Python程序设计与实践

第七课：字典、递归

2025. 4

Today

- 字典的定义与用法
- 字典的主要方法
- 字典应用实战
- 递归的意义与场景
- 递归的编写与使用

什么是字典：

- 假如我们想存储和使用一组学生的成绩信息，使用之前学到的应该怎么做？
- 可以使用两个list，分别存放学生姓名和成绩

```
names = ['Ana',  'John', 'Matt', 'Katy']
grades = ['A+' ,  'B' ,   'A' ,   'A' ]
```

- 可以通过索引位置，间接地访问某个学生及其成绩

- 什么是字典：
  - 可以用嵌套list：

```
eric = ['eric', ['ps', [8, 4, 5]], ['mq', [6, 7]]]
ana = ['ana', ['ps', [10, 10, 10]], ['mq', [9, 10]]]
john = ['john', ['ps', [7, 6, 5]], ['mq', [8, 5]]]
grades = [eric, ana, john]
```

```
def get_grades(who, what, data):
    for stud in data:
        if stud[0] == who:
            for info in stud[1:]:
                if info[0] == what:
                    return who, info
```

代码过于复杂

什么是字典：

- 更好的数据结构：
  - 使用一个数据对象，不用分成多个list
  - 可以通过自定义的方式索引并获取其中的数据

0	Elem 1
1	Elem 2
2	Elem 3
3	Elem 4
…	…

list

Key 1	Val 1
Key 2	Val 2
Key 3	Val 3
Key 4	Val 4
…	…

字典

什么是字典：

- 字典是由键（key）-值（value）对构成的数据结构
  - Key：唯一的标识符
  - Value：由key绑定的数据
- 现实中事物的类比
  - 电话薄：姓名（keys）+电话号码（values）
  - 英文字典：首字母（keys）+单词定义（values）
  - 居民身份系统：身份证号（keys）+居民个人信息（values）

- Python中的字典：
  - 字典数据类型（Dict）
    - 由大括号表示 { … }
    - key和value之间由冒号分隔
    - 每一对key/value由逗号分隔

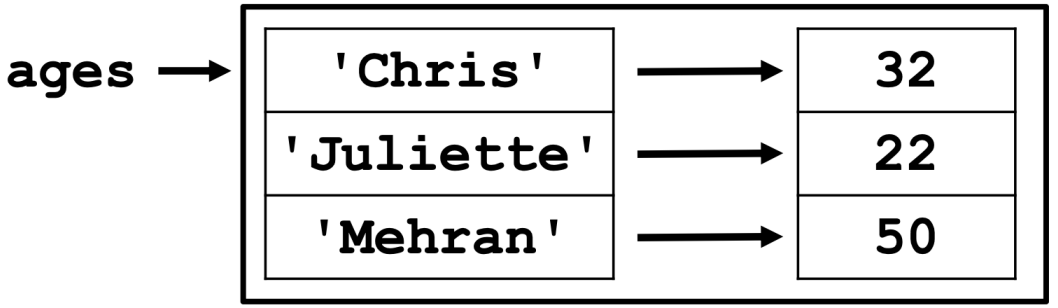
```
ages = {'Chris': 32, 'Juliette': 22, 'Mehran': 50}

squares = {2: 4, 3: 9, 4: 16, 5: 25}

phone = {'Pat': '555-1212', 'Jenny': '867-5309'}

empty_dict = {}
```

Dictionary

- Python中的字典：
  - 字典数据类型（Dict）

```
ages = {'Chris': 32, 'Juliette': 22, 'Mehran': 50}
```

  - 类似于一系列由key索引形成的变量

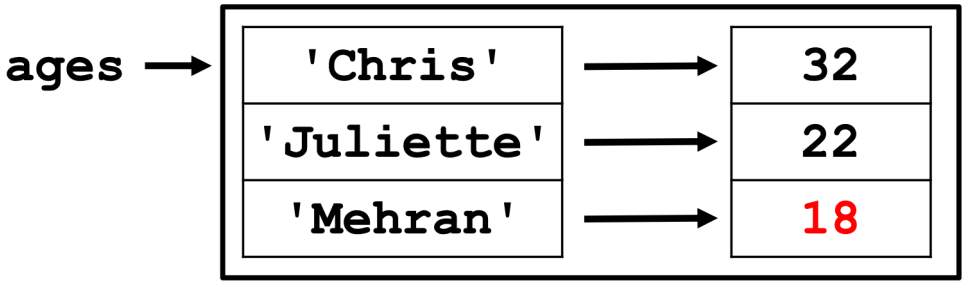


  - 可以通过key获取被索引的value：
    - `ages['Chris']` → `32`
    - `ages['Mehran']` → `50`

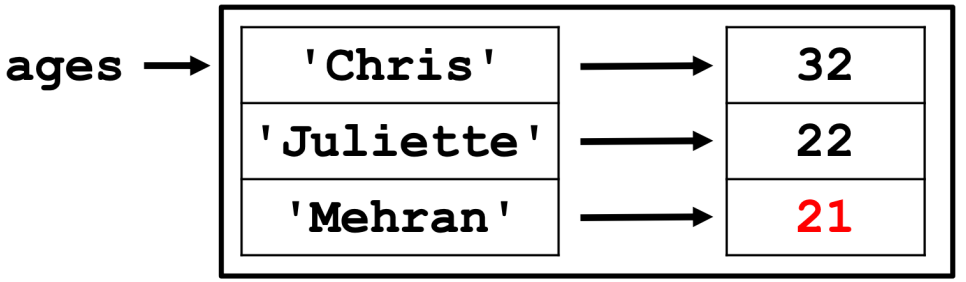
- Python中的字典：
  - 字典数据类型（Dict）

```
ages = {'Chris': 32, 'Juliette': 22, 'Mehran': 50}
```

  - 类似于一系列由key索引形成的变量



  - 可以将value当作一般的变量来使用：
    - `ages['Mehran'] = 18`
    - `ages['Mehran'] += 3`



Dictionary

- Python中的字典：
  - 字典数据类型（Dict）

```
ages = {'Chris': 32, 'Juliette': 22, 'Mehran': 50}
```

  - 类似于一系列由key索引形成的变量

  - 通过key获取value要注意key是否存在：

```
>>> juliettes_age = ages['Juliette']
>>> juliettes_age
22
>>> santas_age = ages['Santa Claus']
KeyError: 'Santa Claus'
```

Dictionary

- Python中的字典：
  - 字典数据类型（Dict）

```
ages = {'Chris': 32, 'Juliette': 22, 'Mehran': 50}
```

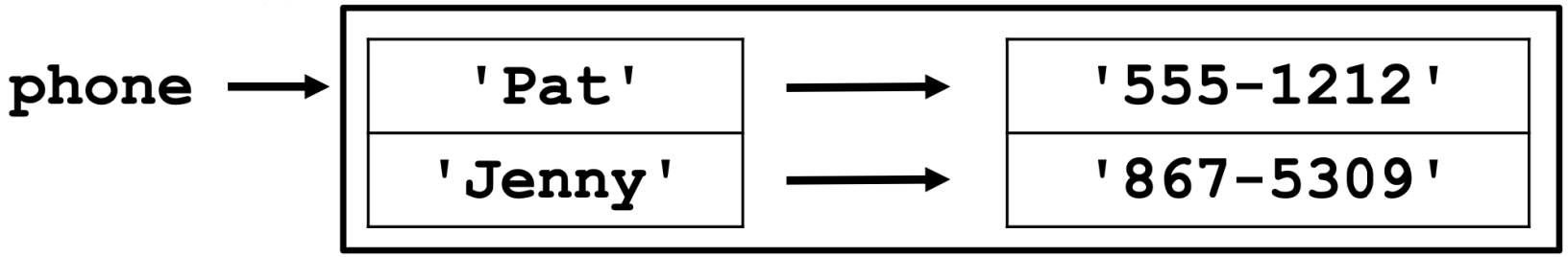
  - 类似于一系列由key索引形成的变量

  - 可以通过key查看某条记录是否存在：

```
>>> 'Juliette' in ages
True
>>> 'Santa Claus' not in ages
True
```

- Python中的字典：
  - 字典数据类型（Dict）
  - 向字典中添加记录：

```
phone = {}
phone['Pat'] = '555-1212'
phone['Jenny'] = '867-5309'
```



- Python字典key/value注意事项：
  - key一定是不可变类型（immutable）
    - 可以是int、float、string、tuple，不能是list
    - 如果需要修改key，要先删除key/value记录，再用新的key添加记录
  - value可以是可变（mutable）或不可变类型（immutable）
    - 可以是int、float、string、list、dictionary
    - value可以在原位置被修改
  - 字典本身是可变类型（mutable）
    - 可以在一个字典数据中任意添加、修改、删除记录

■ 修改Python字典

```
def have_birthday(dict, name):
    print("You're one year older, " + name + "!")
    dict[name] += 1

def main():
    ages = {'Chris': 32, 'Juliette': 22, 'Mehran': 50}
    print(ages)
    have_birthday(ages, 'Chris')
    print(ages)
    have_birthday(ages, 'Mehran')
    print(ages)
```

```
{'Chris': 32, 'Juliette': 22, 'Mehran': 50}
You're one year older, Chris!
{'Chris': 33, 'Juliette': 22, 'Mehran': 50}
You're one year older, Mehran!
{'Chris': 33, 'Juliette': 22, 'Mehran': 51}
```

Python字典的其他用法

- 安全地获取key对应的value
- *dict*.get(key, default)
  - 返回key对应的值，如果key不存在，返回default

```
>>> print(ages.get('Chris', 100))
32
>>> print(ages.get('Santa Claus', 100))
100
```

Python字典的其他用法

- 获取字典中的所有key：
- *dict*.keys()
  - 返回一个迭代器，可以用于遍历获取每一个key

```
for key in ages.keys():
    print(str(key) + " -> " + str(ages[key]))
```

```
Chris -> 32
Juliette -> 22
Mehran -> 50
```

  - 可以将所有key转换为一个list：

```
list(ages.keys()) → ['Chris', 'Juliette', 'Mehran']
```

Python字典的其他用法

- 遍历一个字典
- 可以在一个字典上使用for循环，遍历字典的每一个key

```
for key in ages:
    print(str(key) + " -> " + str(ages[key]))
```

```
Chris -> 32
Juliette -> 22
Mehran -> 50
```

Python字典的其他用法

- 遍历一个字典
- 可以在一个字典上使用for循环，遍历字典的每一个value

```
for value in ages.values():
    print(value)
```

```
32
22
50
```

- 可以将values函数的输出转换为list：

```
list(ages.values()) → [32, 22, 50]
```

Python字典的其他用法

- 遍历一个字典
- 可以在一个字典上使用for循环，遍历字典的每一个key/value对

```
for key, value in ages.items():
    print(str(key) + " -> " + str(value))
```

```
>>> Chris -> 32
    Juliette -> 22
    Mehran -> 50
```

Python字典的其他用法

- 从字典中删除记录
- *dict*.pop(key)：从字典中删除key和它绑定的value，返回value

```
>>> ages
>>> {'Chris': 32, 'Juliette': 22, 'Mehran': 50}
>>> ages.pop('Mehran')
50
>>> ages
{'Chris': 32, 'Juliette': 22}
```

- *dict*.clear()：删除所有字典中的key/value记录

```
ages.clear() → {}
```

Python字典的其他用法

- 从字典中删除记录
- `del dict[key]`：从字典中删除key和它绑定的value，没有返回值

```
>>> ages
{'Chris': 32, 'Juliette': 22, 'Mehran': 50}
>>> del ages['Mehran']
>>> ages
{'Chris': 32, 'Juliette': 22}
```

- `len(dict)`：返回字典中key/value对的个数

Python字典的可变性

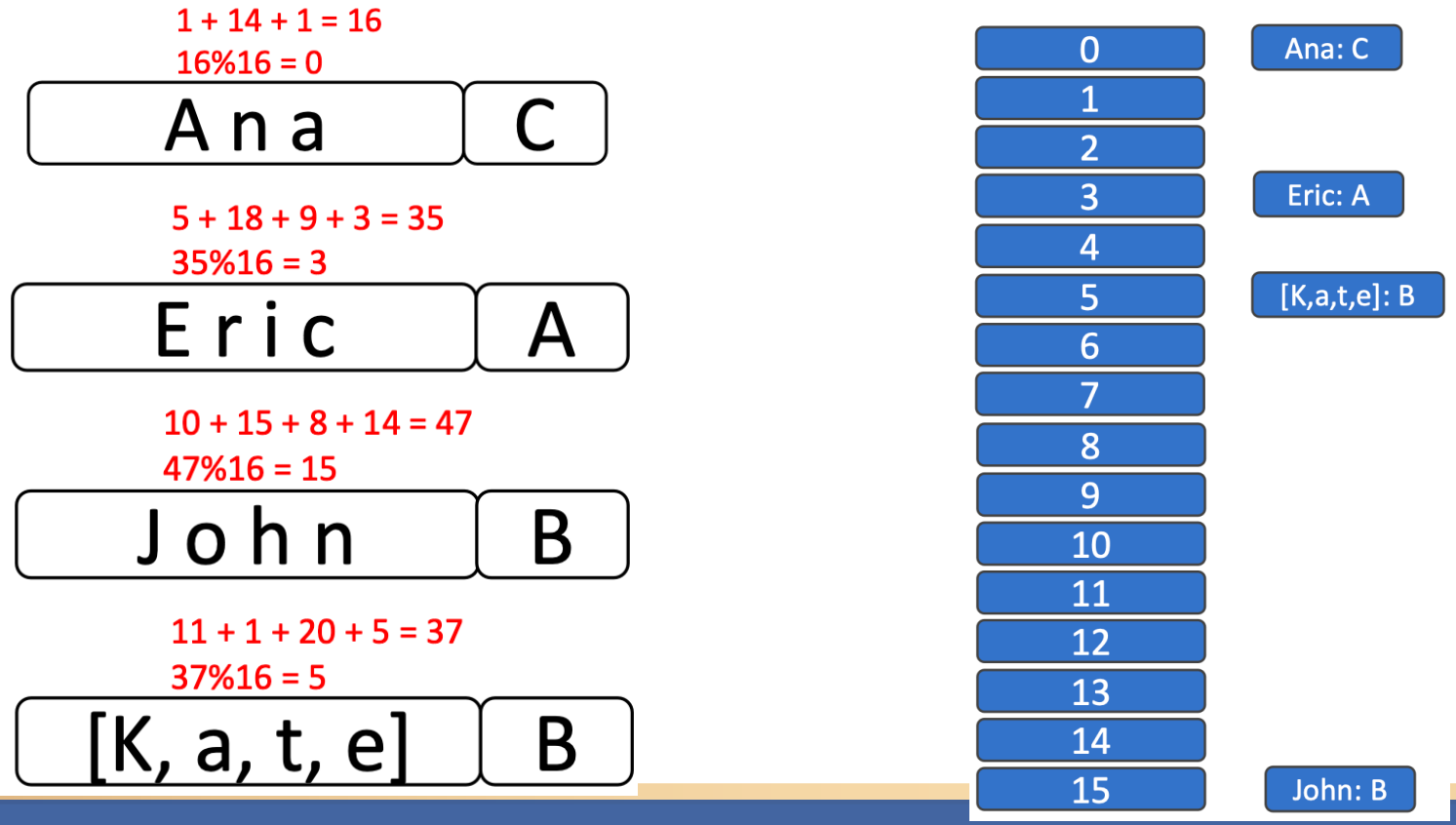
- 字典是可变数据类型（遵守别名和克隆原则）
  - 可以使用赋值操作创建别名
  - 可以使用 dict.copy() 创建克隆
- values可以是任意类型，可以被复制
- keys必须是唯一的，必须是不可变类型，要当心使用float作为key时的情况

Python字典的可变性

- 为什么字典的key一定是不可变的?
- 跟字典在内存中特殊的存储方式有关
- 存储字典时，对于每一个key，先通过一个函数转换为一个整数
- 每一个整数对应着一块内存地址的位置
- 将key绑定的value存储到对应的地址上
- 查询字典时，如果key时不可变的，可以始终获取到同一个地址
- 如果key可变，获取到的地址会发生变化，value发生变化

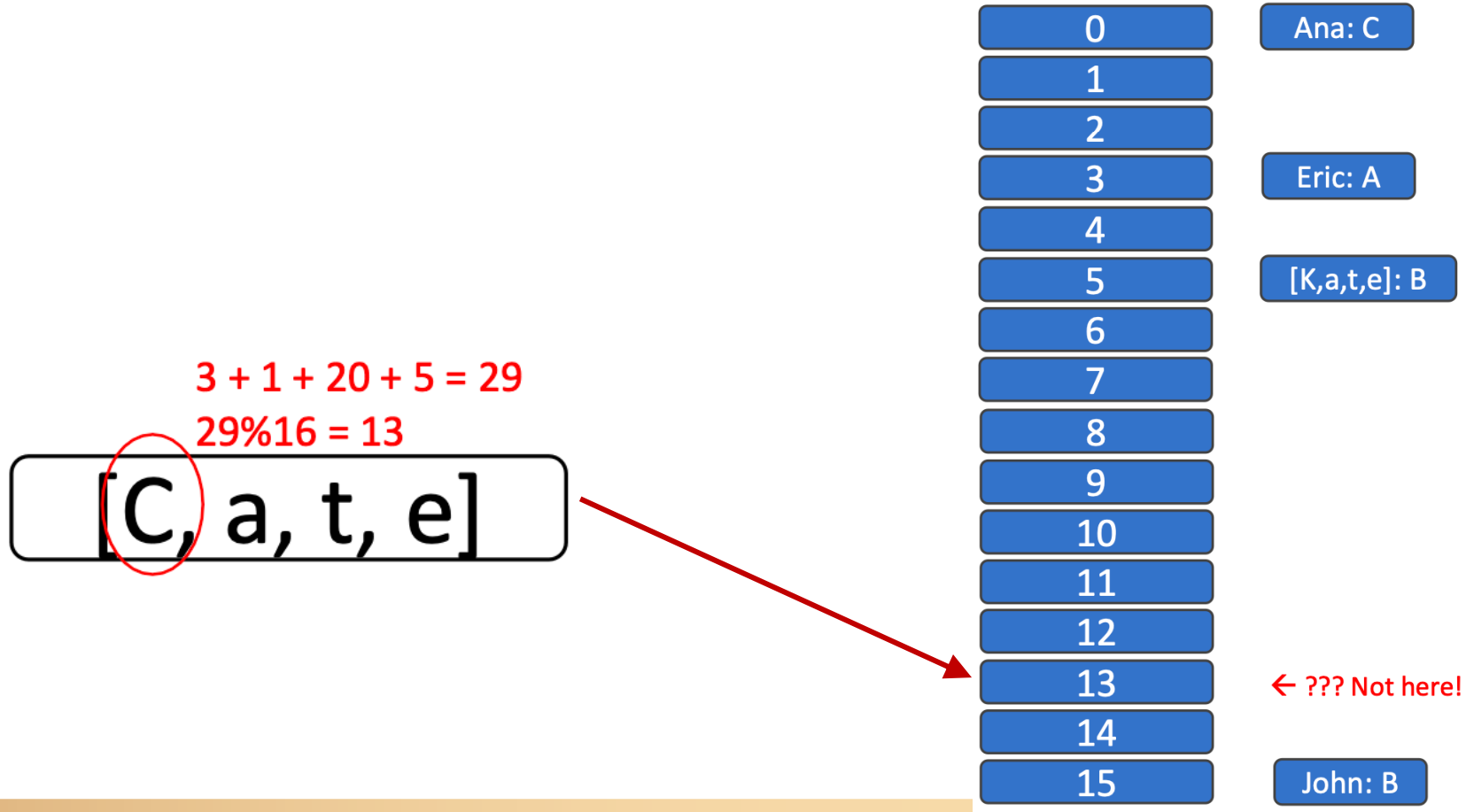
Python字典的可变性

- 将key转换为地址的函数，叫做哈希函数（hash function）
- 假设一个哈希函数为：将key中每一个字母的序号相加再取除以16的余数



■ Python字典的可变性

- 如果这时，将key为Kate的名字改为了Cate，查询她成绩时会发生什么？



Python字典的可变性

- 字典的value可以是可变或不可变类型，如dictionary、list
- 例子：学生多门课的多个成绩

```
grades = {'Ana':{'mq':[5,4,4], 'ps': [10,9,9], 'fin': 'B'},
          'Bob':{'mq':[6,7,8], 'ps': [8,9,10], 'fin': 'A'}}
```

'Ana'	'mq'	[5,4,4]
	'ps'	[10,9,9]
	'fin'	'B'
'Bob'	'mq'	[6,7,8]
	'ps'	[8,9,10]
	'fin'	'A'

字典的整体类型：str: dict

字典每个value的类型：

str: list

str: str

Python字典与列表的对比

`list`

- 由排好序的元素组成的序列
- 通过整数索引查询元素
- 索引也是按顺序的
- 元素的值可以是任意类型

`dictionary`

- 由配对好的key和value组成
- 通过key查询value的值
- 字典的key和记录没有固定顺序
- key可以是任意不可变类型
- value可以是任意类型

- Python字典应用示例：从一首歌词中找出出现次数最多的单词
  - 创建一个频次字典，类型为 `str: int`
  - 找到出现频次最多的单词，并给出次数：
    - 使用一个`list`记录次数对应的单词（可能多个单词出现一样多次）
    - 返回一个元组`(list, int)`用于表示(单词列表，最多次数)
  - 找到出现至少X次的单词
    - 作为一个参数，让用户选择 `X`
    - 返回一个元组构成的`list`，每个元组为`(list, int)`，表示单词列表和对应的出现次数

Python字典应用示例：从一首歌词中找出出现次数最多的单词

```
song = "RAH RAH AH AH AH ROM MAH RO MAH MAH"
def generate_word_dict(song):
    song_words = song.lower()
    words_list = song_words.split()
    word_dict = {}
    for w in words_list:
        if w in word_dict:
            word_dict[w] += 1
        else:
            word_dict[w] = 1
    return word_dict
```

将歌词统一转换为小写，便于统计

将歌词转换为单词列表，默认空格分割

遍历歌词中的每个单词

如果单词已经出现在统计结果中，更新次数+1

如果单词还没出现在统计结果中，添加记录

返回 str:int 类型的字典

- Python字典应用示例：从一首歌词中找出出现次数最多的单词

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}

def find_frequent_word(word_dict):
    words = []
    highest = max(word_dict.values())
    for k,v in word_dict.items():
        if v == highest:
            words.append(k)
    return (words, highest)
```

找到最大次数的值

遍历字典找出最大次数对应的单词

找出所有最大次数的单词，追加到list中

返回元组：([word1, word2,...], frequency)

Python字典应用示例：从一首歌词中找出出现次数最多的单词

```
word_dict = {'rah':2, 'ah':3, 'rom':1, 'mah':3, 'ro':1}

def occurs_often(word_dict, x):
    freq_list = []
    word_freq_tuple = find_frequent_word(word_dict)
    while word_freq_tuple[1] > x:
        word_freq_tuple = find_frequent_word(word_dict)
        freq_list.append(word_freq_tuple)
        for word in word_freq_tuple[0]:
            del(word_dict[word])
    return freq_list
```

初始化一个单词元组

只要频次还大于x就继续遍历

将大于x次数的单词存入结果中

修改字典，移除当前频次的单词记录，继续遍历，找出现第二多次数的单词

■ Python字典应用示例：从一首歌词中找出出现次数最多的单词

- 该例子的一些注意事项：

1. 将长字符串转换为单词列表可以让我们使用列表方法处理数据

```
words_list = song_words.split()
```

2. 可以利用列表天然的结构进行单词遍历

```
for w in words_list:
```

3. 可以利用字典天然的结构进行单词和频次的遍历与处理

```
for k,v in word_dict.items():
```

4. 字典的可变性可以让我们迭代式地处理字典记录

```
del(word_dict[word])
```

■ 字典相关知识回顾：

- **核心概念**
  - key/value对结构：{key: value}
  - key：唯一、不可变类型（字符串、整数等）
  - value：任意数据类型（包括列表、字典）
- **基础操作**
  - 创建：grades = {'Ana': 'A', 'Bob': 'B'}
  - 访问：grades['Ana']
  - 添加/修改：grades['Chris'] = 'A+'
  - 删除：del grades['Bob']

- 字典相关知识回顾：
  - **常用方法**
    - keys(), values(), items()
    - get(key, default)：安全访问key/value
    - pop(key)：删除并返回value

■ 字典练习1：学生成绩管理系统

- **场景需求**
  - 存储学生姓名、各科成绩、小测验分数
  - 支持添加、查询、更新、删除操作
- **代码实现**

```
# 初始化嵌套字典
students = {
    'Ana': {'math': 90, 'physics': 85, 'quizzes': [8, 9, 7]},
    'Bob': {'math': 78, 'physics': 92, 'quizzes': [6, 8, 7]}
}
```

■ 字典实战练习1：学生成绩管理系统

- **代码实现**

```
# 添加学生
students['Chris'] = {'math': 88, 'physics': 90, 'quizzes': [9, 9]}

# 查询物理成绩
print(students['Ana'].get('physics', '未录入'))

# 更新数学成绩
students['Bob']['math'] = 85

# 删除学生
del students['Chris']
```

■ 字典练习2：词频统计工具

- **场景需求**
  - 统计文本文件中每个单词的出现次数
  - 输出最高频单词及其频率
- **代码实现**

```
def word_frequency(file_path):
    # 你的代码
# 示例调用
print(word_frequency('poem.txt'))
```

■ 字典练习2：词频统计工具

- **代码实现**

```
def word_frequency(file_path):
    word_count = {}
    with open(file_path, 'r') as file:
        for line in file:
            words = line.strip().lower().split()
            for word in words:
                word_count[word] = word_count.get(word, 0) + 1
    max_freq = max(word_count.values())
    common_words = [k for k, v in word_count.items() if v == max_freq]
    return common_words, max_freq
```

■ 字典练习3：电话簿应用

- **场景需求**
  - 实现添加、查找、删除联系人功能
  - 支持输入错误处理（如重复添加）
- **代码实现**

```
phonebook = {}

def add_contact(name, number):

def find_contact(name):

def delete_contact(name):
```

■ 字典练习2：词频统计工具

- **代码实现**

```
def add_contact(name, number):
    if name in phonebook:
        print(f"{name} 已存在！")
    else:
        phonebook[name] = number

def find_contact(name):
    return phonebook.get(name, "未找到联系人")

def delete_contact(name):
    if name in phonebook:
        phonebook.pop(name)
    else:
        print(f"{name} 不存在！")
```

```
# 示例调用
add_contact('Alice', '123-4567')
print(find_contact('Alice'))
delete_contact('Bob')  # 错误处理
```

■ 字典练习4：数据聚合与统计

- **场景需求**
  - 计算每个学生的总分和平均分
  - 按科目统计全班平均分
- **代码实现**

```
students = {
    'Ana': {'math': 90, 'physics': 85},
    'Bob': {'math': 78, 'physics': 92}
}
```

■ 字典练习4：数据聚合与统计

- **代码实现**

```
# 学生总分与平均分
for name, scores in students.items():
    total = sum(scores.values())
    avg = total / len(scores)
    print(f"{name}: 总分={total}, 平均分={avg:.1f}")

# 科目平均分
math_scores = [s['math'] for s in students.values()]
physics_avg = sum(s['physics'] for s in students.values()) / len(students)
print(f"数学平均分: {sum(math_scores)/len(math_scores):.1f}")
```

■ 字典练习5：学生成绩分析系统（拓展）

- **场景需求**
  - 存储学生各科成绩，计算每个学生的总分、平均分
  - 统计各科目全班的平均分
  - 处理学生或科目不存在的情况
- **代码实现**

```
students = {
    'Ana': {'math': 90, 'physics': 85, 'chemistry': 78},
    'Bob': {'math': 78, 'physics': 92, 'chemistry': 88}
}
```

■ 字典练习5：学生成绩分析系统（拓展）

- **代码实现**

```
def calculate_student_stats():
    for name, scores in students.items():
        try:
            total = sum(scores.values())
            avg = total / len(scores)
            print(f"{name}: 总分={total}, 平均分={avg:.1f}")
        except ZeroDivisionError:
            print(f"{name} 无成绩记录！")
```

■ 字典练习5：学生成绩分析系统（拓展）

- **代码实现**

```
def calculate_subject_avg(subject):
    scores = []
    for student in students.values():
        try:
            scores.append(student[subject])
        except KeyError:
            print(f"警告: {subject} 科目不存在于部分学生记录中")
    if scores:
        avg = sum(scores) / len(scores)
        print(f"{subject} 平均分: {avg:.1f}")
    else:
        print(f"无有效 {subject} 成绩记录")
```

■ 字典练习5：学生成绩分析系统（拓展）

- **代码实现**

```
# 调用示例
calculate_student_stats()
calculate_subject_avg('math')
calculate_subject_avg('biology')   # 触发异常处理
```

■ 字典练习6：用户登录验证系统

- **场景需求**
  - 从JSON文件加载用户数据（用户名、密码）
  - 验证用户登录，支持注册新用户
  - 用户数据持久化到JSON文件
- **代码实现**

```
import json
def load_users(file_path):
def save_users(users, file_path):
def login(users):
def register(users):
```

■ 字典练习6：用户登录验证系统

- **代码实现**

```
def load_users(file_path):
    try:
        with open(file_path, 'r') as file:
            return json.load(file)
    except FileNotFoundError:
        return {}

def save_users(users, file_path):
    with open(file_path, 'w') as file:
        json.dump(users, file)
```

■ 字典练习6：用户登录验证系统

- **代码实现**

```
def login(users):
    username = input("用户名：")
    password = input("密码：")
    if users.get(username) == password:
        print("登录成功！")
    else:
        print("用户名或密码错误！")
```

■ 字典练习6：用户登录验证系统

- **代码实现**

```
def register(users):
    username = input("新用户名: ")
    if username in users:
        print("用户名已存在！")
        return
    password = input("密码: ")
    users[username] = password
    print("注册成功！")
```

■ 字典练习6：用户登录验证系统

- **代码实现**

```
# 主程序
users_file = "users.json"
users = load_users(users_file)
action = input("登录(L) / 注册(R): ").upper()
if action == 'L':
    login(users)
elif action == 'R':
    register(users)
    save_users(users, users_file)
else:
    print("无效操作！")
```

■ 字典练习7：商品库存管理系统

- **场景需求**
  - 管理商品ID、名称、价格、库存
  - 支持添加商品、查询库存、更新库存
  - 记录操作日志（时间、操作类型）
- **代码实现**

```
import datetime
products = {
    'P001': {'name': '键盘', 'price': 299, 'stock': 50},
    'P002': {'name': '鼠标', 'price': 150, 'stock': 30}
}
logs = []
```

■ 字典练习7：商品库存管理系统

- **代码实现**

```
def add_product(product_id, name, price, stock):
    if product_id in products:
        print("商品ID已存在！")
        return
    products[product_id] = {
        'name': name, 'price': price, 'stock': stock}
    logs.append(
        (datetime.datetime.now(), f"添加商品 {product_id}"))
```

■ 字典练习7：商品库存管理系统

- **代码实现**

```
def update_stock(product_id, quantity):
    try:
        products[product_id]['stock'] += quantity
        logs.append(
            (datetime.datetime.now(), f"更新库存 {product_id}")
    except KeyError:
        print("商品不存在！")
```

■ 字典练习7：商品库存管理系统

- **代码实现**

```
def show_logs():
    for log in logs:
        print(f"[{log[0]}] {log[1]}")

# 示例调用
add_product('P003', '耳机', 199, 20)
update_stock('P001', -10)  # 卖出10个键盘
show_logs()
```

■ 字典练习8：电影推荐系统

- **场景需求**
  - 从JSON文件读取用户电影评分数据
  - 根据当前用户喜好推荐相似用户的高评分电影
  - 处理文件格式错误
- **代码实现**

```
import json

def load_ratings(file_path):

def recommend_movies(user_ratings, all_ratings):
```

■ 字典练习8：电影推荐系统

- **代码实现**

```
def load_ratings(file_path):
    try:
        with open(file_path, 'r') as file:
            return json.load(file)  # 字典（用户：评分）
    except json.JSONDecodeError:
        print("文件格式错误！")
        return {}
    except FileNotFoundError:
        print("文件不存在！")
        return {}
```

■ 字典练习8：电影推荐系统

- **代码实现**

```
def recommend_movies(user_ratings, all_ratings):
    similar_users = []
    for user, ratings in all_ratings.items():
        similarity = sum([
            1 for movie in user_ratings
            if movie in ratings and ratings[movie] >= 4])
        if similarity >= 2:
            similar_users.append(user)
```

■ 字典练习8：电影推荐系统

- **代码实现**

```
def recommend_movies(user_ratings, all_ratings):
    ……
    recommendations = {}
    for user in similar_users:
        for movie, score in all_ratings[user].items():
            if score >= 4 and movie not in user_ratings:
                recommendations[movie] = recommendations.get(movie, 0) + 1
    return sorted(recommendations.items(), key=lambda x: -x[1])
```

■ 字典练习8：电影推荐系统

- **代码实现**

```
# 示例数据
ratings = load_ratings("ratings.json")
current_user = {'MovieA': 5, 'MovieB': 4}
print("推荐电影:", recommend_movies(current_user, ratings))
```

什么是递归?

- 递归是一种通过**函数调用自身**的方式解决问题的方法
- 用来将一个复杂问题分解为结构相似但规模更小的问题
- 核心包括:
  - **基本情况（base case）**：直接返回结果的终止条件
  - **递归步骤（recursive step）**：把问题转换为更小的同类问题
- **递归的意义**：递归让代码更简洁，逻辑更贴近人类对问题的思考方式，特别适合处理具有层级结构或重复子结构的问题

■ 递归的应用场景

- ➢ 数学运算：阶乘、幂、斐波那契数列等
- ➢ 数据结构遍历：树、图、链表等
- ➢ 分治算法：快速排序、归并排序等
- ➢ 实际问题：如文件夹遍历、图像处理、AI搜索等

- 递归在这些场景中，能帮助我们简洁地描述问题的结构

迭代式地进行乘法计算

```
def mult_iter(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result
```

- 通过循环不断累加a，共循环b次，最终得到结果
- 是一种逐步推进的“状态更新”方法

递归式地进行乘法计算

- 想象我们要计算 5 × 4：
- 5 + 5 + 5 + 5
- 也可以看作：5 + (5 × 3)
- 继续分解：5 + (5 + (5 × 2)) → 直至最后：5 + 5 + 5 + 5
- **递归思想**：通过问题的重复结构，自然过渡到递归方式

■ 递归乘法代码

```
def mult_recur(a, b):
    if b == 1:
        return a
    else:
        return a + mult_recur(a, b-1)
```

递归过程解释:

- 当 `b = 1`, 返回 `a` (基本情况)
- 否则返回 `a + mult_recur(a, b-1)`

■ 递归乘法的执行过程

```
mult_recur(5, 4)
= 5 + mult_recur(5, 3)
= 5 + (5 + mult_recur(5, 2))
= 5 + (5 + (5 + mult_recur(5, 1)))
= 5 + 5 + 5 + 5 = 20
```

```
def mult_recur(a, b):
    if b == 1:
        return a
    else:
        return a + mult_recur(a, b-1)
```

■ 递归乘法的执行过程

调用栈展开:

```
mult_recur(5,4)
→ mult_recur(5,3)
   → mult_recur(5,2)
      → mult_recur(5,1) → return 5
```

每次调用都有独立的变量空间

返回过程:

```
5 + 5 → 10
10 + 5 → 15
15 + 5 → 20
```

返回值向上传递

■ 递归调用的通用写法

```
def recursive_fn(...):
    if base_case:
        return result
    else:
        return recursive_fn(smaller_problem)
```

理解要点:

- 找到终止条件
- 将问题缩小，逐步接近终止条件

过程：问题规模逐步减小 → 到达终止点 → 回溯返回

■ 递归计算幂

```
def power_recur(n, p):
    if p == 0:
        return 1
    else:
        return n * power_recur(n, p-
1)
```

```
power(2,3)
→ 2 * power(2,2)
→ 2 * 2 * power(2,1)
→ 2 * 2 * 2 * power(2,0) = 1
→ 回溯结果 = 8
```

示例：power(2, 3) → 2 * power(2, 2) → 2 * 2 * power(2,1)...

- 每次递归把指数p减小，直到为0

■ 递归计算阶乘

$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

示例：`fact(4) = 4 × 3 × 2 × 1 = 24`

```
fact(4)
→ 4 * fact(3)
→ 4 * 3 * fact(2)
→ 4 * 3 * 2 * fact(1) = 1
```

→ 回溯：`1 → 2 → 6 → 24`

■ 递归的调用机制

- 每个函数调用都在内存中创建新的执行环境（栈帧）
- 当调用层层深入 → 形成函数调用栈
- 当遇到基本情况 → 回溯逐层返回值
- 函数嵌套像俄罗斯套娃，最里面先返回

■ 递归与迭代对比

	递归	迭代
编写简洁	✅	❌
可读性强	✅	✅
内存占用	❌（栈帧多）	✅
性能	较慢	较快
容易调试	❌	✅

建议：如能轻松使用迭代，优先考虑性能更佳的迭代方式

■ 何时使用递归?

- 适合递归的场景:
  - 问题本身具有递归结构
  - 使用迭代写法复杂、冗长
- 不适合递归的场景:
  - 输入规模非常大（风险：栈溢出）
  - 能用循环更直接地解决

■ 递归练习：斐波那契数列

- 斐波那契定义：F(n) = F(n-1) + F(n-2), F(0)=0, F(1)=1

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

- 问题：fib(4) → fib(3)+fib(2) → 分支爆炸
- 此版本存在大量重复计算，效率低（如 fib(2)计算多次）

■ 递归练习：斐波那契数列

- 优化版递归：记忆化斐波那契

```
memo = {}
def fib_mem(n):
    if n in memo:
        return memo[n]
    if n <= 1:
        memo[n] = n
    else:
        memo[n] = fib_mem(n-1) + fib_mem(n-2)
    return memo[n]
```

- 使用字典缓存已有结果，避免重复计算，提高效率

■ 递归练习：字符串长度的递归实现

```
def str_len(s):
    if s == '':
        return 0
    else:
        return 1 + str_len(s[1:])
```

- 每次去掉第一个字符，直到字符串为空
- `str_len("cat")` → `1 + str_len("at")` → `...` → 得到3

■ 递归的陷阱与调试技巧

- 常见错误：
  - 忘记基本情况 → 无限递归
  - 基本情况写错 → 错误结果
- 调试建议：
  - 手动画出调用过程
  - 使用 print 或其他Python 工具将递归过程可视化出来

■ 递归练习1：递归计算列表中数字的和

- 编写一个函数 `list_sum(lst)`，递归计算列表中所有数字的和
- 示例：`list_sum([1, 2, 3, 4])` → `10`
- 提示：
  - 基本情况：列表为空时返回`0`
  - 否则返回第一个元素 `+` 剩余元素的和

■ 递归练习1：递归计算列表中数字的和

```
def list_sum(lst):
    # 基本情况：列表为空
    if not lst:
        return 0
    # 递归情况：返回第一个元素 + 剩余元素的和
    return lst[0] + list_sum(lst[1:])
```

■ 递归练习2：支持嵌套列表的数字求和

- 实现函数 `nested_sum(lst)`，支持列表中包含嵌套子列表的情形：
- `nested_sum([1, [2, [3, 4]], 5])` → `15`
- 提示：使用递归判断元素是否为列表，再递归展开
- 思考：这个问题用递归是否比用迭代更简洁？

■ 递归练习2：支持嵌套列表的数字求和

```
def nested_sum(lst):
    total = 0
    for item in lst:
        if isinstance(item, list):
            total += nested_sum(item)  # 如果是子列表，则递归调用
        else:
            total += item              # 否则直接加
    return total
```

Reading and QA Time

See you next week !