**Nesta aula…**

# Conteúdo



# 1 Listas em compreensão

**Processamento de listas**

Duas operações comuns:

- selecionar elementos que verificam uma condição
- calcular uma expressão para cada elemento

**Calcular quadrados**

Usando um ciclo para construir a lista dos quadrados:

```
def quadrados(xs):
    "Calcula o quadrados de cada elemento de xs."
    ys = []
    for x in xs:
       ys =  ys+[x**2]
    return ys
```

```
>>> quadrados(range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

**Listas em compreensão**

Calculando os quadrados usando uma *expressão em compreensão*:

```
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Notação inspirada na teoria de conjuntos:

$$\{x^2 : x \in \{0, 1, \ldots, 9\}\}$$

**Sintaxe**

```
[expressão for variável in sequência]
```

```
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> [(c,ord(c)) for c in "pedro"]
[('p', 112), ('e', 101), ('d', 100),
 ('r', 114), ('o', 111)]
```

**Listas em compreensão com condições**

```
[expr for variável in sequência if condição]
```

Exemplo: quadrados dos múltiplos de 3 menores que 10.

```
>>> [x**2 for x in range(10) if x%3==0]
[0, 9, 36, 81]
```

**Listas em compreensão embricadas**

Podemos usar uma expressão em compreensão dentro de outra.

Por exemplo, a *matriz identidade* de $4 \times 4$:

```
>>> [[int(i==j) for j in range(4)]
                for i in range(4)]
[[1, 0, 0, 0],
 [0, 1, 0, 0],
 [0, 0, 1, 0],
 [0, 0, 0, 1]]
```

**Compreensões com múltiplas sequências**

Análogo ao producto cartesiano de dois conjuntos:

```
>>> [(x,y) for x in "ABC" for y in range(4)]
[('A', 0), ('A', 1), ('A', 2), ('A', 3),
 ('B', 0), ('B', 1), ('B', 2), ('B', 3),
 ('C', 0), ('C', 1), ('C', 2), ('C', 3)]
```

A ordem do resultado depende da ordem das sequências:

```
>>> [(x,y) for y in range(4) for x in "ABC"]
[('A', 0), ('B', 0), ('C', 0), ('A', 1),
 ('B', 1), ('C', 1), ('A', 2), ('B', 2),
 ('C', 2), ('A', 3), ('B', 3), ('C', 3)]
```

**Sintaxe geral**

```
[expr for var1 in seq1 if cond1
      for var2 in seq2 if cond2
      ⋮
      for varN in seqN if condN]
```

# 2 Exemplo: obfuscar um texto

**Exemplo: obfuscar um texto**

- um método para evitar que um texto electrónico seja lido inadvertidamente
- usando em grupos de discussão e forums para ocultar conteúdos (comentários sobre jogos, filmes, livros, etc.)
- não é um segredo: qualquer pessoa pode descodificar o texto se quiser

**ROT13: *ROTate by 13 places***

Cada letra é transformada noutra à distância de 13 posições:

$$\overbrace{\begin{array}{cccccccccccccc} A & B & C & D & E & F & G & H & I & J & K & L & M \end{array}}^{\text{13 letras}}$$

$$\text{ROT13} \quad \begin{array}{cccccccccccccc} \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow \\ N & O & P & Q & R & S & T & U & V & W & X & Y & Z \end{array}$$

$$\begin{array}{ccccccc} & H & E & L & L & O \\ \text{ROT13} & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ & U & R & Y & Y & B \end{array}$$

Para descodificar basta aplicar ROT13 outra vez:

$$\begin{array}{ccccccc} & U & R & Y & Y & B \\ \text{ROT13} & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ & H & E & L & L & O \end{array}$$

**Implementar ROT13**

- uma função em Python `rot13(txt)` para transformar uma cadeia de caracteres `txt` usando ROT13
- espaços, algarismos e símbolos de pontuação ficam inalterados
- letras maiúsculas e minúsculas devem ser transformadas

**Ideia**

- uma função auxiliar `rot13char` para transformar apenas um caracter
- aplicamos `rot13char` a todos os caracteres de `txt`
- vamos usar expressões em compreensão

**Transformar um caracter**

```
def rot13char(c):
  "Transforma um caracter por rotação 13."
  # será letra maiúscula ou mínuscula?
  if (c>='a' and c<='m') or (c>='A' and c<='M'):
      return chr(ord(c) + 13)
  elif (c>='n' and c<='z') or (c>='N' and c<='Z'):
      return chr(ord(c) - 13)
  else:
      return c  # outros símbolos: inalterados
```

**Transformar uma cadeia (1ª tentativa)**

```
def rot13(txt):
  "Transforma txt por rotaçao 13."
  return [rot13char(c) for c in txt]

>>> rot13('HELLO')
['U', 'R', 'Y', 'Y', 'B']
>>> rot13('Ola, mundo!')
['B', 'y', 'n', ',', ' ', 'z', 'h', 'a',
 'q', 'b', '!']
```

O resultado é uma *lista* em vez de uma *cadeia de caracteres*.

Precisamos de uma função para concatenar uma lista de cadeias...

**Transformar uma cadeia (corrigido)**

```
def concat(lista):
   "Concatena uma lista de cadeias."
   conc = ""   # resultado da concatenação
   for txt in lista:
       conc = conc+txt
   return conc

def rot13(txt):
  "Transforma txt por rotaçao 13."
  return concat([rot13char(c) for c in txt])
```

**Exemplos**

```
>>> rot13("Python e' uma linguagem poderosa!")
"Clguba r' hzn yvathntrz cbqrebfn!"

>>> rot13("Clguba r' hzn yvathntrz cbqrebfn!")
"Python e' uma linguagem poderosa!"
```

**Variante**

- quero tornar o texto ainda mais obfuscado!
- os sinais de pontuação e espaços dão indicação sobre onde começam e acabam as palavras
- vamos modificar a função para remover todos os caracteres *excepto* as letras
- podemos fazer isso usando uma condição na expressão de compreensão

**Modificação para remover não-letras**

```
import string

def rot13(txt):
  "Transforma txt por rotaçao 13."
  return concat([rot13char(c) for c in txt
                   if c in string.letters])
# string.letters == 'abcdef... ABCDEF...'
```

O resto do programa fica inalterado!

**Execução**

```
>>> rot13("Python e' uma linguagem poderosa!")
"Clgubarhznyvathntrzcbqrebfn"

>>> rot13("Clgubarhznyvathntrzcbqrebfn")
"Pythoneumalinguagempoderosa"
```

Não obtemos exactamente o texto orignal, mas ainda é compreensível...

**Sumário**

- expressões em compreensão são uma notação sucinta para dois tipos comuns de processamento de listas:
  1. selecionar elementos por uma condição
  2. calcular uma expressão para cada elemento
- é possível fazer as mesmas operações apenas com ciclos
- as expressões em compreensão são mais sucintas e simples
- apenas algumas linguagens de programação permitem expressões em compreensão: Haskell, Python, Perl 6, C# 3.0, F#...