组合数据对象-3

- 共享和拷贝，函数的表参数
- 序列的比较
- 不同类型元素的表
- 用表记录计算的中间结果
- 表的遍历，典型遍历模式
- 两个重要内置函数：**map** 和 **filter**
- 表和字符串的相互转换
- **tuple**（元组）：不变序列类型和对象

共享和拷贝

- 两个变量可以共享同一个值（对象）

```
x = y = [1, 3, 6]
z = x
w = [1, 3, 6]
```

- 只看几个变量的值，不能区分其中的共享情况。通过赋值

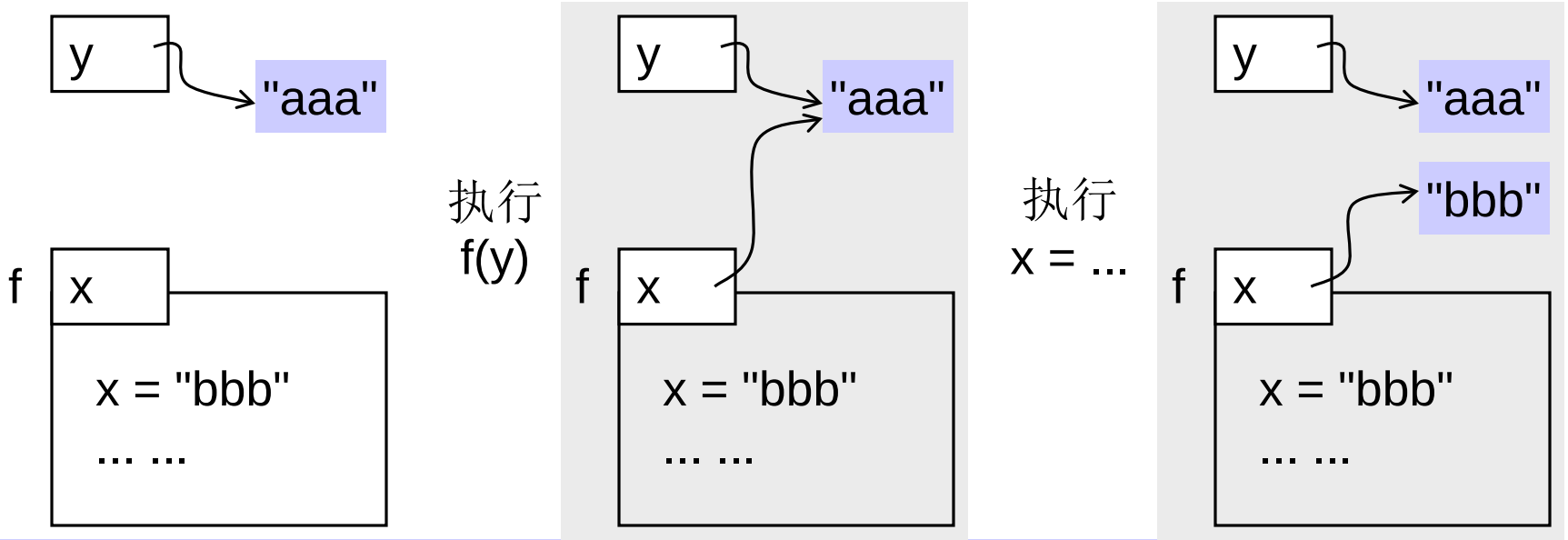
```
x[2] = 10  # y 和 z 值都变了（它们共享），不影响 w
```

- 说明：
  - x, y 和 z 的值是同一个对象
  - w 的值是另一对象，开始时正好与前一对象的内容一样
- 特别注意：空表表达式 [] 也生成一个表对象

```
a = b = []; c, d = [], []  # 情况不同，a和b共享，c和d无关
```

函数参数

- 函数参数是局部变量
  - 函数调用时，参数约束到实参表达式计算的结果（实参是一个对象，具体是什么由实参表达式决定）
  - 在函数里给（形式）参数赋值，将使其约束到另一对象，不会影响函数调用的实参



函数的表参数

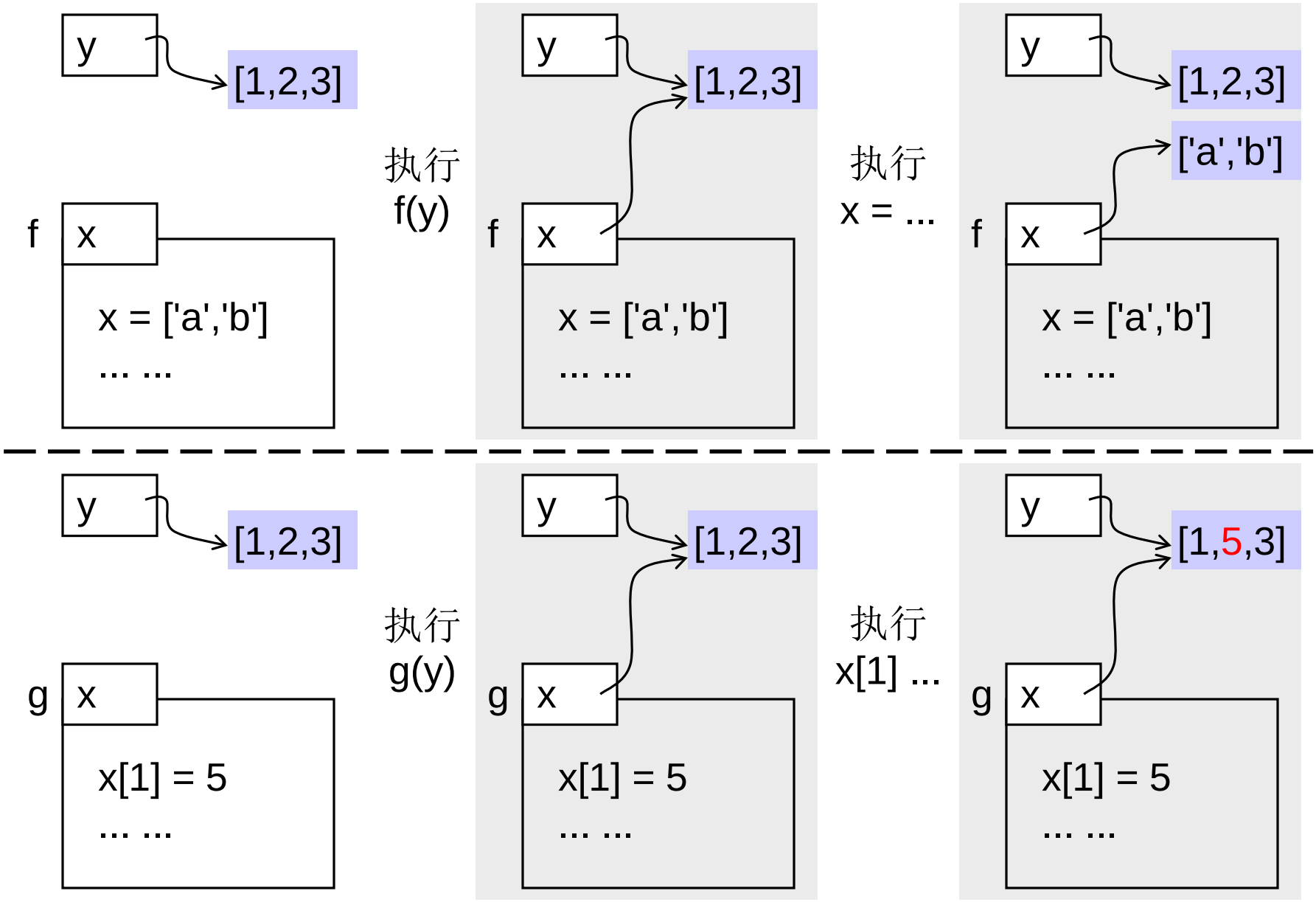
- 如果送给函数形参的实参是一个表
  - 给形参赋值，同样是改变其约束，不会改变实参表
  - 如果通过形参执行修改表的操作，情况就不一样了
    - 由于形参引用着的是一个表对象
    - 通过形参修改表，就会真的修改了那个表
- 例子：假定 **g(x)** 里只有语句 **x[1] = 5**，在

  y = [1, 2, 3]

  g(y)

  之后，**y** 的值就变成 **[1, 5, 3]**

```
def g(x):
    x[1] = 5
```



看一个简单例子：定义函数实现表元素反转的操作

比较复合对象

- 有时需要比较复合对象，这时会出现两种“相等”
  - 两个对象“内容”是否相同（“值”相等）
  - 是否为同一个对象
- 前面用 ==，!= 判断相等和不等，

  实际上是判断两个对象的“内容”是否相同
- 运算符 is 和 is not 判断是否同一个对象

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

  a == b 和 a == c 都会给出 True

  a is b 给出 False，而 a is c 给出 True
- 用== 判断的相等称为“按结构相等”。两个对象相等，当且仅当其结构相同，而且元素相等。对象类型不同时结果为 **False**

比较复合对象和序列

- 任何两个对象都可以用 **is** 和 **is not** 比较
  - 判断是否为同一个对象，总能得到 **True/False** 结果
  - 内置函数 **id** 取得对象标识，每个对象有唯一标识（**identity**）
  - **is/is not** 比较对象的 **id**（标识），不涉及内容，不会出错
- 此外，两个同类型的序列可以比较大小（**list** 也可以比较）
  - 用 **<=，<，>=，>** 运算符
  - 比较的方式与字符串类似，采用“字典序”，从左到右逐一比较对应的元素，第一对不同元素的大小决定序列的大小
  - 如果元素本身还是复合对象，将递归处理
  - 如果用 **<=，<，>=，>** 比较不同类型的对象，或者比较元素时遇到不同类型的元素，操作将报 **TypeError**

实例：反转表中元素的函数

- 作为修改表中内容的实例，考虑定义一个功能函数，其功能与序列的 **reverse** 操作类似

  def rev (lst): … # 反转实参表里的元素

  希望调用 **rev (list1)** 将 **list1** 里的元素实际反转

- 操作的方法：
  - 用一对变量，指向表中需要反转的一对元素
  - 交换两个元素的值
  - 修改两个变量的值，使之相向而行，指向另一对元素并继续
  - 直到两个变量重合（只剩一个元素），或者交错（无元素）
- 结合 **Python** 语言的机制，结合 **for** 循环、负数下标值和并行赋值语句，函数的实现很简单

不同类型元素的表

- 一个表的元素可以不属于同一个类型，例如

  ["Zhang San", 22, "Math"]

  ["Laptop", "Lenovo", "S3", 9300.00, [2014, 3]]

- 这类数据也常用后面介绍的元组（**tuple**）表示

  表是可变结构，**tuple** 是不变结构

  不变的数据记录，更适合采用 **tuple** 表示

- 许多数据可以用表的表记录（一个元素表记录一项数据）：
  - 学校的学生记录（姓名，学号，...）
  - 商品库存，等等
  - 这类数据也常用后面介绍的字典（**dictionary**）表示

用表记录得到的结果

- 程序中可能多次需要某个（某些）计算的结果
  - 可以考虑引进变量，保存通过计算得到的结果

    再次需要时直接拿来使用
  - 记录结果和重复计算可以相互替代（计算和存储）
  - 如计算代价高，或者重复使用次数多，存储结果可能有利
- 前面很多例子里用一个或一组变量保存计算的中间结果，实际上有时也可能需要用表保存，典型情况：
  - 需要保存的中间数据很多
  - 编程时不能确定要保存的数据的项数
  - 需要按特定的同样方式使用这些数据（表元素可以通过一个变量名和下标表达式的显式使用）

递归 Fibonacci 函数的改进

- 以 Fibonacci 数的计算作为存储计算结果的实例
- 考虑前面递归实现的求斐波那契值的函数
  - 函数执行中出现大量重复计算，多次重复计算各 F_k 的值
  - 造成的结果：对不大的 n，需要极长时间才能算出结果
- 若记录算过 F_k，再次需要时直接取用不重新计算，可能节省时间
  - 保存计算得到的结果，积累已知信息，有可能提高效率；但要付出存储的代价，其量与 n 成正比
  - 权衡利弊，对 Fibonacci 数的计算，可能值得这样做
- 基本考虑：要保存一批 F_k 值，通过 k 检查和使用

  需要保存的数据项数与调用 fib(n) 时的 n 值有关，定义函数时不能确定，可以考虑用表记录

递归 Fibonacci 函数的改进

- 函数实现的基本想法：
  - 建一个表，保存计算中得到的结果（得到的 F_i）
  - 函数调用中先检查这个表：如果算过就直接取出使用；没算过的就算出来，返回结果前先存入表中相应位置
- 函数名仍然用 **fib(n)**，用一个表记录计算出的 **Fibonacci** 值

  从 **0** 到 **n** 需记录 **n + 1** 个元素，初始置 **0**（或其他合理值）
- 从函数定义的合理性考虑，把这个表作为为 **fib** 局部变量的值
  - 定义局部函数 **fib0**，递归定义，完成主要计算
    - 检查内部表，如果有结果（非 **0**）就取出返回
    - 否则就计算并返回，也保存到相应表项里
  - **fib** 的工作：做好初始表，然后调用 **fib0**

表的遍历

- 后面还会遇到很多用表保存中间信息的实例
- 程序里经常使用同类型元素的表，这类结构有一些典型操作
- 一类重要的表操作称为表遍历
  - 顺序扫描和处理表里的每一个元素
  - 对表中每个元素执行同样的操作
- 遍历操作的分类：
  - 做出一个新表，其各元素是原表元素的操作结果，实现一种表变换，从一个表做出一个结构相同的新表
  - 通过遍历表累积元素包含的信息，得到累积结果
  - 直接修改被操作的表，遍历的效果通过表的修改体现
  - 其他操作，例如选择一些元素做成新表等

通过遍历生成新表

- 设 lst 是一个表，元素都是某类型的对象；f 是参数是这个类型的元素的函数，它产生一个结果
- 遍历 lst 生成一个新表，其元素为 f 作用的结果，操作过程应从 lst 的每个元素算出一个新元素加入新表。框架是：

```
new_list = [ ]
for x in lst :
    new_list.append(f(x))
```

- 例（假设已导入 math 包里的 sin 和 pi）：

```
lst = [2*pi*(x/100) for x in range(101)]
sin_list = [ ]
for x in lst :
    sin_list.append(sin(x))
# 这时 sin_list 里保存着计算出了一批值
```

通过遍历做累积

- 累积：用（一个）变量记录已得到的信息

  通过计算中不断更新该变量的值，记录下更多信息

- 在遍历表的过程中累积由表元素得到的信息，经常需要这种工作
  - 用一个变量，设为 s，遍历前给定初值，处理遇到的表元素时，将相关信息累积到变量中

    如何积累，可以通过一个函数 g(s, x) 将 x 积累进 s
  - 累积变量的最终值是这段计算的结果
- 下面操作框架通过函数 g 完成累积

```
s = initValue
for x in lst :
    s = g(s, x)
```

处理表的高阶函数

- 通过遍历修改表的操作与前面两类操作类似

  处理中不积累信息，也不积累新表的部分结构

  直接修改表元素

- 易见：表遍历操作是表上的典型操作
  - 存在一些操作模式
  - 具体操作体现在同样模式中对每个元素的处理方式
- 在 **Python** 里可以实现这类操作模式
  - 把具体的操作模式定义为函数
  - 通过函数的函数参数提供具体元素操作
  - 下面看几个例子

处理表的高阶函数

- 基于一个表构造同样结构的新表：**map**

  map(fun, lst)

- 累积表元素的信息：**reduce**

  reduce(fun, lst, init)

- 遍历和修改：**do_each**

  do_each(op, lst)

- 选择表里的一些元素

  filter(pred, lst)

- 代码和应用

- 实例：计算前 **n** 个 **Fibonacci** 数中所有整除 **3** 的数之和

表操作的应用和分析

- **map-reduce** 是 **Google** 的网络搜索系统的核心技术
- 处理表的高阶函数实现表操作的分解，分别考虑：
  - 对每个元素的操作
  - 对表的遍历方式和表操作的组合
  - 可能用于把处理一组数据的复杂计算过程分解为一些顺序的步骤，实现每一步只需实现具体的元素操作
- 重看前面实例，考虑计算中的开销？
  - 我们定义的 **map** 和 **filter** 生成新的表
  - 每步生成的表都可能很大
- 使用 **Python** 的内置函数 **map** 和 **filter**，可以避免这种（可能巨大的）存储开销

内置函数 map 和 filter

- map 和 filter 都有一个函数参数和一个可迭代对象参数，其功能与前面的定义类似，但生成迭代器，不生成新表
- filter(fun, iterable) 得到一个可迭代对象，fun 是谓词函数

  list(filter(lambda x : x != 0, numbs)) 得到非 0 元的表

  filter(lambda x : x > 0, xl) 得到 xl 中去除负值的迭代器
- map(fun, iterable, ...) 得到可迭代对象，元素是对 iterable 各元素顺序调用 fun 的结果。fun 多元时需用多个 iterable 为其提供成组参数，做到第一个迭代器用完为止

  list(map(lambda x : x**2, lst)) 得到各元素求平方的表
- filter 和 map 得到的对象常用于生成所需的序列（上例即是），也常用作 for 语句的迭代描述
- 在计算前面的 sum_fibs 中，并不实际构造任何的表

字符串和表

- 介绍几个字符串和表之间来回变换的函数
- **s.split(*sep=None*, *maxsplit=-1*)**
  - 得到字符串 **s** 切分为一些子串的表
  - 如果不给 **sep** 参数，默认为是由连续空白字符（空格/换行/制表符）分隔的子串，开头或结尾的空白字符全部丢掉
  - 通过 **sep** 可以指定特殊的分隔符（指定切分方式）
  - **maxsplit** 指定（从左向右处理的）最大切分项数
    - 到达项数后剩下的串作为一个子串放在结果表的最后
    - 默认（或值为 **–1** 时）做完整个串的切分
- **s.rsplit(*sep=None*, *maxsplit=-1*)** 按从右到左的顺序切分。显然，只在指定切分项数的情况下才有用

字符串和表

- **s.splitlines([*keepends*])** 得到把 **s** 切分为正文行的表
  - 以串中的换行作为分隔符
  - 如果没有参数，换行符号本身不包含在子串里
  - 如果给参数 **keepends** 值 **True**，行最后的换行符保留
- **sep.join(lst)** 相当于 **split** 的逆，用串 **sep** 作为分隔符把参数 **lst**（应包含一些字符串）连成一个串。例

  ", ".join(["break", "continue", "return"]) 将得到

  "break, continue, return"
- **s.strip()，s.lstrip()，s.rstrip()** 前面已经介绍过，它们得到 **s** 删去两端空白（或删去左边空白，或删去右边空白）后的串

  在整理字符串时也经常使用

一个简单的计算器

- 考虑实现一个简单的交互式计算器
  - 反复要求用户输入只包含一个二元运算符的简单算术式
  - 输出计算结果
- 程序的基本框架：
  - 基本部分是一个输入控制的循环
  - 确定一种循环结束方式，例如判断输入是否为 quit
  - 分解每个输入行，取得运算符和运算对象
  - 计算并输出，而后重复
  - 为简单，要求用户用空格分隔运算数和运算符

元组（tuple）

- 元组（tuple）是一种序列类型，
  - 元组对象是不变对象，创建后不能修改
  - 可以包含任意多个（同类型或不同类型的）元素
  - 支持所有不变序列操作，没有其他特殊操作
- 创建元组对象：
  - 直接描述

    (1, 2, 3)　　可以不写括号：　**1, 2, 3**
  - 用 **tuple(**可迭代对象**)** 的形式创建

    tuple(range(10)) 得到元素为 **0** 到 **9** 的元组

    tuple("abc") → **('a', 'b', 'c')**

元组

- 两个情况说明：
  - **tuple()** 建立空元组对象
  - 单元素元组要写逗号，括号不重要（系统总显示括号）

    x = 12,
- 内置函数 **enumerate** 构造元组的序列。例如

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
>>> list(enumerate(seasons, start=1))
[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]
```

元组

- Python 教程里的例子（及其他）

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # 元组可以嵌套:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # 元组里可以有可变元素:
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
>>> v[1][2] = 5
>>> v
([1, 2, 3], [3, 2, 5])
```