar o número pela função `ord()`, você pode identificá-los aqui. Repare que tem uma tabela com os valores em inteiro, binário, octal e hexadecimal já exemplificados. Vamos pegar um exemplo:

O caracter que representa o espaço em branco é o de número `32` em inteiro, que fica `100000` em binário, `40` em octal e `20` em hexadecimal. Já o caracter que representa o `A` é o de número `65` em inteiro, que fica `1000001` em binário, `101` em octal e `41` em hexadecimal.

Repare que, ignorando as regras específicas desse laboratório, se convertêssemos direto pra octal sem o `0o` a mensagem "`A A`" ficaria: "`101040101`". Agora suponha que você mesmo tenha que decodificar essa mensagem, por onde começaria? Pegaria os caracteres um a um? Dois a dois? Não tem como saber!

Por isso o enxerto é importante: ele diz de quantos em quantos caracteres você precisa olhar pra decodificar uma mensagem codificada. No entanto, repare que se pegarmos simplesmente "`101040101`" com um exerto de `3` também não daria certo, pois o espaço em branco foi codificado apenas com dois caracteres. Por isso devemos usar a informação sobre o enxerto tanto na codificação quanto na decodificação. Na codificação ele só é importante quando **o código do caracter convertido possui um número de caracteres menor que o enxerto**, sendo necessário "completá-lo", no nosso caso, com o número 0, na esquerda. Repare que o enxerto *jamais pode ser* ***menor*** *que o mínimo de dígitos necessários para a codificação de um caracter.*

Usando o mesmo exemplo aqui, com o enxerto aplicado desde a codificação, teríamos:

"`A A`" = "`101 040 101`" e poderíamos ler de 3 em 3 que obteríamos a mensagem original de volta (os espaços em branco foram deixados para facilitar a leitura, eles não estão presentes na codificação).

Dicas para esta tarefa

Seu *caderninho de dicas*, sempre presente no seu bolso, está com novas páginas que podem te ajudar nesta tarefa:

Lendo os valores de entrada:

Assim como nos demais laboratórios, quando uma entrada tem mais de um dado, separados por um caracter em comum, você pode usar a `split()` em conjunto com o `input()` para pegar os valores.

Lembre-se que a `input()` retorna o valor lido como `string` e a `split()` retorna uma lista com os valores separados em cada posição e na ordem que foram lidos.

Obtendo o valor ASCII de um caracter e vice-versa:

Dado um caracter, você pode usar a função `ord()` para pegar seu valor ASCII, que é um inteiro na base 10. Com um mesmo inteiro na base 10, você pode usar a função `chr()` para obter o caracter correspondente. Veja um exemplo feito no terminal do Python:

```
>>> caracter = 'z'
>>> a = ord(caracter)
>>> print(a)
122

>>> numB10 = 77
>>> b = chr(numB10)
>>> print(b)
M
```

Convertendo entre bases no Python:

Neste laboratório **não** é esperado que você faça as conversões manualmente. Você pode converter um número na base 10 para a base 16 (hexadecimal) usando a função `hex(num)`. Já para converter da base 10 para a base 8, pode usar a função `oct(num)`. Em ambas funções, `num` é um número inteiro, e o valor retornado é uma `string`. Veja alguns exemplos:

```
>>> numB10 = 12648243
>>> numB16 = hex(numB10)
>>> print(numB16)
0xc0ff33

>>> numB8 = oct(numB10)
>>> print(numB8)
0o60177463
```

Repare que as `string` retornadas em `numB8` e `numB16` **estão** com os prefixos `0o` e `0x`, respectivamente. E, mais especificamente, `numB16` está com as letras **minúsculas**... tenha isso em mente se for usá-las na sua solução.

E para converter de volta? Podemos usar a própria função `int()` temos usado desde o começo! Ela na verdade pode receber dois parâmetros: `int(x, base)` em que `x` é o que se deseja converter (podendo ser um `int` ou uma `string`), e `base` é a base em que `x` está. Por padrão, `base = 10` é utilizado, caso você não especifique. Vamos usar o mesmo exemplo convertendo de volta:

```
>>> numB16 = "0xc0ff33"
>>> numB10 = int(numB16, 16)
>>> print(numB10)
12648243

>>> numB8 = "60177463"
>>> numB10 = int(numB8, 8)
>>> print(numB10)
12648243
```

Repare que a `int()`, quando especificado a `base` como segundo parâmetro, **não precisa** ter o prefixos `0x` ou `0o` presentes no primeiro parâmetro `x`, como mostrado propositalmente no exemplo acima.

Obtendo o inverso de uma string:

Uma forma de obter o inverso de uma `string` é com a noção de *slices* vistos em aula:

```
>>> bat = "Eu sou a noite"
>>> tab = bat[::-1]
>>> print(tab)
etion a uos uE
```

Preenchendo valores com zeros à esquerda:

Dada uma `string`, você pode usar o método `zfill(num)` para retornar a mesma `string` preenchida com zeros à esquerda de tal forma que ela contenha um total de `num` caracteres. Aqui temos algumas observações:

`zfill()` é um **método** da classe `string` do Python, então sua forma de utilizá-lo é um pouco diferente:

```
>>> strOriginal = '7'
>>> strPreenchida = strOriginal.zfill(3)
>>> print(strOriginal)
7
>>> print(strPreenchida)
007
```

Repare que o `zfill()`, embora precise ser chamado pela variável que contém a `string` original (`strOriginal`) **não** altera seu valor. Uma outra observação é que se o número passado no método `zfill()` for **menor ou igual** que a quantidade de caracteres que a `string` original possui, *nada acontece*.

```
>>> strOriginal = 'E4'
>>> strPreenchida = strOriginal.zfill(2)
>>> print(strOriginal)
E4
>>> print(strPreenchida)
E4
```