Fundamentos de Programação 2014-15

Diogo Gomes dgomes@ua.pt
João Rodrigues jmr@ua.pt

Listas

- Exemplos de listas:
  - [10, 20, 30, 40]
  - [‘sapo’, ‘macaco’, ‘porco da índia’]
  - [‘coisas’, 2.0, 5, [10, 20]]
- Lista vazia
  - []
- Aceder a um elemento de uma lista
  - L = [1,2,4,8,16]

```
>>> print L[1]
2
```

Listas são mutáveis

```
>>> Numeros = [1,2,4,4]

>>> Numeros[2] = 3

>>> print Numeros

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

- O operador **in** pode ser usado nas listas

```
>>> queijos = [‘serra’, ‘mozarella’, ‘flamengo’]

>>> ‘flamengo’ in queijos

True

>>> ‘brie’ in queijos

False
```

Atravessar uma lista

```
for queijo in queijos:

    print queijo
```

- range() devolver uma lista de numeros
- Mesmo que uma lista tenha outras listas contidas, o **for** só atravessa os elementos da lista em questão e não os elementos das listas internas.
- Uma lista vazia [] não é iteravel, mas também não constitui um erro

Operações com listas

- Concatenação

```
>>> a = [1,2,3]
>>> b = [4,5,6]
>>> print a+b
[1,2,3,4,5,6]
>>> [0] * 4
[0,0,0,0]
>>> [1,2,3] * 3
[1,2,3, 1,2,3, 1,2,3]
```

Operações com listas

- Partição

```
>>> t = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]

>>> t[1:3]

[‘b’, ‘c’]

>>> t[:4]

[‘a’, ‘b’, ‘c’, ‘d’]

>>> t[3:]

[‘d’, ‘e’, ‘f’]
```

Operações com listas

- Métodos
  - lista.append(elemento)
  - lista.extend(outra_lista)
  - lista.sort()
  - x = lista.pop(posicao)
  - del lista[posicao]
  - del lista[posicao:posicao2]
  - lista.remove(elemento)

Listas e Strings

```
>>> s = ‘spam’
>>> t = list(s)
>>> print t
[‘s’, ‘p’, ‘a’, ‘m’]
>>> f = ‘a vida custa Costa’
>>> t = f.split()
>>> print t
[‘a’, ‘vida’, ‘custa’, ‘Costa’]
```

- string.split(delimiter)

Dicionários

- São como as listas, mas os índices não precisam de ser inteiros.
- Um dicionário mapeia um índice (**chave**) num valor.
- A associação de uma chave a um valor chama-se de **item**

```
>>> en2pt = dict()
>>> print en2pt
{}
```

Acrescentar itens

- Pode usar []

```
>>> en2pt[‘one’] = ‘um’
```

- Pode inicializar com uma estrutura

```
>>> en2pt = {‘one’: ‘um’, ‘two’: ‘dois’,
‘three’: ‘tres’}
```

- O operador **in** indica se algo é uma chave do dicionário

```
>>> ‘one’ in en2pt
True
>>> ‘um’ in en2pt
False
```

Listar

- Pode listar todas as chaves de um dicionário

  >>> en2pt.keys()

- Ou listar todos os valores

  >>> en2pt.values()

Tuplos

- Tuplos são uma sequencia de valores

```
>>> t = ‘a’, ‘b’, 1, ‘ola’
```

- A maioria dos operadores sobre listas também funcionam com tuplos

```
>>> t[1:3]

(‘b’, 1)
```

- Mas não podemos modificar os elementos! (TypeError)

Tuplos, Listas e Dicionários

- Função **zip** pega em duas ou mais sequencias e junta elemento a elemento

```
>>> s = ‘abc’
>>> t = [0, 1, 2]
>>> zip(s,t)

[(‘a’, 0), (‘b’, 1), (‘c’, 2)]
```

- A funçao items, dado um dicionário retorna uma lista de tuplos

```
>>> d = {‘a’: 0, ‘b’: 1, ‘c’: 2}
>>> t = d.items()
>>> print t

[(‘a’, 0), (‘b’, 1), (‘c’, 2)]
```

Ficheiros

- Já sabemos ler!
- Abrir um ficheiro com intenções de escrita:

```
fout = open('output.txt', 'w')
```

- Escrever para um ficheiro

```
fout.write('Hello World\n')
```

- Fechar o ficheiro

```
fout.close()
```

Nomes de ficheiros e caminhos

- O modulo “os” disponibiliza funções para manipular ficheiros/directorias
- Directorio actual:

```
>>> import os
>>> cwd = os.getcwd()
>>> print cwd
```

/home/utilizador

- Caminho completo de um ficheiro

```
>>> os.path.abspath(‘file.txt’)
```

‘/home/utilizador/file.txt’

Nomes de ficheiros e caminhos

- Verificar se um ficheiro existe

```
>>> os.path.exists('file.txt')
True
```

- Verificar se um nome é uma directoria

```
>>> os.path.isdir('file.txt')
False
```

- Listar os ficheiros numa directoria

```
>>> os.listdir(cwd)
['Documentos', 'Musica', 'Fotos', 'file.txt']
```

Instrução de atribuição

- Podemos usar a operação de atribuição para decompor estruturas
- Exemplo:
  - `triplo = (1, 2, 3)`
  - `(i, j, k) = triplo`
    - Como resultado, `i=1`, `j=2`, `k=3`

Expressões Lambda

- São expressões cujo valor é uma função
- Sintaxe:

```
lambda lista_argumentos:  expressao
```

- Exemplos:

  `f = lambda x : x+1`
  - Função que dado um valor `x`, devolve `x+1`

  `m = lambda x, y : x + y`
  - Função que calcula a soma de dois elementos

A função map()

- Permite aplicar uma função a cada elemento de uma sequência

```
r = map(func, seq)
```

- Primeiro argumento é uma função e o segundo uma sequencia (lista, string, etc)

```
>>> Celsius = [39.2, 36.5, 37.3, 37.8]
>>> Fahrenheit = map(lambda x: (float(9)/5)*x + 32, Celsius)
>>> print Fahrenheit
[102.56, 97.700000000000003, 99.140000000000001,
100.03999999999999]
```

A função filter()

- Permite filtrar os elementos de uma sequencia com base numa função

```
r = filter(func, seq)
```

- Primeiro argumento é uma função que retorna um valor booleano e o segundo uma sequencia (lista, string, etc). O resultado será uma nova sequencia cujos os elementos processados por **func** geraram o valor **True.**

```
>>> fib = [0,1,1,2,3,5,8,13,21,34,55]
>>> result = filter(lambda x: x % 2, fib)
>>> print result
[1, 1, 3, 5, 13, 21, 55]
```

A função reduce()

- Permite reduzir os elementos de uma sequencia a um unico com base numa função

```
r = reduce(func, seq)
```

- Primeiro argumento é uma função que recebe dois argumentos e retorna apenas um. O resultado será apenas um valor resultado da aplicação sucessiva de **func** sobre todos elementos da sequencia.

```
>>> reduce(lambda x,y: x+y, [47,11,42,13])

113
```

Listas de compreensão

- Do inglês "*list comprehension*"
- Mecanismo compacto para processar alguns ou todos os elementos numa lista
  - Pode ser aplicado a listas, tuplos e cadeias de caracteres
  - O resultado é uma lista
- Síntaxe:

```
[<expr> for <var> in <sequência> if <condição>]
```

Codificar/Descodificar JSON

- Formato JSON permite facilmente trocar informação de forma estruturada entre aplicações.
- Converter uma lista ou dicionario:

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
```

- Importar de um objecto JSON:

```
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]‘)
['foo', {'bar': ('baz', None, 1.0, 2)}]
```

Pesquisa

- Linear ou Sequencial
- Percorremos a nossa estrutura de dados atravessando todos os elementos sequencialmente até encontrar o nosso elemento.

```
a = [5,2,6,7,42,8,4,1]
for e in a:
    if e == 42:
        print “encontrei o 42!”
```

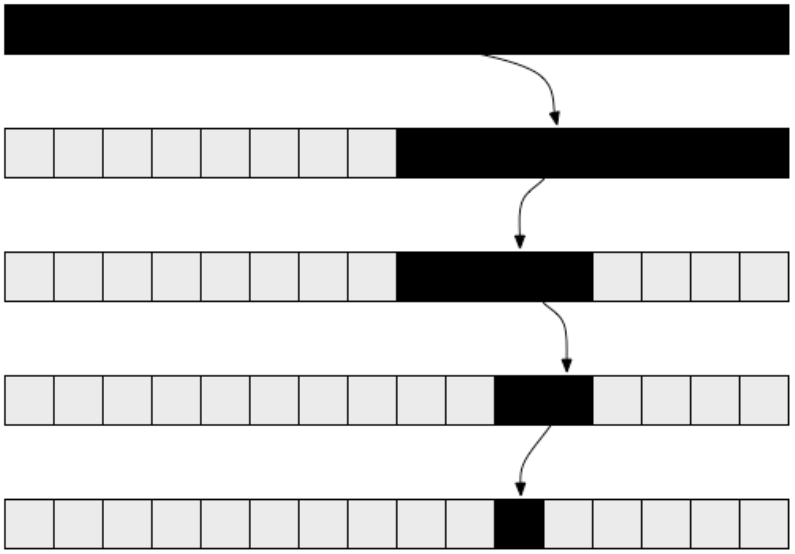
Pesquisa

- Atravessar um lista recuperando o indice:

```
a = [5,2,6,7,42,8,4,1]
for i, e in enumerate(a):
    if e == 42:
        print “encontrei o 42!”
        print “na posição {}”.format(i)
```

Pesquisa binária

- “Dividir para conquistar”
- Dada uma lista ordenada



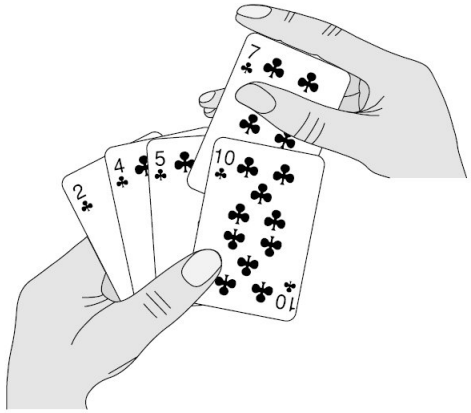
Pesquisa

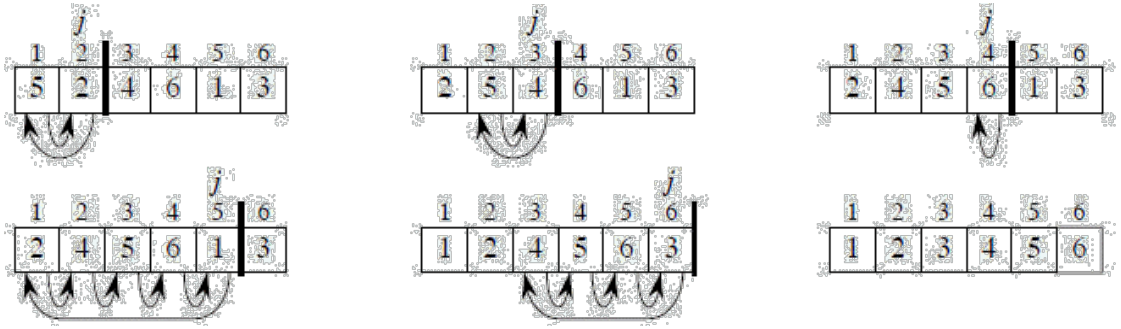
- Binária
- Apenas válida em listas ordenadas!

```
def binSearch(x, lst):
  first = 0
  last = len(lst)    # primeiro que
não pode ser solução
  while first < last:
    mid = (first+last)//2
    if x <= lst[mid]:
      last = mid
    else:
      first = mid+1
  return first
```

Ordenação – Insertion Sort

- Ordenação por **inserção (insertion sort)**





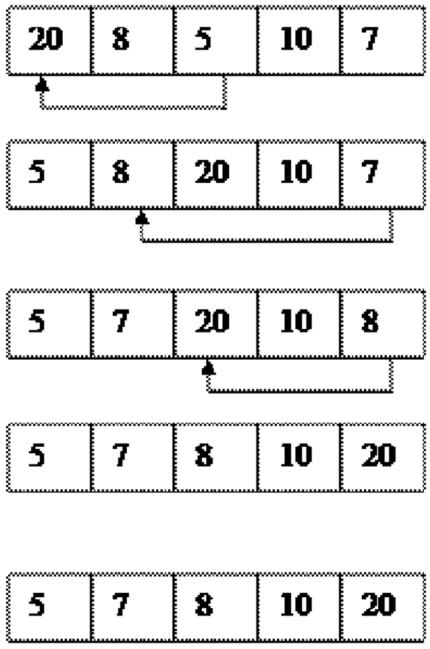
Ordenação - Insertion Sort

- Para cada elemento da estrutura de dados comparamos com os anteriores até encontrar a sua posição. Neste processo vamos trocando de posição com os elementos com os quais vamos comparando.

```
def insertionSort(lista):
    for i in xrange(1, len(lista)):
        j = i-1
        s = lista[i]
        while j>=0 and lista[j] > s:
            lista = swap(lista, j, j+1)
            j-=1
        lista[j+1]=s
    return lista
```

Ordenação – Selection Sort

- Ordenação por **seleção (selection sort)**



Ordenação – Selection Sort

- Para cada posição da estrutura de dados procuramos o menor/maior elemento em toda a estrutura. Fazemos uma troca entre esses dois elementos.

```
def selectionSort(lista):
    for i in xrange(0, len(lista)):
        m = min(lista[i:])
        lista = swap(lista, i, i+m)
    return lista
```

Ordenação

- Quicksort
- “Dividir para Conquistar”

```
def quickSort(lista):
    if len(lista) <= 1:
        return lista

    pivot = lista[0]
    prv, nxt = [], []

    for i in lista[1:]:
        if i < pivot:
            prv.append(i)
        else:
            nxt.append(i)

    return quickSort(prv) \
           + [pivot] + quickSort(nxt)
```

Ordenação

- Quicksort
- Usando listas de compreensão:

```
def quickSort(lista):
    if len(lista) <= 1:
        return lista

    return
        quicksort([p for p in
lista[1:] if p < lista[0]]) +
        [lista[0]] +
        quicksort([n for n in
lista[1:] if n > lista[0]])
```