字典操作示例

用 setdefault 方法处理键不存在，能减少键查询，更高效

```
>>> dic = {}
>>> dic.update(math=['51800'])
>>> dic
{'math': ['51800']}
>>> dic.setdefault('bio', []).append('61500')
>>> dic
{'math': ['51800'], 'bio': ['61500']}
>>> dic.setdefault('math', []).append('51804')
>>> dic
{'math': ['51800', '51804'], 'bio': ['61500']}
```

```
# 效果等价于
if 'math' not in dic: dic['math'] = []
dic['math'].append('51804')
```

字典观察对象 (dictionary view object)

- 一种依附于具体字典的、具有迭代器性质的动态观察对象
  - 如被依附的字典改变，从观察对象也将 (实时) 看到有关改变
  - 以下三个操作可生成字典 dic 的观察对象：

dic.keys()	得到 dic 的所有关键字，顺序由元素插入顺序确定
dic.values()	得到 dic 的所有值，顺序由元素插入顺序确定
dic.items()	得到 dic 的所有关联对，顺序由元素插入顺序确定，关联对用形式为 (k, v) 的二元组表示

  - 可以利用字典观察对象在字典的关键字或值上迭代
  - 但是，在迭代字典的过程中，不要同时修改该字典！
    - 在关键字上进行迭代 (包括利用内置函数 iter)，可以修改已有关键字的关联值，但是不能增删字典里的元素

字典观察对象示例

```
>>> pku = {'math': 114, 'phys': 247, 'chem': 306}
>>> keys = pku.keys()
>>> values = pku.values()
>>> items = pku.items()
>>>
>>> list(keys)
['math', 'phys', 'chem']
>>> list(values)
[114, 247, 306]
>>> list(items)
[('math', 114), ('phys', 247), ('chem', 306)]
>>>
>>> pku['bio'] = 361
>>> list(keys)
['math', 'phys', 'chem', 'bio']
>>> list(values)
[114, 247, 306, 361]
>>> list(items)
[('math', 114), ('phys', 247), ('chem', 306), ('bio', 361)]
```

在字典上迭代

- 字典不是序列，但在程序中也经常需要顺序处理一个字典里的所有元素
  - 几个典型的字典迭代模式：

```
for k in dic.keys():                    # 按默认的关键字顺序处理
    ... k ... dic[k] ...

for k in sorted(dic.keys()):            # 按关键字排序后的顺序处理
    ... k ... dic[k] ...

for k, v in dic.items():                # 按元素的默认顺序处理
    ... k ... v ...
```

Note：第一个模式可简化为 (Python 自动地调用 keys())：

```
for k in dic:
    ... k ... dic[k] ...
```

字典与函数定义中的双星号形参

- 在函数形参列表的最后，可以有一个前面带两个星号的形参
  - 函数定义里，双星号形参的值是一个 (自动建立的) 字典对象，从各关键字映射到相应的实参值
  - 函数调用时，该形参约束到所有未得到匹配的关键字实参形成的字典，默认情况下约束到一个空字典
- 例：定义函数，接受并输出任意多个关键字实参

```
PKU:
math: 114
phys: 247
chem: 336
bio:  361
```

```
def print_data(x, **dic):
    print(x)
    for k, v in dic.items():
        print("{:6}{}".format(str(k) + ":", v))

print_data("PKU:", math=114, phys=247, chem=336, bio=361)
```

字典拆分实参和关键字实参

- Python 允许通过拆分字典的方式为函数提供一组关键字实参
  - 形式：函数调用式中，在描述字典的实参表达式 (例如，值为字典的变量等) 前加两个星号
  - 语义：实参字典中的键 (key) 被看作是函数调用中关键字实参的关键字，与之关联的值作为实参值
  - 要求：实参字典的键与函数形参的名字相互匹配，或者在函数定义的形参表里有双星号形参

```
def print_them(math, phys, chem, **others):
    form1 = "{:6}{}"
    print(form1.format("math:", math))
    print(form1.format("phys:", phys))
    print(form1.format("chem:", chem))
    print("others:", others)

uni = {"chem": 336, "math": 114,  "phys": 247, "bio": 234}
print_them(**uni)
```

```
math:  114
phys:  247
chem:  336
others: {'bio': 234}
```

字典的简单应用实例 1

- 在 **Fib** 序列的记忆型递归计算中，采用字典作为数据缓存
  - 以整数作为字典关键字，实现方式和以表作为缓存类似 **(**实际上，字典的关键字可以更具一般性**)**
  - 基于关键字的元素检索
    - **dic[k]**：**k** 不存在时将报错，应先检查 **k in dic** 是否成立
    - **dic.get(k[, default])**：**k** 不存在时返回 **None** 或 **default (**不会报错**)**

字典的简单应用实例 1

- 在 **Fib** 序列的记忆型递归计算中，采用字典作为数据缓存
  - 以整数作为字典关键字，实现方式和以表作为缓存类似 (实际上，字典的关键字可以更具一般性)
  - 基于关键字的元素检索

```
def fib(n):
    fibs = {0:0, 1:1} # 初始字典，前两项是 fibs[0]=0 和 fibs[1]=1

    def fib0(k):
        if k in fibs:
            return fibs[k]
        fibs[k] = fib0(k-2) + fib0(k-1)
        return fibs[k]

    return fib0(n)
```

字典的简单应用实例 2

- 问题：统计文本中各英文字母出现的频率
  - 每个字母对应一个计数器，字典是合理选择，可按需动态地加入字母/计数器对
  - 以计数器字典作为统计函数的参数：可以把统计结果反映到外部字典中 (字典是可变对象)，也适合反复处理多个文本，累积统计信息

字典的简单应用实例 2

```
#### 用字典实现一组计数器
#### 统计参数字符串（文本）里各英文字母出现的频率
def char_count(s, dic):
    for c in s:
        if not c.isalpha():
            continue
        c = c.lower()
        dic[c] = dic.get(c, 0) + 1

d1 = {}
char_count("Python is an programming language.", d1)
print(d1)

text = """Return true if all characters in the string are alphabetic
and there is at least one character, false otherwise. Alphabetic
characters are those characters defined in the Unicode character
database as “Letter”, i.e., those with general category property
being one of “Lm”, “Lt”, “Lu”, “Ll”, or “Lo”. Note that
this is different from the “Alphabetic” property defined in the
Unicode Standard."""
char_count(text, d1)
print(d1)
```

集合

- 数学里的集合是一批元素的汇集
  - 元素无所谓的“顺序”，只支持元素判断 (是否属于集合？)
  - 有一组集合运算：并集，交集，补集等
- 程序里也可能需要具有集合性质的对象，Python 为此提供了内置集合类型，其基本集合类型是 **set**
- **set** 是一种可变组合类型，基本情况：
  - 集合中的元素只能是可哈希的 **(hashable)**
    - 内置的数值类型、字符串、**bool** 对象，及它们的元组都满足上述要求
  - 一个集合对象里的元素唯一，且无重复，元素之间无顺序
- **frozenset** (冻结集合) 是不变集合类型，支持所有不改变集合对象的操作

集合的构造

- 集合的基本构造方式：
  - 建立集合时，将自动消除重复元素
  - {表达式, ...} 描述一个集合，以 表达式的值 作为集合的元素
  - 用 set(...) 可以从序列或可迭代对象生成相应的集合
  - 空集合只能用 set() 生成
  - 不变集合对象用 frozenset(…) 生成
- *Note*：集合与字典都用 {...} 描述
  - {…} 里写 “表达式：表达式”，是在生成字典
  - {…} 里写其他形式的表达式，是在生成集合
  - 空集只能用 set() 生成，空字典可以用 {} 或 dict()

集合的构造和转换

- 字典/集合推导式：{ 生成器表达式 }
  - 生成表达式为 表达式：表达式 时，生成字典
  - 生成表达式为简单形式的 表达式 时，生成集合
- **forzenset(生成器表达式)** 生成不变集合

```
>>> frozenset(n**2 for n in range(-20, 20))  # 自动消除重复元素
```

- **set 和其他类型之间的转换**
  - 与元素满足要求的 **tuple** 可相互转换，转到 **set** 将消除重复元素，未必保持原顺序，**set** 转 **tuple** 顺序由内部确定
  - 元素满足 **set** 要求的 **list** 对象可以转换到 **set**，**set** 对象总能转到 **list**，但顺序由内部确定
  - **set("abbcdf")** 得到集合 **{'b', 'c', 'a', 'f', 'd'}**
  - **set(dic)** 得到字典 **dic** 的关键字集合

```
>>> x = -3
>>> {abs(x), x, x + 5, x**3}
{-27, 2, 3, -3}
>>>
>>> {"math", "phys", "chem"}
{'chem', 'phys', 'math'}
>>>
>>> set() is set()     # 两次生成的空集合是不同的对象
False
>>>
>>> frozenset() is frozenset()  # 不变空集只有唯一一个
True
>>>
>>> {n**2 for n in range(-10, 10)}  # 集合元素唯一
{64, 1, 0, 100, 36, 4, 9, 16, 81, 49, 25}
>>>
>>> frozenset(n**2 for n in range(-10, 10))  # 注意 frozenset 的输出形式
frozenset({64, 1, 0, 100, 36, 4, 9, 16, 81, 49, 25})
>>>
>>> {1, 1.0}  # 消除重复元素时用 == 判断元素相等，值 1 和 1.0 相等
{1}
>>>
>>> {True, 1} # True 和 1 等值
{True}
>>>
>>> {0, False} # False 和 0 等值
{0}
```

集合的操作

- 集合不是序列类型，但支持一些序列操作：
  - **x in s**, **x not in s** 判断 x 是否在集合 s 里出现
  - **len(s)** 得到集合 s 的元素个数
- 集合可以作为 for 语句的数据源 (设 s 是集合)

```
for x in s :
    ... x ...
```

```
for x in sorted(s) :
    ... x ...
```

  - 得到集合元素的顺序由内部确定；如果要按确定的顺序循环，可以通过内置函数 sorted 得到集合元素的排序表
- **max(s)**, **min(s)** 可求得集合中最大的或最小的元素
  - 这里要求集合的元素可以用关系运算符比较大小

集合的比较

- 在比较集合时，并不区分 set, frozenset
- 集合之间可以用 ==，!= 运算符判断相等和不等
  - 当且仅当两个集合的元素完全相同时，用 == 比较得到 True，否则 False (符合数学定义)

```
>>> {1, 2, 3} == frozenset((1, 2, 3))      # 结果为 True
```

s <= s1 s.issubset(s1)	当且仅当 s 为 s1 的子集时返回 True
s < s1	当且仅当在 s 为 s1 的真子集时得到 True；真子集说明 s1 至少包含一个不属于 s 的元素
s >= s1 s.issuperset(s1)	判断 s 是否为 s1 的超集
s > s1	判断 s 是否为 s1 的真超集
s.isdisjoint(s1)	判断两个集合是否不相交，两个集合不相交就是它们没有公共元素

集合的运算

- 一组与数学中集合运算对应的运算 (生成新集合)：
  - 对非运算符形式的运算，实参可以是集合或任意可迭代对象
  - 运算符形式的运算必须作用于集合

s1.union(s2, ...) **s1 | s2 | ...**	生成一个新集合，它是这些参数集合的并集。一个元素属于这个并集，当且仅当它属于参加运算的某个参数集合
s1.intersection(s2, ...) **s1 & s2 & ...**	生成一个新集合，它是这些参数集合的交集。一个元素属于这个交集，当且仅当它属于参加运算的每一个参数集合
s1.difference(s2, ...) **s1 – s2 – ...**	生成第一个集合减去后面集合的差集
s1.symmetric_difference(s2) **s1 ^ s2**	生成 **s1** 和 **s2** 的对称差集。该集合包含所有属于 **s1** 但不属于 **s2** 的元素，再加上属于 **s2** 但不属于 **s1** 的元素
s.copy()	生成 **s** 的一个浅拷贝

修改集合操作和集合更新操作 (只用于 set 对象)

s.add(x)	将元素 x 加入集合 s
s.remove(x)	从 s 里删除元素 x，在 s 没有 x 时报错
s.discard(x)	如果 s 里有 x 就丢弃它，没有 x 时什么也不做
s.pop()	从 s 里删除某个 (任意的) 元素并返回它，具体元素由集合内部确定。如果操作时 s 为空则报错
s.clear()	清除 s 里的所有元素

s1.update(s2, ...) s1 |= s2 | ...	修改 s1，使之包含属于其它集合的元素
s1.intersection_update(s2, ...) s1 &= s2 & ...	修改 s1，使之只包含同属于所有集合的公共元素
s1.difference_update(s2, ...) s1 –= s2 | ...	修改 s1，从中去除属于其它集合的元素
s1.symmetric_difference_update(s2) s1 ^= s2	修改 s1，使之只包含仅属于 s1 或 s2 之一但不同时属于两者的元素

集合操作的简单说明

- 一些集合和集合的比较 (除相等和不等)、运算和操作都有两个版本：运算符版本、带点号的方法调用版本
  - 两者得到的结果 (或产生的效果) 相同
  - 运算符版本只能用于集合 (set 或 frozenset) 对象，而非运算符方式的参数可以是任何合适的可迭代对象
- 如果两个参与运算的集合分别为 set 和 frozenset，结果的类型与第一个操作对象的类型相同
  - 显然，修改集合操作不能作用于 frozenset
  - 但修改集合操作的参数集合可以是 frozenset
- 注意： set 与 list 的差异
  - list 对元素没有任何限制，set 元素只能是不变对象
  - list 元素有序且允许重复，set 元素无序且不重复

字典和集合的简单应用实例：多项式表示和计算

- 多项式的字典表示：只保存非 0 系数来更紧凑地表示多项式
  - 用系数非 0 项的次数作为字典关键字，项系数则为对应值
  - 例如，$1 - 3x + 2x^{10000}$ 可用 **{0: 1, 1: -3, 10000: 2}** 表示
- 多项式的算术运算

```
#### 数乘
def poly_num_times(a, p1):
    res = {}
    for d in p1:
        res[d] = p1[d] * a
    return res

poly1 = {0: 1, 1: -3, 10000: 2}
poly2 = poly_num_times(3, poly1)
```

字典和集合的简单应用实例：多项式表示和计算

- 多项式的字典表示：只保存非 0 系数来更紧凑地表示多项式
  - 用系数非 0 项的次数作为字典关键字，项系数则为对应值
  - 例如，$1 - 3x + 2x^{10000}$ 可用 **{0: 1, 1: -3, 10000: 2}** 表示

```
#### 多项式加法
def poly_plus(p1, p2):
    res = {}
    degrees = set(p1) | set(p2)
    for d in degrees:
        coeff = p1.get(d, 0) + p2.get(d, 0)
        if coeff != 0:
            res[d] = coeff
    return res

poly1 = {0: 1, 1: -3, 10000: 2}
poly2 = {1: 3, 2: -2, 1000: 5, 10000: 6}
poly3 = poly_plus(poly1, poly2)
```

一个实例：括号匹配问题

- 在很多正文里可包含多种不同类型的括号，如 (), [], {}, ‘’, “” 等等；一系列的括号存在是否正确配对的问题：
  - 每种括号都包括相互对应的 开/左括号 和 闭/右括号；括起来的片段还可能 (多重) 嵌套

  例如：[]()、[()]、{()}[] 属于各种括号正确配对的情况；([)]、{[)}、{[(、[()]{ 属于错误配对的情况

- 配对原则：在扫描正文过程中
  - 所遇到的闭括号，应匹配此前最近遇到的、尚未获得匹配的开括号
  - 由于存在多种括号、各种括号可多次出现且可能嵌套，为检查配对情况，需要保存扫描中所遇到的开括号
    - 配对时，需逐对检查：当前闭括号应与前面最近的、尚未有匹配的开括号匹配，下一个闭括号应与更前面次近的开括号匹配
    - 已配对的开括号应该删除，为后面的匹配做准备

一种基本数据结构 — 栈 (Stack)

- 注意：在扫描并配对的过程中，后遇到并保存的开括号，将先被匹配且先删除 —— 按出现的 (时间) 顺序，后进先出，适合用栈实现
- 栈 (stack，堆栈)：使用最广泛的基本数据结构之一
  - 是一种保存数据元素的容器，可以将元素存入其中 (压栈)，或者从中取出元素使用 (查看，或弹出)
  - 主要用于在计算过程中临时性地保存元素 (计算过程中发现或产生的中间数据，需要在之后使用 —— 缓冲存储/缓存)
  - 基本特点：保证缓存元素后进先出 (Last In First Out, LIFO)
    - 在任何时候，下次访问或删除的栈元素都默认地唯一确定
    - 只有新元素的压入或弹出操作会改变下次默认访问的元素
  - 支持几个简单操作
    - 基本操作：放入元素、取得/弹出元素
    - 辅助操作：创建空栈，检查栈深度，判断栈空/满等

栈的实现

- 栈的 LIFO 性质属于逻辑层面，实际实现时可采用任何满足要求的技术
  - 一种简单技术是利用元素的排列顺序表示缓存元素的时间顺序 —— 可用线性表作为栈的具体实现
  - 例如：用表的后端插入实现元素的压栈，栈里最新的 (下次访问或删除的) 元素总是最后一个元素 (即栈顶元素)
  - ➔ 栈可以实现为只在一端 (栈顶) 插入和删除的表
- Python 的 list 及其操作已经具备栈的功能，可作为栈使用
  - 空栈对应于空表
  - 基于 list 的栈不会满
  - st.append(x) 实现压栈 (O(1) 复杂度)
  - st.pop() 实现栈顶元素的弹出 (O(1) 复杂度)
- (完全也可以自定义栈类，略)

栈的应用 (1)

- 栈是算法和程序里最常用的辅助结构，基本用途基于两个方面：
  - 用栈可以很方便地保存和取用信息，因此常作为算法或程序里的辅助存储结构，保存临时信息，供之后的操作使用
  - 利用后进先出的特点，可以得到特定的存储和取用顺序
- 最简单应用：得到有序元素组的反序序列 (代码略)
  - 如允许入栈和出栈操作任意交错，可得到有规律的其它排列
- 利用栈解决括号匹配问题 (演示代码)
  - 顺序检查正文 (一个字符串) 里的一个个字符 (可设不含无关字符，如有也可直接跳过)
  - 遇到开括号时，将其压入栈
  - 遇到闭括号时，弹出栈顶元素 (如栈非空) 与之匹配：如匹配，则继续；否则，匹配检查失败

栈的应用 (2)：算术表达式的计算

- 算术表达式的常见表示形式：
  - 中缀形式：**(3 - 5) * (6 + 17 * 4) / 3**
    - 最常见、最习惯的形式；其实并不统一 (一元运算符采用前缀形式)，且需引入括号、运算符的优先级与结合性
  - 前缀形式 (波兰表达式)：**/ * - 3 5 + 6 * 17 4 3**
    - 运算符总是写在其运算对象之前
  - 后缀形式 (逆波兰表达式)：**3 5 - 6 17 4 * + * 3 /**
    - 运算符总是写在其运算对象之后
- 对于前缀表达式和后缀表达式，不需要引入括号，也不需要规定优先级和结合性，已足以描述任意复杂的计算顺序
  - 每个运算符的运算对象，即为其后面 (前面) 出现的一个 (两个) 完整表达式
  - 相对而言，后缀表达式最适合计算机处理

后缀表达式的求值

- 假定：所处理的后缀算术表达式
  - 其中的运算对象是浮点数形式表示的数
  - 运算符只含 +、-、*、/ 二元运算符
- 分析计算的过程：设法得到下一个运算对象或者运算符
  - 如遇运算对象，则需要记录，以备之后使用
  - 如遇到运算符，则需要根据其元数取得前面最近遇到的几个运算对象或已做运算得到的结果，执行计算并记录结果
- 问题：如何记录信息？
  - 需要记录的是已经取得、但还不能立即使用的中间结果，即缓存
  - 遇到运算符时，要使用的是此前最后记录的一些结果

  ➔应该用栈作为缓存结构

基于栈求值后缀表达式

- 计算的基本框架

```
# st 是一个栈
while 还有输入:
    x = …                          # x 为下一个运算对象或运算符
    if not is_operator(x):         # x 是运算对象，则转换后压栈
        st.push(float(x))
    else:                          # 否则，x 即是二元运算符
        right = st.pop()           # 弹出右运算对象
        left = st.pop()            # 弹出左运算对象
        … …                        # 根据运算符计算
        … …                        # 计算结果压入栈
```

(具体实现见演示代码)

函数参数总结：形参

- Python 的函数定义有丰富的参数形式，参数列表中可以有：
  - 任意多个常规形参：最前面可以有只用参数名描述的常规形参，之后可以有任意多个带默认值的常规形参
  - 至多一个单星号形参
  - 至多一个双星号形参
- 说明：
  - 为了意义清晰，Python 规定一旦形参表里的某个常规形参带了默认值，位于其后直至单星号形参的所有常规形参都必须带默认值
  - 单星号形参前后都可有常规形参 (可带/可不带默认值)
  - 带双星号形参只能出现在形参列表的最后，否则是语法错

函数参数总结：形参

- Python 的函数定义有丰富的参数形式，参数列表中可以有：
  - 任意多个常规形参：最前面可以有只用参数名描述的常规形参，之后可以有任意多个带默认值的常规形参

```
>>> def f1(a, b=1, *arg1, c=[], **arg2):
        pass

>>> def f2(a, b=1, c, *args, d):
        pass
SyntaxError: non-default argument follows default argument
>>>
>>> def f3(a, b=0, c=1, *args, d):
        pass

>>> def f4(a, b=1, *arg1, c=[], **arg2, d={}):
        pass
SyntaxError: invalid syntax
```

函数参数总结：实参

- 一般情况：调用式里首先是一些按位置的普通实参，随后是一些关键字实参，其中可以出现若干个拆分实参和字典拆分实参 (调用时上述几种实参都可以没有)
- 函数调用式中实参和形参的匹配规则
  - 如果调用中有拆分实参，先将其打开得到一串实参，把它们排在调用式中的相应位置；解释器首先按照位置顺序地把这些实参与函数形参表中的形参一一匹配
  - 根据形参名字为各关键字实参确定对应的形参匹配；如果有字典分拆实参，将其打开得到的关联作为附加的关键字实参；如发生应该匹配的形参已有匹配的情况，就报告 **TypeError**
  - 未得到匹配的带默认值形参与其默认值匹配
  - 剩下的所有普通实参按顺序做成一个元组，约束到被调用函数的带星号形参 (如果有)，剩下的所有关键字实参做成一个字典，约束到双星号形参 (如果有)

函数参数总结：实参

- 实参太多或太少的情况
  - 如果完成上述匹配后，还存在没有得到约束值的形参 (即实参表达式太少)，或者存在多余的实参 (在函数定义的参数表里中没有带星号/双星号形参时可能出现这种情况)，将报告 **TypeError**
- 应该注意：
  - 由于存在拆分实参，通过拆分产生的实参个数在实际调用时才能确定，故函数调用的实参错误只能在运行中检查和发现
  - **Python** 提供了丰富的形参定义和实参使用形式，是为了实际编程方便，各种机制都有相应的用处
  - 但是，太过复杂的形参和实参会使程序中函数调用的意义不清晰，不应该随意混用上述机制