字典与集合

车万翔

哈尔滨工业大学

什么是字典（Dictionary）？

- 一系列“键-值（key-value）”对
- 通过“键”查找对应的“值”
- 类似纸质字典，通过单词索引表找到其相应的定义
  - C++: map、Java: HashTable or HashMap
- 例如：电话本

姓名（键）	电话号码（值）
John	86411234
Bob	86419453
Mike	86412387
……	……

字典的使用

- **创建字典**
  - 使用 { } 创建字典
  - 使用 : 指明 键:值 对
    - my_dict = {'John': 86411234, 'Bob': 86419453,'Mike': 86412387}
  - 键必须是不可变的且不重复，值可以是任意类型
- **访问字典**
  - 使用 [ ] 运算符，键作为索引
    - print my_dict['Bob']
    - print my_dict['Tom'] #WRONG!
    - 增加一个新的对
      - my_dict['Tom'] = 86417639

字典运算符和方法

- **len(my_dict)**
  - 字典中键-值对的数量
- **key in my_dict**
  - 快速判断 key 是否为字典中的键：O(1)
  - 等价于 my_dict.has_key(key)
- **for key in my_dict:**
  - 枚举字典中的键，注：键是无序的
- **更多的方法**
  - my_dict.items() – 全部的键-值对
  - my_dict.keys() – 全部的键
  - my_dict.values() – 全部的值
  - my_dict.clear() – 清空字典

示例：字母计数

- **读取一个字符串，计算每个字母出现的个数**
- **方案一**
  - 生成 26 个变量，代表每个字母出现的个数
- **方案二**
  - 生成具有 26 个元素的列表，将每个字母转化为相应的索引值，如 a ➔ 0，b ➔ 1，…

```
count = [0] * 26
for i in 'abcdad':
    count[ord(i) - 97] += 1
```

- **方案三**
  - 生成一个字典，字母做键，对应出现的次数做值

```
count = {}
for i in 'abcdad':
    if i in count:
        count[i] += 1
    else:
        count[i] = 0
```

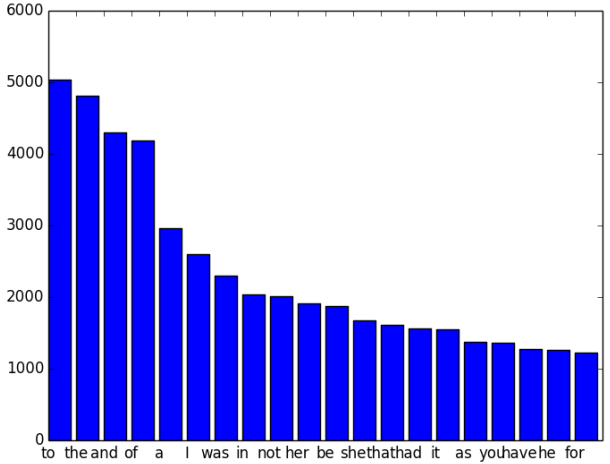
示例：单词计数

- **读取小说"emma.txt"，打印前 10 个最常见单词**
  - 是否还能直观的将每个单词转化为相应的数字？

```
f = open('emma.txt')
word_freq = {}
for line in f:
    words = line.split() # split a line by blanks
    for word in words:
        if word in word_freq:
            word_freq[word] += 1
        else:
            word_freq[word] = 1

word_freq_lst = []
for word, freq in word_freq.items():
    word_freq_lst.append((freq, word))

word_freq_lst.sort(reverse = True)

for freq, word in word_freq_lst[:10]:
    print word, freq
```

示例：单词计数

- 图形输出
  - Zipf 分布
    - 一个单词出现的频率与它在频率表里的排名成反比




```
26 import matplotlib.pyplot as plt
27
28 y = [i[1] for i in word_freq_lst[:20]]
29 words = [i[0] for i in word_freq_lst[:20]]
30 x_pos = range(len(words))
31
32 plt.bar(x_pos, y)
33 plt.xticks(x_pos, words)
34 plt.show()
35
```

示例：翻转字典

- **生成一个新字典，其键为原字典的值，值为原字典的键**
  - 同一个值，可能对应多个键，需要用列表存储

```
def invert_dict(d):
    inverse = {}
    for key in d:
        val = d[key]
        if val in inverse:
            inverse[val].append(key)
        else:
            inverse[val] = [key]
    return inverse
```

集合（Set）

- **集合**
  - 无序不重复元素（键）集
  - 和字典类似，但是无“值”
- **创建**
  - x = set()
  - x = {key1, key2, ...}
- **添加和删除**
  - x.add('body')
  - x.remove('body')
- **集合的运算符**

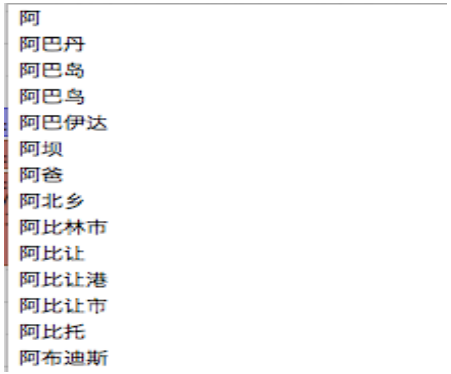
运算符	含义
-	差集
&	交集
|	并集
!=	不等于
==	等于
in	成员
for key in set	枚举

示例：中文分词

- **我爱北京天安门。➔ 我/爱/北京/天安门/。**
- **算法：正向最大匹配**
  - 从左到右取尽可能长的词
  - 如：研究生命的起源 ➔研究生/命/的/起源
    - “研究生”是词，且比“研究”更长

示例：中文分词

- 加载词典：lexicon.txt




```
def load_dic(filename):
    f = open(filename)
    word_dic = set()
    max_length = 1
    for line in f:
        word = unicode(line.strip(), 'utf-8')
        word_dic.add(word)
        if len(word) > max_length:
            max_length = len(word)
    return max_length, word_dic
```

示例：中文分词

正向最大匹配分词

```
def fmm_word_seg(sentence, word_dic, max_length):
    begin = 0
    words = []
    sentence = unicode(sentence, 'utf-8')

    while begin < len(sentence):
        for end in range(min(beg + max_len, len(sent))
                                        , beg, -1):
            word = sentence[begin:end]
            if word in word_dic or end == begin + 1:
                words.append(word)
                break
        begin = end
    return words
max_len, word_dic = load_dic('lexicon.dic')
words = fmm_word_seg(raw_input(), word_dic, max_len)
for word in words:
    print word
```

数据结构综合应用示例

哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

- **马尔科夫（Markov）分析**
  - 给定一系列单词，下一个单词出现的概率
- **如马丁路德金的演讲：I have a dream**
    - *I have a dream that one day this nation will ...*
    - *I have a dream that one day on the red hills ...*
    - *I have a dream that one day even the state ...*
  - "have a" 后面总是跟着 "dream"
  - "one day" 后面可能跟 "this", "on" 或 "even"
- **给定任意 n 个单词作为前缀，然后随机生成下一个单词，新的单词和前面 n-1 个单词组成新的前缀，并重复以上过程，生成一段随机文本**

生成马尔科夫词典

- 任意 n-1 个连续单词，获得其曾经出现过的所有后缀单词（列表）

```
def create_markov_dict(n):
    f = open('emma.txt')
    markov_dict = {}
    prefix = ('',) * n
    for line in f:
        words = line.strip().split()
        for suffix in words:
            if prefix in markov_dict:
                markov_dict[prefix].append(suffix)
            else:
                markov_dict[prefix] = [suffix]
            prefix = prefix[1:] + (suffix,)
    return markov_dict
```

马尔科夫分析

- 根据输入的 n 个单词，生成一段随机文本

```
def markov_analysis(start, markov_dict, word_num):
    markov_lst = list(start)
    prefix = start
    for i in range(word_num):
        suffix = random.choice(markov_dict[prefix])
        markov_lst.append(suffix)
        prefix = prefix[1:] + (suffix,)
    return markov_lst
markov_dict = create_markov_dict(2)
print markov_analysis(('I', 'have'), markov_dict, 100)
```

数据结构对比

	string	list	tuple	set	dict
Mutable					
Sequential					
Sortable					
Slicable					
Index/key type					
Item/value type					
Search complexity					

数据结构对比

	string	list	tuple	set	dict
Mutable	No	Yes	No	Yes	Yes
Sequential	Yes	Yes	Yes	No	No
Sortable	No	Yes	No	No	No
Slicable	Yes	Yes	Yes	No	No
Index/key type	Int	Int	Int	Immut	Immut
Item/value type	Char	Any	Any	No	Any
Search complexity	O(n)	O(n)	O(n)	O(1)	O(1)