Programação I
Aula 11 — Mais algoritmos sobre listas

Pedro Vasconcelos
DCC/FCUP

Nesta aula

1. Eliminar repetidos
2. Crivo de Eratóstenes
3. Listas em compreensão: extras

Eliminar elementos repetidos

Vamos definir uma função para **eliminar os elementos repetidos** de uma lista.

Exemplo: se a lista original for

```
[2, 1, 3, 4, 1, 4, 1]
```

então o resultado deve ser a lista

```
[2, 1, 3, 4]
```

em que cada valor ocorre apenas uma vez e pela mesma ordem da lista original.

Eliminar elementos repetidos

Vamos implementar duas soluções para eliminar repetidos:

1. **construir uma lista nova** sem repetidos;
2. **modificar a lista original** apagando os repetidos.

Eliminar repetidos: Algoritmo 1

Seja *xs* a lista original:

1. Construimos uma nova lista *ys*
2. Inicialmente $ys = []$
3. Para cada elemento *x* na lista dada:
   se *x* não ocorre ainda em *ys*, então acrescentamos *x* à lista *ys*

Eliminar repetidos: Implementação 1

```
def elimrep1(lista: List[Any]) -> List[Any]:
    "Constroi uma nova lista sem repetidos."
    # uma nova lista (inicialmente vazia)
    nova: List[Any] = []
    # iterar sobre os elementos da lista dada
    for x in lista:
        # se x ainda não ocorreu
        if not (x in nova):
            # acrescenta à nova lista
            nova.append(x)
    # retorna a lista sem repetidos
    return nova
```

Exemplos

```
>>> elimrep1([2,1,3,3,1,4,1])
[2, 1, 3, 4]

>>> elimrep1([2,2,2])
[2]

>>> elimrep1([1,2,3])
[1, 2, 3]
```


Eliminar repetidos: Algoritmo 2

Seja *xs* a lista original:

1. Fazermos um ciclo sobre os índices *i*
2. Enquanto *i* é menor que o comprimento de *xs*:
   - Se $xs[i]$ ocorre em $xs[0], x[1], \ldots, xs[i-1]$
     então removemos $xs[i]$;
     caso contrário, incrementamos *i*
3. *Invariante do ciclo*: não há repetidos entre 0 e *i*

$$\overbrace{xs[0], \quad \ldots, \quad xs[i-1],}^{\text{sem repetidos}} \overbrace{xs[i], \quad \ldots, \quad xs[n-1]}^{\text{pode haver repetidos}}$$

4. No final do ciclo: $i = \texttt{len}(xs)$ e a lista não tem repetidos

Eliminar repetidos: Implementação 2

```
def elimrep2(lista: List[Any]):
    "Remover repetidos da lista dada."
    i = 0
    while i<len(lista):
        if lista[i] in lista[0:i]:
            # elemento repetido; apagamos
            del lista[i]
        else:
            # não é repetido; avançamos
            i = i+1
```

NB: esta função modifica a lista dada (não retorna uma nova lista).

Exemplos

```
>>> lista = [2,1,3,3,1,4,1]
>>> elimrep2(lista)
>>> lista
[2, 1, 3, 4]

>>> lista = [2,2,2]
>>> elimrep2(lista)
>>> lista
[2]

>>> lista = [1,2,3]
>>> elimrep2(lista)
>>> lista
[1, 2, 3]
```

Sumário

As duas soluções resolvem o mesmo problema de formas distintas.

`elimrep1`
- constroi uma nova lista;
- é mais simples de compreender.

`elimrep2`
- modifica a lista original;
- é mais eficiente (não necessita de criar uma nova lista).

Crivo de Eratóstenes

Construir a tabela dos números primos até n:

1. escrevemos todos os inteiros de 2 até n
2. o primeiro número (2) é primo e riscamos todos os seus múltiplos
3. o segundo número (3) é primo e riscamos todos os seus múltiplos
4. repetimos o processo: o próximo número não riscado é primo e riscamos todos os seus múltiplos
5. no fim: a tabela contém todos os números primos menores do que n

Crivo de Eratóstenes

Começamos com todos os inteiros de 2 até 30 O número 2 é primo e riscamos os seus múltiplos O número 3 é primo e riscamos os seus múltiplos O número 4 não é primo (já foi riscado) O número 5 é primo e riscamos os seus múltiplos Repetimos o processo até esgotar a tabela; restam apenas os primos inferiores a 30.

		2	3	4	5	6	7	8	9	10	11				
		12	13	14	15	16	17	18	19	20	21				
		22	23	24	25	26	27	28	29	30					
2	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~	11	**2**	**3**	~~4~~	5	~~6~~	7
~~12~~	13	~~14~~	15	~~16~~	17	~~18~~	19	~~20~~	21	~~12~~	13	~~14~~	~~15~~	~~16~~	17
~~22~~	23	~~24~~	25	~~26~~	27	~~28~~	29	~~30~~		~~22~~	23	~~24~~	25	~~26~~	~~27~~

Implementação do crivo

- Representamos a tabela de números por uma lista
- Removemos da lista os números compostos
- No final restam apenas os primos

Implementação do crivo

```
def crivo(lista: List[int]):
    "Remove números compostos duma lista."
    i = 0
    while i<len(lista):
        p = lista[i] # p é primo
        j = i+1      # vamos remover múltiplos
        while j<len(lista):
            if lista[j]%p == 0:
                del lista[j]
            else:
                j = j+1
        i = i+1      # próximo número
    # fim da função crivo
```

Esta função modifica a lista dada (não retorna um resultado).

Construir uma lista de primos

```
def primos(n: int) -> List[int]:
    "Constroi a lista de primos menores que n."
    numeros = list(range(2,n)) # entre 2 e n-1
    crivo(numeros)             # remove os compostos
    return numeros             # retorna os primos
```

Exemplos

Calcular todos os primos até 100:

```
>>> primos(100)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,
 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Quantos primos inferiores a 1000?

```
>>> len(primos(1000))
168
```


Listas em compreensão embricadas

Considere a matriz triangular superior 4×4:

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Temos que

$$A_{ij} = \begin{cases} 1\,, & \text{se } i \leq j \\ 0 & \text{caso contrário.} \end{cases}$$

Listas em compreensão embricadas (cont.)

Podemos construir esta matriz usando uma lista em compreensão dentro de outra:

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

As compreensões correspondem a *dois ciclos encaixados*:

$$[[\overbrace{\texttt{int(i<=j)}}^{A_{ij}}\overbrace{\texttt{for j in range(4)}}^{\text{colunas}}]\ \overbrace{\texttt{for i in range(4)}}^{\text{linhas}}]$$

Compreensões com vários geradores

Também podemos usar compreensões com vários geradores.

O resultado é uma só lista com *todas* as combinações de elementos:

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

Outro exemplo

```
>>> cidades = ['porto', 'lisboa', 'coimbra']
>>> cursos = ['mat', 'cc', 'fis']

>>> [x+' '+y for x in cursos for y in cidades]
['mat porto', 'mat lisboa', 'mat coimbra', 'cc porto',
'cc lisboa', 'cc coimbra', 'fis porto', 'fis lisboa',
'fis coimbra']
```

A ordem dos resultados depende da ordem dos geradores:

```
>>> [x+' '+y for y in cidades for x in cursos]
['mat porto', 'cc porto', 'fis porto', 'mat lisboa',
'cc lisboa', 'fis lisboa', 'mat coimbra', 'cc coimbra',
'fis coimbra']
```