Programação e Base de Dados /Computação para Deteção Remota

Introdução à linguagem Python

Sérgio Crisóstomo,
(adaptado dos slides de Rita Ribeiro,
Pedro Vasconcelos e João Pedro Pedroso)

2022/2023

Departamento de Ciência de Computadores

U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Conteúdo

Listas

Listas

- Sequências ordenadas possivelmente com repetições
- Podem conter elementos de quaisquer tipos
- Os elementos são identificados pelos índices

Listas por extensão

- Lista com *n* elementos: `[e1, e2, ..., en]`
- A ordem é significativa
- Podem ocorrer elementos repetidos
- Pode ser a lista vazia: `[]`

Operações básicas

- comprimento

```
>>> len([1,'dois',3])
3
```

- concatenação

```
>>> [1,'dois',3]+[4,5,6]
[1,'dois',3,4,5,6]
```

- repetição

```
>>> 2*[1,'dois',3]
[1,'dois',3,1,'dois',3]
```

- pertença

```
>>> 3 in [1,'dois',3]
True
```

- iteração

```
>>> for x in [1,'dois',3]:
        print(x)
1
dois
3
```

Acesso aos elementos

- Operador de indexação: `lista[i]`
- Índices entre 0 e `len(lista)-1`
- Índices negativos: acesso a partir do fim
- Índices inválidos dão um erro de execução

Exemplo

```
>>> alimentos = ['pão', 'pão', 'queijo', 'queijo']
>>> alimentos[0]
'pão'
>>> alimentos[1]
'pão'
>>> alimentos[2]
'queijo'
>>> len(alimentos)
4
```

Fatias

`lst[i:j]`	elementos entre i e $j-1$ inclusíve
`lst[i:]`	elementos entre i até ao final
`lst[:j]`	elementos do primeiro até $j-1$ inclusíve
`lst[:]`	todos os elementos (cópia da lista)

```
>>> vogais = ['a','e','i','o','u']
>>> vogais[1:4]
['e', 'i', 'o']
>>> vogais[:3]
['a','e','i']
>>> vogais[3:]
['o','u']
>>> vogais[:]
['a', 'e', 'i', 'o', 'u']
```

Fatias (cont.)

Forma geral

`lst[i:j:k]` elementos de i a $j-1$ com incrementos k

Incrementos negativos: percorrer a lista ao contrário.

```
>>> vogais[::2]        # índices pares
['a','i','u']
>>> vogais[1::2]       # índices ímpares
['e','o']
>>> vogais[::-1]       # inverter a lista
['u', 'o', 'i', 'e', 'a']
```

Percorrer os índices e elementos

```
for i in range(len(lista)):
    print(i, lista[i])
```

- Ciclo sobre índices i de 0 até $len(lista) - 1$
- Elemento $lista[i]$ associado ao índice i

Percorrer todos os elementos

```
for valor in lista:
    print(valor)
```

- Evita manipular explicitamente o índice
- Preferível quando necessitamos dos valores mas não dos índices

Listas são mutáveis

Podemos modificar ou acrescentar elementos:

```
>>> beatles = [1, 2, 3]
>>> beatles[0] = "john"
>>> beatles[2] = "ringo"
>>> beatles
['john', 2, 'ringo']

>>> beatles[1:2] = ['paul', 'george']
>>> beatles
['john', 'paul', 'george', 'ringo']
```

Remover elementos duma lista

```
>>> beatles = ['john', 'paul', 'george', 'ringo']
>>> del beatles[0]
>>> beatles
['paul', 'george', 'ringo']
```

Alternativa:

```
>>> beatles = ['john', 'paul', 'george', 'ringo']
>>> beatles[0:1] = []
>>> beatles
['paul', 'george', 'ringo']
```

Nomes e objectos

É importante distinguir o nome da lista dos valores associados.

Nomes e objectos (1)

Dois nomes, duas listas separadas:

```
>>> a = [1,2,3]
>>> b = [1,2,3]
>>> a[0] = 'oops'
>>> print(a, b)
['oops', 2, 3] [1, 2, 3]
```

Nomes e objectos (2)

Dois nomes, apenas uma lista:

```
>>> a = [1,2,3]
>>> b = a
>>> a[0] = 'oops'
>>> print(a, b)
['oops', 2, 3] ['oops', 2, 3]
```

Nomes e objectos (3)

Dois nomes, duas listas (fazendo uma cópia):

```
>>> a = [1,2,3]
>>> b = a[:]
>>> a[0] = 'oops'
>>> print(a, b)
['oops', 2, 3] [1, 2, 3]
```

Métodos sobre listas

Alguns métodos pré-definidos:

`append` acrescentar um elemento ao final

`insert` acrescentar um elemento numa posição

`remove` remover um elemento

`sort` ordenar os elementos por ordem crescente

- Utilização: `lista.método(argumentos)`
- Modificam a lista

Exemplos

```
>>> beatles = ['john','paul']
>>> beatles.append('george')
>>> beatles.append('ringo')
>>> beatles
['john', 'paul', 'george', 'ringo']
>>> beatles.insert(0, 'paul')
>>> beatles
['paul', 'john', 'paul', 'george', 'ringo']
>>> beatles.sort()
>>> beatles
['george', 'john', 'paul', 'paul', 'ringo']
```

Para obter mais informação:

```
>>> help(list)
```

Listas dentro de listas

- As listas podem conter outras listas
- Podemos assim representar tabelas ou matrizes

```
>>> matriz = [[1,2,-1],
              [3,1,0],
              [0,1,-2]]
>>> matriz[1][0]
3
>>> matriz[1][0] = -3
>>> matriz
[[1, 2, -1], [-3, 1, 0], [0, 1, -2]]
```

Tuplos

Tuplos

- Sequências ordenadas de elementos:

```
(e1, e2, ..., en)
```

- Acesso aos elementos por índices
- Ao contrário das listas, os tuplos são imutáveis

Operações básicas

- comprimento

```
>>> len(('Pedro',12))
2
```

- concatenação

```
>>> ('Pedro,12)+('João',14)
('Pedro,12,'João',14)
```

- repetição

```
>>> 2*('Pedro',12)
('Pedro',12,'Pedro',12)
```

- pertença

```
>>> 12 in ('Pedro',12)
True
```

- iteração

```
>>> for x in ('Pedro',12):
        print(x)
Pedro
12
```

Acesso aos elementos

```
>>> nota = ('Pedro', 12)
>>> nota[0]
'Pedro'
>>> nota[1]
12
>>> nota[0] = 'Joao'
TypeError: 'tuple' object does not support
item assignment
```

Atribuição a tuplos de variáveis

```
>>> (x,y) = (5,-7)
>>> x
5
>>> y
-7
```

Ou simplesmente:

```
>>> x,y = 5,-7
>>> x
5
>>> y
-7
```

Listas e tuplos combinados

Vamos representar uma agenda como uma lista de pares *nome*/*email*:

```
[('Maria João', 'mj@mail.pt'),
 ('José Manuel', 'jm@mail.pt'),
 ('João Pedro', 'jp@mail.pt')]
```

Operações:

- acrescentar uma entrada (nome e email)
- procurar email pelo nome

Acrescentar uma entrada

```
def acrescentar(agenda, nome, email):
    "Acrescentar um nome e email a agenda."
     agenda.append((nome,email))
```

Procurar um nome (1)

```
def procurar(agenda, txt):
    "Procurar emails por parte do nome."
    emails = []
    for par in agenda:
        if txt in par[0]:          # txt ocorre no nome?
            emails.append(par[1]) # acrescenta email
    return emails
```

Procurar um nome (2)

```
def procurar(agenda, txt):
    "Procurar emails por parte do nome."
    emails = []
    for (nome,email) in agenda:
        if txt in nome:          # txt ocorre no nome?
            emails.append(email) # acrescenta email
    return emails
```

Exemplos

```
>>> agenda = []
>>> acrescentar(agenda, "Maria João",
                "mj@mail.pt")
>>> acrescentar(agenda,"João Pedro",
                "jp@mail.pt")

>>> procurar(agenda, "Maria")
['mj@mail.pt']

>>> procurar(agenda, "João")
['mj@mail.pt', 'jp@mail.pt']
```

Usar listas ou tuplos?

- Utilizamos listas para sequências mutáveis (e.g. uma agenda)
- Utilizamos tuplos para sequências imutáveis (e.g. um par nome, telefone)
- Os tuplos são necessários em casos especiais: (e.g. *chaves de dicionários* — próximas aulas)
- Podemos sempre converter entre os dois tipos de sequência:

  list(...) converter para lista
  tuple(...) converter para tuplo

Agregações

Somar valores duma sequência

A função pré-definida `sum` soma os valores duma sequência.

Exemplo:

```
>>> sum([0.5, 1, 0.5])
2.0
```

Vamos definir a nossa própria implementação desta função.

Somar valores duma sequência

```
def soma(xs):
    "Somar valores duma lista xs."
    # variável 'total' vai acumular a soma
    total = 0
    # percorre todos os valores
    for x in xs:
        # acrescenta ao total
        total = total + x
    # o resultado da função é 'total'
    return total
```

Exemplos

```
>>> soma([1, 2, 3])
6

>>> soma([1.0, 0.5, 2.0])
3.5

>>> soma([1.0])
1.0

>>> soma([])
0
```

Outras agregações

- A média aritmética
- O produto
- O máximo ou mínimo

Média aritmética duma sequência

```
def media(xs):
    "Calcula a média aritmética duma lista."
    # média = soma / número de elementos
    return soma(xs)/len(xs)
```

Exemplos

```
>>> media([2.0, 1.0, 1.0])
1.33333333333

>>> media([2.0])
2.0
```

A média não está definida para a lista vazia:

```
>>> media([])
ZeroDivisionError: integer division or
modulo by zero
```

Produto duma lista

Análogo à soma, substituindo:

- o operador $+$ por $\star$;
- o valor inicial 0 por 1 (elemento neutro de $\star$).

Produto duma lista

```
def produto(xs):
    "Produto duma lista xs."
    # variável 'total' vai acumular o producto
    total = 1
    # percorrer todos os valores
    for x in xs:
        # acrescenta ao total
        total = total * x
    # retorna o resultado
    return total
```

Exemplos

```
>>> produto([2, 3, 4])
24

>>> produto(range(1,5))
24

>>> produto([])
1
```

Valor máximo duma lista

- Guardar o maior valor já encontrado
- Inicialmente é o primeiro elemento da sequência
- Operação de agregação: tomar o máximo

Valor máximo duma lista

```
def maximo(xs):
    "Maior valor duma sequência xs."
    # inicialmente: maior é o primeiro valor
    maior = xs[0]
    # percorrer os restantes elementos
    for x in xs[1:]:
        maior = max(maior, x)
    # retorna o maior
    return maior
```

Exemplos

```
>>> maximo([1.0, 2.5, 2.0, -1.0])
2.5

>>> maximo([2.0])
2.0
```

O máximo não está definido para a sequência vazia:

```
>>> maximo([])
IndexError: list index out of range
```

Eliminar repetidos

Eliminar elementos repetidos

Vamos definir uma função para eliminar os elementos repetidos duma lista.

Duas soluções:

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

Eliminar repetidos (1)

- Construir uma nova lista *ys*
- Inicialmente $ys = [\,]$
- Para cada elemento *x* na lista dada:
  se $x \notin ys$ acrescenta *x* a *ys*

Eliminar repetidos (1)

```
def elimrep1(xs):
    "Constrói uma nova lista sem repetidos."
    # ys é a lista sem repetidos
    # inicialmente vazia
    ys = []
    for x in xs:
        # se x não é repetido
        if not (x in ys):
            # acrescenta a nova lista
            ys.append(x)
    # retorna a lista sem repetidos
    return ys
```

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 (2)

- Uma lista xs com n elementos
- Um ciclo sobre os índices $0 \leq i \leq n-1$
- Invariante: não há repetidos nos índices inferiores a i

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

- Actualização: se $xs[i] \in \{xs[0], xs[1], \ldots, xs[i-1]\}$ então apaga $xs[i]$; senão $i \leftarrow i+1$

Eliminar repetidos (2)

```
def elimrep2(xs):
    "Eliminar repetidos em xs."
    i = 0
    while i<len(xs):
        if xs[i] in xs[0:i]:
            # xs[i] é repetido: apaga-o
            del xs[i]
        else:
            # xs[i] não é repetido: avança
            i = i+1
```

Note que o `elimrep2` modifica a lista `xs` em vez de retornar 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

- Ambas as soluções são correctas
- A primeira solução:
  - constrói uma nova lista;
  - é mais simples de compreender.
- A segunda solução:
  - modifica a lista original;
  - pode ser mais eficiente (necessita de menos memória).

Listas em compreensão

Listas em compreensão

É muito comum construir uma lista partindo de uma outra:

- selecionando elementos usando uma condição;
- aplicando uma transformação a cada elemento.

Exemplo: calcular quadrados

Construir a lista dos quadrados dos números inteiros de 1 a 9.

Exemplo: calcular quadrados (cont.)

Solução usando um ciclo for:

```
lista = []
for x in range(1, 10):
    lista.append(x**2)
print(lista)
```

Solução usando uma lista em compreensão:

```
lista = [x**2 for x in range(1,10)]
print(lista)
```

Sintaxe

```
[x**2 for x in range(1,10)]
```

Notação inspirada na teoria de conjuntos:

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

Mais geralmente:

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

Mais exemplos

```
>>> [i**2 for i in [2,3,5,7]]
[4, 9, 25, 49]

>>> [1+x/2 for x in [0, 1, 2]]
[1.0, 1.5, 2.0]

>>> [ord(c) for c in "ABCDEF"]
[65, 66, 67, 68, 69, 70]

>>> [len(s) for s in
     "As armas e os barões assinalados".split()]
[2, 5, 1, 2, 6, 11]
```

Listas em compreensão com condições

Exemplo: quadrados dos múltiplos de 3 inferiores a 10.

```
[x**2 for x in range(10) if x%3==0]
```

Mais geralmente:

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

Outro exemplo

Um número natural *n* é *primo* se não tem divisores próprios (i.e. maiores do que 1 e menores do que *n*).

Para testar se *n* é primo podemos testar se a lista dos divisores próprios é vazia.

Testar primos

```
def primo(n):
    # lista dos divisores próprios
    divs = [d for d in range(2,n) if n%d==0]
    # n é primo se e só se a lista for vazia
    return len(divs)==0
```

Nota: esta solução é ineficiente (calcula *todos* os divisores em vez de terminar após encontrar o primeiro...)

Listas em compreensão embricadas

Podemos usar uma lista em compreensão dentro de outra.

Exemplo:

```
>>> [[i*j for j in range(1,11)] for i in range(1,11)]
```

produz a matriz da multiplicação:

```
[[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
 [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
 [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
...
 [9, 18, 27, 36, 45, 54, 63, 72, 81, 90],
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]]
```

Compreensões com múltiplas sequências

A ordem do resultado depende da ordem das sequências:

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

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

Sintaxe geral

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

Dicionários

Dicionários

- Estrutura de dados para tabelas de associações
- Cada chave é associada a um (e um só) valor
- Analogia: dicionário bilingue (e.g. português-inglês)
- Podemos usar como chave:
  - números
  - cadeias de caracteres
  - tuplos
  - combinações dos anteriores

  (apenas *tipos imutáveis*)
- A ordem dos pares (chave, valor) pode não ser a esperada, já que é estabelecida por algoritmos desenhados com vista ao acesso rápido aos elementos

Exemplo: um inventário

Um inventário que associa *quantidades disponíveis* a *frutas*.

Item	Quantidade
bananas	25
peras	12
laranjas	10

Exemplo: um inventário (cont.)

Poderiamos representar como uma lista de pares:

```
[('bananas',25), ('laranjas',10), ('peras',12) ]
```

Problemas:

- necessário percorrer a lista para procurar o valor associado a uma chave
- permite duplicar chaves; por exemplo

  ```
  [('bananas',1), ('bananas',25),
   ('laranjas',10), ('peras',12),]
  ```

  Representa 1, 25 ou 26 bananas?

Criar um dicionário

```
>>> inv = {'bananas':25,'laranjas':10,'peras':12}
```

Podemos consultar valores apartir da chave:

```
>>> inv['bananas']
25
>>> inv['bananas'] = inv['bananas'] + 1
>>> inv
{'laranjas':10, 'peras':12, 'bananas':26}
```

Criar um dicionário (cont.)

Mais alguns exemplos:

```
>>> emails = { 'Maria João':'mj@mail.pt',
               'João Pedro':'jp@mail.pt' }

>>> dirs = { 'N':(0,1), 'S':(0,-1),
             'W':(-1,0), 'W':(1,0) }

>>> vazio = {}      # inicializar um dicionário vazio
```

Algumas operações sobre dicionários

- Obter o valor associado à chave (erro se não existir)

  `dict[chave]`

- Atribuir um valor a uma chave

  `dict[chave] = valor`

- Testar existência de uma chave (resultado: True ou False)

  `chave in dict`

Exemplos

```
>>> inv = {'bananas':25,'laranjas':10,'peras':12}
>>> inv['bananas']
25

>>> inv['kiwis']
KeyError: 'kiwis'

>>> 'bananas' in inv
True
>>> 'kiwis' in inv
False
```

Mais operações

Obter o valor ou *default* se a *chave* não existir:

```
dict.get(chave,default)
```

Exemplos:

```
>>> inv = {'bananas':25,'laranjas':10,'peras':12}
>>> inv.get('bananas',0)
25
>>> inv.get('kiwis',0)
0
```

Mais operações (cont.)

Percorrer todas as chaves:

```
for chave in dict.keys():
    ...
```

Exemplo:

```
>>> inv = {'bananas':25,'laranjas':10,'peras':12}
>>> for fruit in inv.keys():
...     print(fruit)

bananas
peras
laranjas
```

Nota: as chaves não estão ordenadas!

Mais operações (cont.)

Obter uma lista com todas as chaves:

```
list(dict.keys())
```

Exemplo:

```
>>> inv = {'bananas':25,'laranjas':10,'peras':12}
>>> list(inv.keys())
['bananas', 'peras', 'laranjas']
```

Como anteriormente as chaves não estão ordenadas.

Mais operações (cont.)

Percorrer todos os pares de chaves e valores:

```
for chave,valor in dict.items():
    ...
```

Exemplo:

```
>>> inv = {'bananas':25,'laranjas':10,'peras':12}
>>> for fruit,quant in inv.items():
...     print(fruit,quant)

bananas 25
peras 12
laranjas 10
```

Mais operações (cont.)

Percorrer todos os pares de chaves e valores, ordenados por chave:

```
for chave in sorted(dict):
    ...
```

Exemplo:

```
>>> inv = {'bananas':25,'laranjas':10,'peras':12}
>>> for fruit in sorted(inv):
...     print(fruit,inv[fruit])

bananas 25
laranjas 10
peras 12
```

Contar ocorrências

Contar ocorrências de letras

Problema: contar a ocorrência de cada letra do alfabeto num ficheiro de texto.

Vamos usar um dicionário para associar cada letra à sua contagem.

Uma tabela deste género chama-se um *histograma*.

Um texto de exemplo

O canto I d'*Os Lusíadas*[1].

lusiadas_cantoI.txt

```
1

As armas e os barões assinalados,
Que da ocidental praia Lusitana,
Por mares nunca de antes navegados,
Passaram ainda além da Taprobana,
Em perigos e guerras esforçados,
...
```

[1] Fonte: Instituto Camões `http://www.instituto-camoes.pt`.

Histograma de ocorrências

```
def histograma(fich):
    f = open(fich, "r")  # abrir ficheiro para leitura
    count = {}           # inicializa contagem vazia
    while True:          # ler o texto linha-a-linha
        linha = f.readline().lower()
        if linha == '':
            break
        for c in linha:
            if c>='a' and c<='z':
                count[c] = 1+count.get(c,0)
    f.close()
    return count
```

Contar ocorrências de letras

```
>>> hist = histograma("lusiadas_CantoI.txt")
>>> hist
{'a': 3015, 'c': 658, 'b': 249, 'e': 3064,
 'd': 1217, 'g': 378, 'f': 240, 'i': 1154,
 'h': 239, 'j': 103, 'm': 1033, 'l': 572,
 'o': 2622, 'n': 1294, 'q': 390, 'p': 531,
 's': 1828, 'r': 1578, 'u': 1031, 't': 1229,
 'v': 425, 'y': 1, 'x': 29, 'z': 68}
```

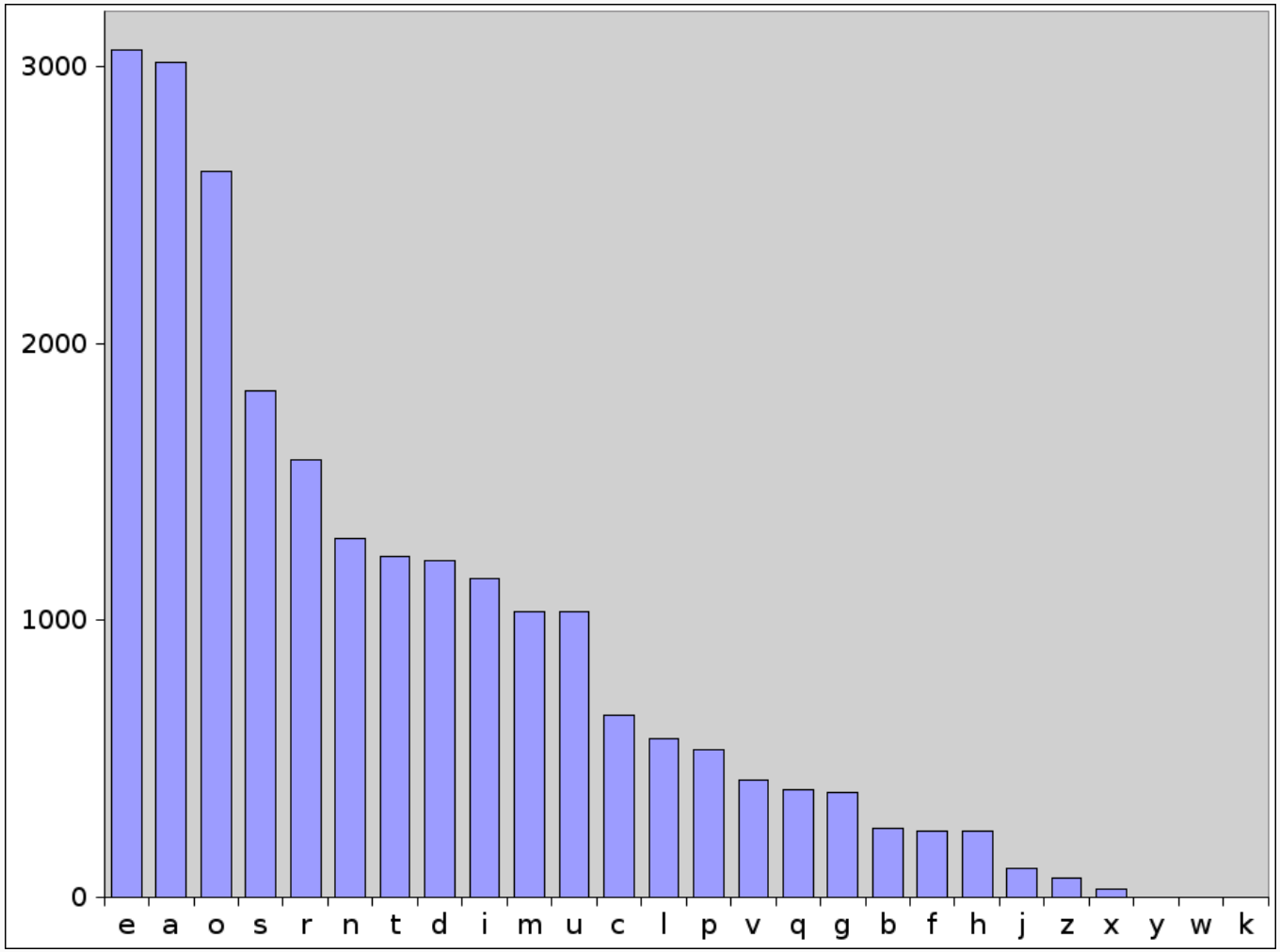
Visualização

Imprimir uma tabela de ocorrências de letras:

```
hist = histograma("lusiadas_CantoI.txt")
# ciclo sobre as chaves
for c in hist.keys():
    print(c, hist[c])       # letra, contagem
```

Podemos importar para uma folha de cálculo e traçar um gráfico.

Gráfico do histograma



Observações

- A letra `'e'` é a que ocorre mais vezes (3064)
- A letra `'y'` ocorre apenas 1 vez
- Contudo: os resultados são incorretos porque não contabilizamos letras acentuadas!
- Para tratar esses casos: vamos converter letras acentuadas em não acentuadas.

Normalização

Em Unicode, uma letra acentuada pode ser representada de duas formas:

NFC um código numérico composto;
por exemplo, ç é U+00C7 (LATIN CAPITAL LETTER C WITH CEDILLA);

NFD um par de códigos da letra e do acento;
por exemplo, ç é U+0043 (LATIN CAPITAL LETTER C) U+0327 (COMBINING CEDILLA).

Normalização (cont.)

O módulo `unicodedata` define uma função `normalize` permite converter entre estas formas.

`normalize('NFC', str)` converte *str* para a forma composta.

`normalize('NFD', str)` converte *str* para a forma decomposta.

Para ignorar os acentos vamos usar a normalização NFD:

‘ã’ ⟶ ‘a’ + ‘~’
‘á’ ⟶ ‘a’ + ‘´’
‘Ç’ ⟶ ‘C’ + ‘¸’
⋮

Histograma de ocorrências (2)

```
from unicodedata import normalize

def histograma(fich):
    f = open(fich, "r")   # abrir ficheiro para leitura
    count = {}            # inicializa contagem vazia
    while True:           # ler o texto linha-a-linha
        linha = f.readline().lower()
        if linha == '':
            break
        for c in normalize('NFD',linha):
            if c>='a' and c<='z':
                count[c] = 1+count.get(c,0)
    f.close()
    return count
```

Contagem revista

```
{'a': 3296, 'c': 739, 'b': 249, 'e': 3180,
 'd': 1217, 'g': 378, 'f': 240, 'i': 1210,
 'h': 239, 'j': 103, 'm': 1033, 'l': 572,
 'o': 2707, 'n': 1294, 'q': 390, 'p': 531,
 's': 1828, 'r': 1578, 'u': 1042, 't': 1229,
 'v': 425, 'y': 1, 'x': 29, 'z': 68}
```

A letra mais frequente é o `'a'` seguido do `'e'`.

Sumário

- Dicionários permitem representar tabelas de associações
- A ordem entre as entradas não é significativa
- Pesquisa pela chave é mais eficiente do que sobre uma lista (de pares)