组合数据对象-4

- 元组，包装和拆分
- 字典（dict）
  - 构造，使用
  - 在字典元素上循环
- 集合（set 和 frozenset）：
  - 构造和操作

实例

创建元组

- 直接描述，或者从其他序列或迭代器转换

  避免以逐步扩充的方式构造（避免构造大量中间拷贝）

- 用描述式生成元组：

  tuple(x2 for x in range(10))** # 生成：

  (0, 1, 4, 9, 16, 25, 36, 49, 64, 81)

- 注意，形式为 **(x**2 for x in range(10))** 的表达式称为生成器表达式，不是**tuple**（圆括号括起，内部形式与表生成器相同）
  - 它并不实际构造组合对象（序列），而是一种迭代器，产生一系列可用的值，可以看作**range**的推广形式
  - 生成器可以用于生成各种序列，通过类型转换的方式，包括生成**list**、**tuple**等。也可以作为 **for** 中循环变量的取值源等

打包和拆分

- 元组构造相当于一种打包（**packing**），如

  tp = 123, 345, 'but', 567 # 构造出一个元组组合对象

- 也可以拆分（解包，**unpacking**）

  a, b, c, d = tp

  四个变量依次取 **tp** 各元素的值。变量个数必须正好合适

- 各种序列都可以拆分（这里要求两边的结构相同）：

  a, b, c = [1, 2, 3]

  a, [b, c] = [1, (2, 3)]

  a, (b, c), d = [1, (2, 3), 4]

  a, b, c = [2 ** i + 1 for i in range(3)]

打包和拆分

- 如果表 **list1** 的元素都是二元组或两个元素的表，可以写

```
for x, y in list1:
    … x … y …
```

循环迭代中**x**和**y**将取得**list1**中一个元素的两个成分

- 如果 **func1(…)** 返回二元组或者两个元素的表，可以写

```
x, y = func1(…)
```

x 和 **y** 分别得到 **func1** 返回值的一个成分

- 前面讲过在 **for** 语句头部直接使用拆分的情况

```
seasons = ['Spring', 'Summer', 'Fall', 'Winter']
for n, x in enumerate(seasons):
    print(n, x); … n … x …
```

打包和拆分

- 拆分还可以以统一形式用在许多地方

  实际中应该灵活使用

- 注意区分函数调用时的简单实参表达式和元组实参：
  - **f(a, b, c + 1)**　　# 用 3 个实际参数调用函数 f
  - **g((a, b, c + 1))** # 用一个元组实参调用函数 g
  - **h(a, (b, c + 1))** # 用两个实参调用函数 h，第一个实参是 a 的值，第二个实参是一个元组
  - **f1(2, 3,)** # 仍被看作两个整数类型的实参

    最后的逗号被忽略，不影响语义
  - **f2(2, (3,))** # 两个实参，第二个是单元素的元组

元组的使用

- 元组的另一常见用法是为循环变量提供不规则的迭代值，为此只需在迭代器的位置列出有关的值

```
for x in 3.44, 5.12, 6.77, 8.05, 4.332:
    print(x) # 这里可以写任何使用 x 的代码
```

可以加或者不加括号

- 序列值可以通过任意表达式获得，例如：

```
for x in square(3.44), sin(5.12), cos(6.77), 8.05:
    print(x) ) # 这里可以写任何使用 x 的代码
```

元组的使用

- 最常见的是用元组包装起若干相关对象，做成一个对象
  - 保存到变量或作为序列元素，或传入传出函数
  - 使用时应特别注意其不变性质
- 考虑定义一个有理数算术包
  - 有理数应是一种不变对象，构造或计算出的具体有理数，不会在程序运行中改变，运算符都是构造新有理数
  - 用两个整型元素的元组表示一个有理数，用第一个元素表示分子，用第二个元素表示分母
- 定义几个最基本的函数，封装有理数对象
  - **rational(n, d)** 做出以 **n** 为分子，**d** 为分母的有理数
  - **num(x)** 和 **den(x)** 分别取得有理数 **x** 的分子分母

有理数程序包

- 上述三个操作形成了对有理数的一种基本封装
  - **rational** 构造有理数，**num** 和 **den** 取出有理数的成分，用于实现有理数操作，必须满足的关系式（**r** 是有理数）

    rational(num(r), den(r)) == r

    num(rational(r, t)) = r　　den(rational(r, t)) == t
  - 如果有关有理数的操作都基于它们定义
- 基于上述操作，可以实现一套有理数算术运算
  - 做出一个有理数算术包，供人们使用
- 发现有理数的分子和分母会在计算中变得越来越大
  - 可以考虑化简，但在那里做？
- 数据抽象的作用：程序容易修改，改变行为，增加新功能

字典（dict，dictionary）

- 元素在序列里有一个位置（下标），支持基于位置的引用
  - 一个序列（或表）可以看作从下标到元素的有穷映射
  - 定义域只能是从 0 开始的连续整数区间
- 序列概念推广是广义的映射：字典，类型名为 dict
  - 概念上是从任意关键码集合到值集合的有穷映射
  - 如果需要保存一批数据，其中用下标不合适，希望用其他对象（例如字符串等）索引值，可以用字典
- 字典（dict 类型的对象）有两个基本操作
  - 把一个（任意类型的）值保存在字典里，将其约束于（关联于）一个给定关键码
  - 通过关键码获得与之相关的值

字典

- 存储信息并在适当的时候取用，是计算的基本需要
  - 字典实现基于关键码的存储和检索
  - 在复杂程序里应用广泛
- **Python** 的字典
  - 可以保存任何类型的值（对象）
  - 只允许用不变对象作为字典的关键码

    例如，可以用各种类型的数，字符串，元组（元素也必须是不变对象）作为字典的关键码
  - 例如，不能用 **list** 对象 **[1, 2]**，也不能用 **([1], [2])** 等

    情况：**Python** 需要求关键码的 **hash** 函数值。这个函数对内置的可变类型没有定义

字典的构造

- 直接描述：

  用 { } 括起一对对 关键码：值 二元组，描述一个字典

  dic1 = {'math' : 62751804, 'math-fax' : 62751801}

  建立一个包含 2 个关键码/值的字典

  dic = {} # 建立一个空字典

- 用 dict 类型名从二元组的表构造字典：

  faculty = dict([('math', 114), ('phys', 247), ('chem', 306)])

  得到 {'math': 114, 'phys': 247, 'chem': 306}

- 如果字典的关键码是简单字符串，可用关键字参数的形式

  dict(math = 114, phys = 247, chem = 366)

字典的构造

- 用字典描述式创建

  字典描述式的形式同表描述式（和生成器），用花括号

  基本形式： **{表达式：表达式 for 变量 in 迭代器}**

  同样允许 **if** 段，允许多个 **for** 段和 **if** 段

- 实例：

  {n : n3 for n in range(4)}**

  得到：**{0: 0, 1: 1, 2: 8, 3: 27}**

- 利用生成器，可以一下产生出一个有规律的大字典
  - 意义有规律（就是有统一的计算规则），意味着可以计算
  - 是一下做很多计算建立一个大型字典，还是需要时再按规则计算，需要根据实际情况考虑

字典操作

- 字典支持一批操作，列举如下

  设 dic 是一个字典，k 是个关键码，v 是一个值

- 最基本的操作（一对）
  - dic[k]　得到 dic 里与 k 关联的值，dic 里不存在 k 时报错
  - dic[k] = v　设 dic 里与 k 关联的值为 v（无论原来有没有）
- len(dic) 得到 dic 的元素（关键码/值关联）个数
- del dic[k]　删除一个关联，不存在 k 时报错
- k in dic　判断，当 dic 里有 k 时返回 True，否则 False
- k not in dic 与 k in dic 相反
- iter(dic)　得到在 dic 关键码上迭代的迭代器，顺序内部确定，不能控制（可以用 sorted 排序）

字典操作

- **dic.get(k), dic.get(k, default)** 得到 **dic** 中与 **k** 关联的值。如果不存在 **k**，得到值 **None** 或 **default**
- **dic.copy()** 建立 **dic** 的拷贝，不建立关键码/元素的拷贝

  注意：关键码是不变对象，值可以是可变对象。两字典共享值
- **dic.clear()** 删除 **dic** 里的所有元素，将其变回空字典
- **dic.pop(k)** 从 **dic** 删除 **k** 并返回 **k** 的关联值，无 **k** 时报错
- **dic.pop(k, v)** 与上面类似，但在 **dic** 里没有 **k** 时返回 **v**
- **dic.popitem()** 以二元组 **(k, v)** 的形式返回 **dic** 里任一元素
- **dic.update([*other*])**

  用另一个字典 **other** 更新 **dic**（可能增加或修改元素）。可以用一批关键码参数做这个操作

  faculty.update(bio = 361, math = 109)

字典操作

- **dic.setdefault(*k* [, *default*])**

  如果 **dic** 里有关键码 **k** 就返回其关联值，如果没有就加入 **k** 与 **default** 的关联，无 **default** 时用 **None** 作为 **k** 的关联值

- 与字典相关的一个概念是“字典观察对象”（**dictionary view object**），这是一种依附于具体字典的对象，如果被依附的字典改变，从观察对象也将看到有关改变

  下面三个操作生成 **dic** 的观察对象：

  dic.keys() 得到 **dic** 所有关键码，顺序内部确定

  dic.values() 得到 **dic** 的所有值，顺序内部确定

  dic.items() 得到 **dic** 所有关联

  可以认为关联的形式为 **(k, v)**

在字典上迭代

- 字典不是序列，但编程中也经常需要顺序处理一个字典里的所有元素。下面是几个典型的迭代模式：

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

for k in sorted(dic.keys()): # 按关键码排序处理
    ... k ... dic[k] ...

for k, v in dic.items(): # 按默认的元素顺序处理
    ... k ... v ...
```

第一个模式可简化为（Python 自动取 keys() ）：

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

字典的应用

- 字典可以用于保存各种需要查询的信息，例如
  - 电话号码本：姓名到号码，号码到姓名
  - 用户记录：用户名（或用户号）到用户信息
  - 银行账户：账户编号到账户信息
  - 计算机系统的用户登录管理：帐号到密码
  - 历史信息记录，如 **2011.7.25** 北京降雨量
  - 列车时刻表，航班实时位置信息记录
  - **Python** 的全局作用域（变量与其值关联）
  - 程序中需要保存的中间数据

    如果需要按某种名字保存和检索使用

集合

- 数学的集合就是一批元素的汇集
  - 元素无所谓的“顺序”，只支持元素判断
  - 有一组重要的集合操作：并集，交集，补集等
- 程序里也经常需要具有集合性质的对象，Python 为此提供了集合类型，其基本集合类型是 set，也是一种复合类型
- set 对象的特点：
  - 元素只能是不变对象，必须能比较是否相等。内置的数值类型、字符串、bool对象，及它们的元组都满足这些要求
  - set 对象里的元素唯一，无重复元素，元素之间无顺序关系
- frozenset 是不变集合类型，支持所有不改变集合对象的操作，这种对象只能用 frozenset(...) 构造

集合

- 基本集合构造：
  - 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 的值为元素
  - 用 set(...) 可以从序列或可迭代对象生成相应的集合
  - 空集用 set() 生成（注意，{} 生成空字典）
  - 这几种形式都可以用于生成 frozenset
- 集合与字典的关系
  - 都用 {...} 描述，元素写 “表达式 : 表达式” 就是字典
  - {...} 里写的是其他形式的表达式，生成的就是集合
  - 空集只能用 set() 生成，空字典可以用 {} 或 dict()

集合

- 集合也可以用描述式生成，用 { 生成器表达式 } 的形式
  - 元素为 表达式 : 表达式 生成字典
  - 元素为简单形式的 表达式 时生成集合
  - 例：**frozenset(n ** 2 for n in range(-20, 20))**
- **set** 和其他类型之间的转换
  - 与 **tuple** 之间可以来回转换，转到集合将消除重复元素，未必保持原顺序，**set** 转到 **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'}**，顺序内定
  - **set(dic)** 得到 **dic** 的关键码集合

集合操作

- 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

```
for x in 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，可以先把集合转换为 **list** 并排序后循环

- 集合之间可以比较相等和不等（==，!=）

两个集合的元素完全相同时 == 得到 **True**，否则 **False**

集合相等是数学概念

集合操作

- 集合关系判断：
  - **s <= s1, s.issubset(s1)** 当 **s** 为 **s1** 的子集时返回 **True**

    s < s1 在 **s** 为 **s1** 的真子集时得到 **True**

    s 是 **s1** 的子集，如果 **s** 的元素都是 **s1** 的元素。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 在 **s** 为 **s1** 的真超集时得到 **True**

    超集关系是子集关系的逆
  - **s.isdisjoint(s1)** 在两个集合不相交时返回 **True**

    不相交就是没有公共元素

集合操作

- 生成新集合的运算：
  - **s1.union(s2, ...),　s1 | s2 | ...**

    生成一个新集合，是这些集合的并集
  - **s1.intersection(s2, ...),　s1 & s2 & ...**

    生成一个新集合，是这些集合的交集
  - **s1.difference(s2, ...),　s1 – s2 – ...**

    生成新集合，是这些集合的差集
  - **s1.symmetric_difference(s2)**

    生成 **s1** 和 **s2** 的对称差集，即由所有属于 **s1** 但不属于 **s2** 的和属于 **s2** 到不属于 **s1** 的元素构成的集合
  - **s.copy()** 生成 **s** 的一个拷贝

集合操作

- 集合元素操作（修改集合）
  - **s.add(x)** 将元素 **x** 加入集合 **s**
  - **s.remove(x)**

    从 **s** 里删除元素 **x**，如果没有就报错
  - **s.discard(x)**

    如果 **s** 里有 **x** 就删除它，没有时什么也不做
  - **s.pop()**

    从 **s** 里删除某个元素并返回它，具体元素内定

    如果 **s** 为空，操作报错
  - **s.clear()** 清除 **s** 里的所有元素

集合操作

- 集合更新操作（修改集合）
  - **s1.update(s2, ...),　s1 |= s2 | ...**

    修改 **s1**，使之也包含属于其他集合的元素
  - **s1.intersection_update(s2, ...),　s2 &= s2 & ...**

    修改 **s1**，使之只包含同属所有集合的公共元素
  - **s1.difference_update(s2, ...),　s1 –= s2 | ...**

    修改 **s1**，从中去除属于其他集合的元素
  - **s1.symmetric_difference_update(s2)**

    修改 **s1**，使之只包含所有只属于 **s1** 或 **s2** 两者之一但不同时属于两者的元素

    这样的集合称为两个集合的对称差

集合

- 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：命名版本和运算符版本
  - 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，而命名版本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

  显然集合更新操作不能用于 **frozenset**

  但集合更新的其他集合（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

生成式和作用域

- 生成式中**for**片段引进的变量，其作用域是这个生成式的基本表达式和位于这个**for**片段之后的部分。看下面两个例子：

```
>>> [i1 + i2 for i1 in range(i2) for i2 in range(4)]

Traceback (most recent call last):
  File "<pyshell#118>", line 1, in <module>
    [i1 + i2 for i1 in range(i2) for i2 in range(4)]
NameError: name 'i2' is not defined
```

在第一个**for**片段里变量**i2**无定义

- 但下面生成式没问题：

```
>>> [i1 + i2 for i1 in range(4) for i2 in range(i1 + 2)]

[0, 1, 1, 2, 3, 2, 3, 4, 5, 3, 4, 5, 6, 7]
```

因为在第二个**for**片段里**i1**已经有定义

生成式和作用域

- 生成式引进局部作用域
  - 这种作用域同样出现在外围作用域中
  - 生成式里可以使用之外的变量，局部变量有遮蔽问题
- 举例：

```
def func(n):
    list1 = [x * n for x in range(n)]
    list2 = [n ** 2 for n in range(10)]
    print(list1)
    print(list2)
    print(n)
```

生成式局部引进的 n 与函数参数 n 无关，只在生成式里起作用

这也是一种作用域嵌套