处理表的基本方法

- **len(lst)** 得到表 lst 的长度，即表元素个数
- 写循环顺序取 (遍历) / 操作表里的所有元素，常用模式：

```
for i in range(len(lst)) :          # 用表长度生成循环范围
    ... lst[i] ...                  # 也可以用 while 语句

for x in lst :          # 简化写法，直接在表上循环
   ... x ...            # 此种写法只能取元素值，不能修改元素值
```

  - 可迭代对象 (迭代器或序列对象) 可直接用作 for 循环变量的取值源，如字符串、表等，但只适用于按顺序取所有元素值
- 也可以处理表中几个元素或一段元素
  - 处理个别元素时，直接通过下标访问
  - 处理表中一段元素，可以利用 for / while 循环语句，通过下标表达式访问指定元素

表的遍历操作：示例

```
###### 表上的循环：计算表中元素之和
def sum1(numbs):
    s = 0
    for i in range(len(numbs)):
        s += numbs[i]
    return s

def sum2(numbs):
    s = 0
    for x in numbs:
        s += x
    return s
```

```
###### 统计某个值在一个表里出现的次数
def count(x, lst):
    n = 0
    for y in lst:
        if y == x:
            n += 1
    return n
```

实例 1：向量内积

- 数据的内部实现：用浮点数表表示 n 维向量
- 定义函数实现向量的内积 (点积)
  - 函数应该以表示向量的两个表为参数，返回一个浮点数
- 函数定义中的细节处理：
  - 需要先确定两个参数是等长，否则返回 float("nan") 或者报告错误/引发异常
  - 也可以进一步检查表的各元素是否是浮点数
  - 循环乘两个向量的各项并累积
  - 返回结果 (演示略)

表中的元素

- **Python 表中元素可以属于不同类型 (但同类型元素的表最常用)**
  - 表的元素可以是任何数据对象，包括表；例如

```
>>> [[1, 2, 3, 4], ["a", "b", "c", "d"], ["age", 23], 13]
```

    - 实例 2，价格汇总函数 (演示)

- 可以用任意表达式给表元素赋值

```
>>> lst = [1, 2, 3]
>>> lst[2] = [1, 2, 3]
>>> lst
[1, 2, [1, 2, 3]]
```

- 嵌套的表推导式可以建立元素为表的表 (即，两层的表)

```
>>> [[1 if i==j else 0 for i in range(5)] for j in range(5)]
[[1, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, 0, 1, 0, 0], [0, 0, 0, 1, 0],
[0, 0, 0, 0, 1]]
```

实例 3：求一元多项式的值

- 一元多项式 $a_0 + a_1x + a_2x^2 + \cdots + a_kx^k$ 的内部实现
  - 用长为 k+1 的浮点数表 p 保存系数，即元素 p[i] 存系数 a_i
- 定义函数 evalPoly(p, x)，计算并返回上述表示下的多项式 p 在给定点 x 的值
  - 直接实现：p 是一个 k+1 项的表，按顺序取项计算即可

```
###### 求一元多项式在给定点的值
###### 其中多项式用表表示，表的元素表示多项式中各次项的系数
p1 = [1, 3, 3, 1]          # 1 + 3 * x + 3 * x**2 + x**3
p2 = [2, 5, 6, 2, 1]       # 2 + 5 * x + 6 * x**2 + 2 * x**3 + x**4
#### 求 p 在 x 的值
def eval_poly(p, x):
    val = 0
    for i in range(0, l
        val += p[i] * x
    return val
```

改进：循环体中，可用一次乘法代替乘幂运算

➔ 利用多项式的 Horner 范式

$$\Big(\big(\cdots\big((a_kx + a_{k-1})\cdot x + a_{k-2}\big)\cdots + a_2\big)\cdot x + a_1\Big)\cdot x + a_0$$

```
#### 另一种实现：基于 Horner 范式
def eval_poly2(p, x):
    val = 0.0
    for i in range(len(p)-1, -1, -1):    # 注意 range 的实参
        val = val * x + p[i]
    return val
```

```
def eval_poly3(p, x):
    val = 0.0
    for a in p[::-1]:    # p[::-1] 生成 p 的倒序切片
        val = val * x + a
    return val
```

```
#### 内置函数 reversed(seq)
#### 生成序列对象（表、字符串）的反向迭代器（结果非序列）
#### 即，从后向前逐项给出值；可以用在 for 语句里描述迭代方式
def eval_poly4(p, x):
    val = 0
    for a in reversed(p):
        val = val * x + a
    return val
```

实例 4：生成斐波那契数列 (用函数生成表)

- 定义函数，以递推的方式，生成保存斐波那契数列前 n 项的表
  - Python 函数的返回值可以是任何对象，包括表对象、函数对象等

```
###### 实例4：生成包含斐波那契数列前 n 项的表
def gen_fibs(n):
    fibs = [0] * (n + 1)
    fibs[1] = 1
    for i in range(2, n + 1):
        fibs[i] = fibs[i-1] + fibs[i-2]
    return fibs

fs = gen_fibs(20)
print("First", len(fs), "Fibonacci numbers:", fs)
```

Python 的自动内存释放/垃圾回收 (简介)

- **Python** 程序运行中所建立的对象，只要还可能被使用 (即还被某变量所关联)，该对象会一直存在
  - 对象的存在并不依赖于其所关联的变量
  - 如果某个对象再也不可能被使用，**Python** 系统 (的一个子系统 —— 存储管理系统) 会自动将其回收
  - 因此，编程者并不需要考虑内存使用的各种细节

garbage collection -- 垃圾回收

释放不再被使用的内存空间的过程。Python 是通过引用计数和一个能够检测和打破循环引用的循环垃圾回收器来执行垃圾回收的。可以使用 gc 模块来控制垃圾回收器。

实例 4：生成斐波那契数列 (续)

- 实现二：也可以在计算过程中逐步建立结果表

```
###### 逐项生成项值后用 append 操作加在表最后
###### 表操作 append：在表末尾增加一个元素
def gen_fibs1(n):
    fibs = [0, 1]
    for i in range(2, n + 1):
        fibs.append(fibs[-2] + fibs[-1]) # -1 和 -2
    return fibs

fs2 = gen_fibs1(20)
print("First", len(fs2), "Fibonacci numbers:", fs2)
```

  - 小细节：迭代过程中，结果表 **fibs** 不断扩充；但是，当前表中最后两个元素的下标总是 **-1** 和 **-2**

实例 5：筛法求素数 (1)

- 新问题：定义函数 sieve(n)，生成不超过参数 n 的素数表
- 埃式筛法 (Sieve of Eratosthenes)：经典的求素数方法，基本计算过程如下
  - 取自然数序列 2 ~ n
  - 2 是素数，从 2 之后的序列中划去 2 的所有倍数
  - 找到序列中下一个未被划去的元素 x (是素数)，从 x 之后的序列中划去 x 的所有倍数
  - 重复上面的操作，直到序列中只剩素数 (无法再划去任何数)

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Prime numbers

动图来自网络

实例 5：筛法求素数 (2)

- 函数中的数据表示：
  - 一个自然数表，其中顺序地记录自然数序列 0 ~ n (表中元素值与下标对应)
  - 用 '将元素置 0' 表示该数已经被划掉，即为非素数 (思考：还有哪些可行的处理方式？)
    - 建好初始自然数表后，需把前两个元素置 0
    - 从 2 开始，对每个新确定的素数，划去其倍数
- 实现时的细节：需用两层嵌套的循环
  - 外层循环：如果 n 不是素数，则 n 一定有小于等于其平方根的因子 ➡ 循环至 '不小于 n 的平方根'
  - 内层循环：检查并划去当前素数的倍数，可用 for 语句
  - 最后，还需收集所有非 0 元素构造出素数表 (演示)

元组 (tuple)

- 元组 (类型名：tuple) ：Python 内置序列类型
  - 可以包含顺序排列的任意多个 (同类型或不同类型的) 元素
  - 元组对象创建之后不能被修改：不能加入/删除元素 (元组结构不变)，不能给其元素赋值 (元组内容不变)
- 创建元组对象
  - 直接描述

```
>>> tp1 = (1, 2, 3)
>>> tp2 = 1, 2, 3          # 描述元组时可以不写括号
```

  - tuple(可迭代对象)

```
>>> tuple('abc')           # 类型转换，得到 ('a', 'b', 'c')
```

  - 元组推导式

```
>>> tuple(s + t for s in "abcd" for t in "123")
```

元组

- 空元组：用 tuple() 或 () 建立
- 对于'单元素元组'：描述时必须要写逗号 (括号并不重要，系统总显示括号)

```
>>> x = 12,                    # (12，)
>>> y = "abc",                 # ('abc',)
>>> z = (12)                   # 12
```

```
>>> t = 12345, 54321, 'hello!'
>>> t
(12345, 54321, 'hello!')
>>> t[0]
12345
>>>
>>> # 元组可以任意嵌套
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

示例

```
>>> # 元组是不变对象
>>> t[0] = 88888
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    t[0] = 88888
TypeError: 'tuple' object does not support item assignment
>>>
>>> # 元组里的元素可以是可变对象:
>>> 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])
```

```
>>> # 表的元素也可以是元组
>>> [(i,j) for i in range(3) for j in range(3) if i <= j]
[(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2)]
```

元组的常见用法

- 利用元组为 for 语句的循环变量提供不规则的迭代值
  - 加或者不加括号皆可

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

  - 序列值可以通过任意表达式获得，例如：

```
for x in 3.44**2, sin(5.12), cos(6.77), 8.05**3:
    print(x)
```

- 用元组包装若干相关的数据对象，建立固定组合对象
  - 比如，用三元浮点数元组表示三维坐标；用二元整数数组表示有理数；用元组封装个人身份信息等
  - 使用元组时，应特别注意其不可变动的性质

序列

- 序列涵盖了具有共同性质的一系列类型 (str, list, tuple 等)
  - 序列对象都可以包含任意多个元素 (组合对象)
  - 序列对象中的元素按线性顺序排列，每个元素都有下标，下标从 0 开始
  - 序列对象可以作为一个整体，给变量赋值，传入/传出函数等
  - 序列对象中的元素也是对象，可以通过下标表达式访问；对序列对象的操作一般通过对其元素的操作实现
- 序列类型之间的差异
  - 不同类型的对象对元素的类型要求不同：str 对象的元素只能是字符，list 和 tuple 对象的元素可以是任何对象
  - 不同类型的对象建立后是否可以变动 (被修改)
    - 可变 (mutable) 类型和不变 (immutable) 类型

不变类型和可变类型

- 不变类型的对象：称为不变对象，在创建后不会变化 (不能修改)
  - 不变对象的操作：取得对象内部的信息和创建新对象
  - 各种基本数值类型和逻辑类型是不变类型
  - **str, tuple** 是不变序列类型，它们的对象是不变对象
  - **range(m, n, d)** 的结果是一个 **range** 类型的迭代器对象
    - **range** 是一种元素值为整数的 (受限的) 不变序列类型，只支持几个序列操作

```
>>> range(10)[2]
2
>>> range(100, 200)[3:84:6]
range(103, 184, 6)
```

- 可变类型的对象：称为可变对象，在创建后可以变化 (可被修改)
  - 发生变动后，仍然是这个对象 (标识不变)，但其内容或结构变了
  - 造成变动的原因一般是成分被修改 (重新赋值)，或者是结构(和内容)的改变 (如一个表被 **append** 新元素)

所有序列类型支持的公共操作

1	x in s	如果 s 中有元素等于 x 则 True，否则 False
2	x not in s	如果 s 中有元素等于 x 则 False，否则 True
3	s + t	s 和 t 的拼接序列
4	s * n 或 n * s	s 的 n 个拷贝的拼接
5	s[i]	s 的第 i 个元素，从 0 开始
6	s[i:j]，s[i:j:k]	s 的切片，从下标 i 到 j 的一段做出的序列，如果有 k 则按步长 k 取元素
7	len(s)	s 的长度
8	min(s)	s 的最小元素
9	max(s)	s 的最大元素
10	s.index(x[, i[, j]])	x 在 s 里首次出现的下标，下标查找范围 [i,j) 如 s 里面不包含 x，报 ValueError [ ] 表示可选：s.index(x), s.index(x, i), s.index(x, i, j)
11	s.count(x)	x 在 s 里出现的总次数

可变序列的变动操作

1	s[i] = x	给 s 中下标为 i 的元素赋值
2	s[i:j] = t	用可迭代对象 t 的内容替代 s 从 i 到 j 的切片； s[i:j:k] = t 的作用类似，其中 i、j、k 可省略表示用默认值，在做这个操作时要求 t 的元素个数正好合适
3	s.append(x)	在序列末尾添加元素 x，等同于 s[len(s):len(s)] = [x]
4	s.insert(i, x)	把 x 插入 s 里由下标 i 指定的位置，等同于s[i:i] = [x]
5	s.extend(t)	用 t 的内容扩展 s，等同于 s[len(s):len(s)] = t
6	s.copy()	创建一个 s 的拷贝 (等同于 s[:])
7	del s[i]	从表中删除下标为 i 的元素；del s[i:j] 相当于 s[i:j] = []； del s[i:j:k] 删除指定元素
8	s.clear()	清除 s 的所有元素 (等同于del s[:])
9	s.pop(), s.pop(i)	返回 s 里下标 i (默认为最后) 的元素，并将其从 s 删除
10	s.remove(x)	删除 s 里首个满足 s[i] == x 的元素 如果 s 里没有元素等于 x，报 ValueError
11	s.reverse()	反转 s 里的所有元素 (前后元素的位置倒置)

序列对象的比较

- 相同类型的两个序列 (如 **list, tuple, str** 等) 可用 **<=，<，>=，>** 比较大小
  - 比较时采用“字典序”：从左到右逐一比较两个序列里的对应元素，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**

```
>>> [1, 2, 3] > [1, 2]
True
>>> [1, 2, 3] > [1, 2, 4]
False
>>> ('a', 'b', 'c') < ('b', 'b')
True
>>> (1, 2, 3) > ('a', 'b', 'c')
Traceback (most recent call last):
  File "<pyshell#100>", line 1, in <module>
    (1, 2, 3) > ('a', 'b', 'c')
TypeError: '>' not supported between instances of 'int' and 'str'
```

表操作 (1)

- 所有序列操作，包括变动操作，都可以用于表
- 对表 lst 使用带下标的操作时，下标不能超出 lst 的范围
  - 赋值：lst[i] = n，要求 $0 \le i < \text{len(lst)}$
  - 切片替换：lst[i:j] = t，要求 $0 \le i,\ j \le \text{len(lst)}$
    - 这里的 t 可以是任何可迭代对象 (序列或迭代器)
    - lst[0:0] = t, lst[len(lst):len(lst)] = t 分别表示在表头/尾加一段
- 表只有一个特殊操作
  - lst.sort()：按 < 关系对 lst 的元素排序 (调整 lst 各元素位置)
  - lst.sort(reverse=True)：将 lst 的元素按 < 的逆序排序
  - sort 方法还有一个 key 关键字参数，可以指定一个从元素计算出值的函数，要求将元素按这个函数的值排序 (见演示)

表操作 (2)

- 对比可用于表的操作：
  - 标准内置排序函数 **sorted**：可用于任何可迭代对象 **(iterable)**，返回一个元素排好序的表，其中是作为参数的可迭代对象里的元素
    - **sorted(lst)** 得到表 **lst** 的排序拷贝 (另一个新表)，**lst** 不变
    - **lst.sort()** 将 **lst** 的内容排序 (修改 **lst**，其中的元素重新排列)
  - 标准内置的序列反转函数 **reversed**：可用于序列对象 (注：不包括迭代器)，返回一个迭代器 (不是序列)，可以用于 **for** 头部
    - **reversed(lst)** 从表 **lst** 得到一个反向迭代器，**lst** 不变
    - **lst.reverse()** 把 **lst** 的内容反转，所有元素逐对对调位置
  - **lst[:]** 和 **lst.copy()** 都生成 **lst** 的拷贝表；**list(lst)** 的结果一样 (但属于做类型转换，实参是任何类型的可迭代对象)
- 序列操作的代价 (具体见演示)

打包和拆分 (1)

- **Python 支持打包 (packing) 和拆分 (unpacking) 操作**
  - 打包：把若干数据项包装成一个整体
  - 拆分：把一个包含几个元素的组合对象拆开之后再使用
- 赋值中的打包和拆分

```
>>> # 创建元组对象即是打包
>>> tp = 123, 'but', 567
>>>
>>> # 赋值时可以做元组的拆分
>>> a, b, c = tp  # 右边变量的数量须等于 tp 元素的个数
>>> print(a, b, c)
123 but 567
>>>
>>> # 各种序列对象都可以在赋值时拆分
>>> # 前提是赋值两边的元素结构相同
>>> a, b, c = [1, 2, 3]
>>> a, [b, c] = [1, (2, 3)]
>>> a, (b, c), d = [1, (2, 3), 4]
```

打包和拆分 (2)

- 在 **for** 语句头部使用拆分
  - 如果序列对象 (表等) 的元素都是二元组或两元表，则可以写：

```
for x, y in lst:                # 其中隐含拆分操作
        ... x ... y ...
# 循环迭代中 x 和 y 将依次取得 lst 中一个元素的两个成分
```

```
sum = 0.0
for price, quantity in invoice:
    sum += price * quantity
```

- 如果函数 **f(...)** 返回一个二元组或者二元表，则可以写

```
>>> x, y = f(...)
# x 和 y 分别关联到 f 返回值的首元素和次元素 (拆分赋值)
```

带 (单) 星号形参和元组

- 函数定义的形参列表，可以有一个前面加一个星号的形式参数

```
# 一个可对任意多个数值求和的函数
def mysum (*args) :
    s = 0
    for x in args : # args 是一个元组
        s += x
    return s
```

```
print(mysum(1, 2, 3, 4, 5))
# args 约束到 (1, 2, 3, 4, 5)

print(mysum())
# args 约束到空元组
```

  - 函数调用时，该形参约束到所有未得到匹配的普通实参构成的元组，默认情况是约束到空元组 (如果没有未匹配的实参)
  - 利用带单星号形参，可以定义允许任意多个实参的函数

- **Python 规定**：如果形参表里的某个普通形参带了默认值，位于其后直至带单星号形参的所有普通形参都必须带默认值
  - 带单星号形参的前后都可有普通形参 (可带/可不带默认值)

函数调用中的实参

- 函数调用时，必须用关键字实参给带单星号形参后面的普通形参提供实参
- 函数调用时，形参和实参的匹配规则：
  1. 普通/位置实参：按位置与形参一一匹配
  2. 关键字实参：按关键字与同名形参匹配
     - 规定：关键字实参只能出现在所有位置实参之后
  3. 没有得到实参、但有默认值的形参取其默认值为实参
  4. 匹配剩下的普通实参：做成一个元组约束到带星号形参
     - 如果没剩下相应种类的实参，带星号实参约束到空元组
     - 如果没有带星号形参，函数调用时出现未匹配的实参，则报参数个数错误 **(TypeError)**

函数形参和调用时实参的一些情况

```
def func1(a, *b, c=0, d=1):
    print(a, b, c, d)

func1(1, 2, 3, d=10)
func1(1)
```

```
1 (2, 3) 0 10
1 () 0 1
```

```
def func2(a, b=0, *c, d=11):
    print(a, b, c, d)

func2(1, 2, 3, 4, 5)
func2(1, 2, 3, d=10)
func2(1)
```

```
1 2 (3, 4, 5) 11
1 2 (3,) 10
1 0 () 11
```

自己分析和测试

```
def func3(a, *b, c, d=11):
    print(a, b, c, d)
func3(1, 2, 3, 4, c=5)
```

```
1 (2, 3, 4) 5 11
```

平行赋值时的拆分 (带单星号变量)

```
>>> a, b, *rest = range(5)
>>> print(f'{a}, {b}, {rest}')
0, 1, [2, 3, 4]
>>>
>>> a, b, *rest = range(3)
>>> print(f'{a}, {b}, {rest}')
0, 1, [2]
>>>
>>> a, b,
>>> print(
0, 1, []
```

```
>>> # 平行赋值时，左侧带单星号的变量可位于任意位置
>>> a, *body, c, d = range(5)
>>> print(f'{a}, {body}, {c}, {d}')
0, [1, 2], 3, 4
>>>
>>> *head, b, c, d = range(5)
>>> print(f'{head}, {b}, {c}, {d}')
[0, 1], 2, 3, 4
```

函数调用时的拆分实参

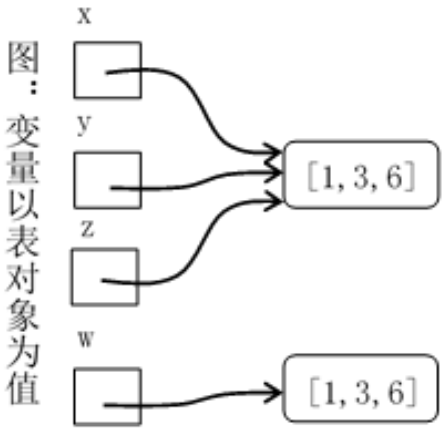
- 拆分实参 (unpacking argument) ： 一种特殊的实参形式
  - 形式：在实参表达式 (表/元组等可迭代对象) 前加一个星号 *
  - 功能：解释器将这个实参 (可迭代对象) 拆分打开，用其元素为函数提供若干个实际参数值
- 例：

```
>>> a = 1, 2, 3, 4, 5
>>> mysum(*a)                    # 结果为 15
                                 # 执行 mysum(a) 将出错

>>> b = [0, 20, 3]
>>> for i in range(*b): …        #  相当于 range(0, 20, 3)

>>> print(*[1, 2, 3], sep=', ', end=';')
1, 2, 3;                         # 按格式输出表元素的简洁写法
```

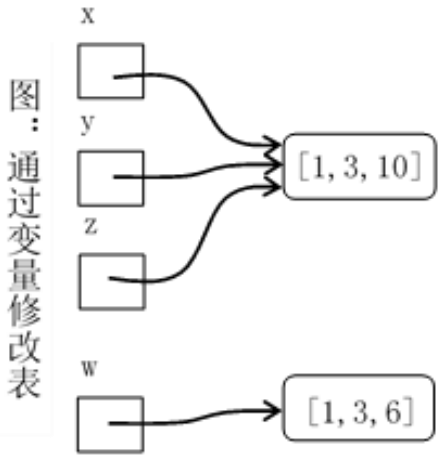
组合对象的共享

- **Python 中的赋值是建立变量与对象之间的关联关系 (引用式变量)**
  - 不同变量可以关联到同一个对象 ➡ 共享同一个值 (对象)

```
>>> x = y = [1, 3, 6]
>>> z = x      # x, y, z 共享同一个表对象
>>> w = [1, 3, 6] # w 关联另一个 (值相同的)
```




```
>>> x[2] = 10
>>> x
[1, 3, 10]
>>> y
[1, 3, 10]
>>> z
[1, 3, 10]
>>> w
[1, 3, 6]
```




```
>>> z.append(20)
>>> x
[1, 3, 10, 20]
>>> y
[1, 3, 10, 20]
```

组合对象的 "相等"

- 比较组合对象时，存在两种“相等”概念
  1. 两个对象“内容”是否相同 (“值”相等)
  2. 是否为同一个对象
- 运算符 == 和 !=：(使用频率更高)
  - 判断两个对象的“内容”是否相同
  - 用 == 判断的相等称为“按结构相等”
    - 两个组合对象相等，当且仅当其结构相同，而且元素相等
    - 可用来比较任意对象，当被比较对象类型不同且不能转换时结果为 **False**
- 运算符 is 和 is not：判断是否为同一个对象

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> c = a
>>> a == b
True
>>> a == c
True
>>> a is b
False
>>> a is c
True
```

```
>>> id(a)
62172936
>>> id(b)
62131912
>>> id(c)
62172936
```

is / is not 运算符

- **is / is not 可用于比较任何两个对象，比较的是对象的标识**
  - 总能得到 **True/False** 结果
  - 不涉及内容的比较，效率高且绝对不出错
  - 内置函数 **id** 取得对象标识，每个对象在生命周期中有唯一且不变的标识 **(identity)**

Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a "stored program computer", code is also represented by objects.)

Every object has an identity, a type and a value. An object's *identity* never changes once it has been created; you may think of it as the object's address in memory. The 'is' operator compares the identity of two objects; the `id()` function returns an integer representing its identity.

CPython implementation detail: For CPython, `id(x)` is the memory address where `x` is stored.

https://docs.python.org/3.8/reference/datamodel.html

is / is not 运算符

- **is / is not 可用于比较任何两个对象，比较的是对象的标识**
  - 总能得到 **True/False** 结果
  - 不涉及内容的比较，效率高且绝对不出错
  - 内置函数 **id** 取得对象标识，每个对象在生命周期中有唯一且不变的标识 **(identity)**

```
# 推荐用法：变量和常量值之间的比较
# 例如，检查变量的值是不是 None
if x is None:
    ...

if x is not None:
    ...
```

组合对象的共享和拷贝

```
>>> # Note: 拷贝对象得到另一个独立的对象
>>> t = x.copy()  # 也可以用 t = x[:]
>>> t is x
False
>>> t == x
True
>>> x.append(100)
>>> t == x
False
```

```
>>> a = b = []
>>> a is b
True
>>> # a、b 共享同一个空表对象
```

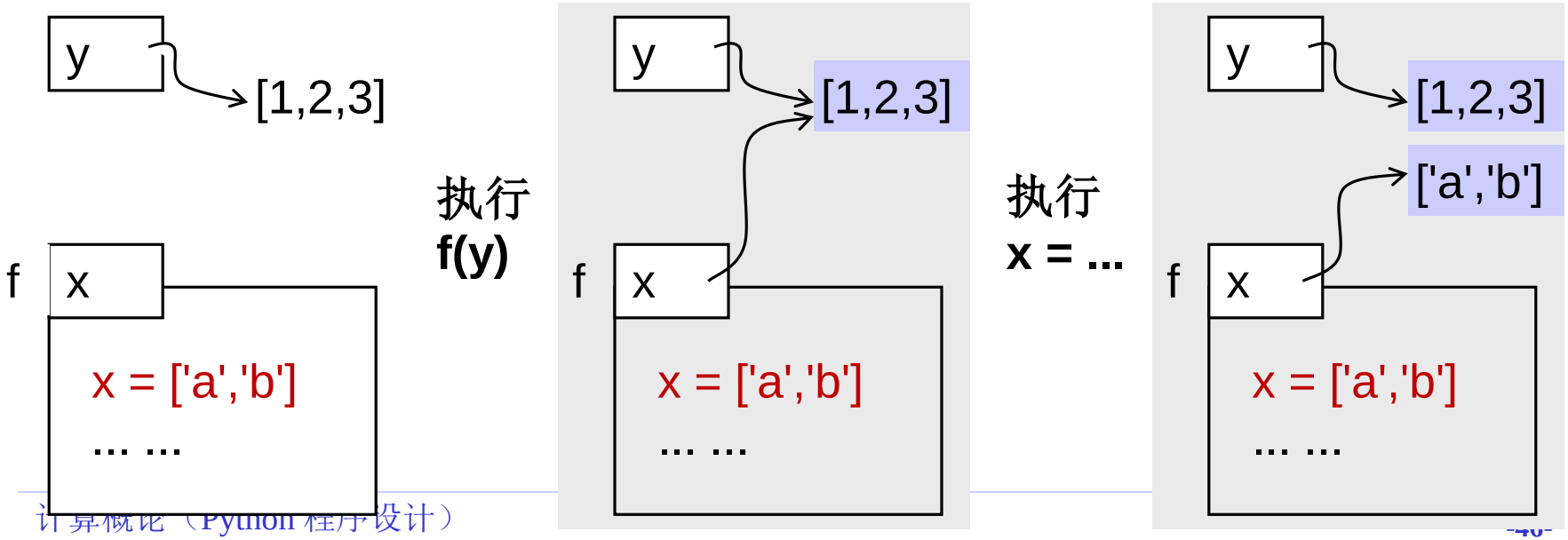
```
>>> c, d = [], []
>>> c is d
False
>>> c == d
True
>>> # c、d 关联到不同的空表对象
```

组合对象的拷贝和变动性

- 设 s 是一个序列对象
  - **s.copy()**：只用于可变对象
  - **s[:]**：全切片操作创建 s 的拷贝
  - **s * n**：s 的 n 个拷贝的拼接
  - **list(s)**：创建一个 s 内容的拷贝表
- 用 copy 或者全切片等操作拷贝一个组合对象时，实际上只做了最外层结构的拷贝，而其中元素仍然是与原对象共享
  - 称之为浅拷贝 (shallow copy) — Python 的默认拷贝方式
  - 当原对象的元素有可变对象时，则修改原对象/拷贝对象中的可变元素，将影响另外一个对象的内容
  - (代码演示)

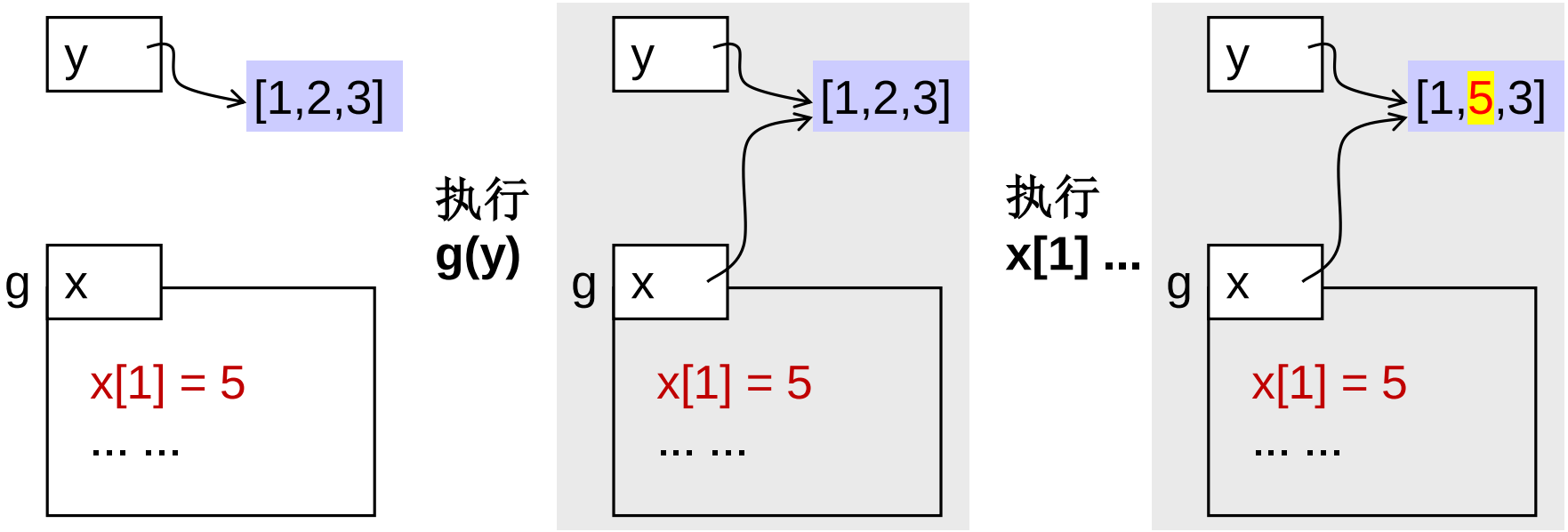
函数的表参数与共享 (1)

- 对于函数调用时的参数传递，**Python 只支持共享传参 (call by sharing)**
  - 实际执行函数体之前，各个形式参数关联到相应的实参对象 (即，各形参是相应实参对象在函数内部的别名)
- 当函数某个形参关联的实参是一个表对象，如果函数体里
  - 对该形参做赋值，则改变其约束关系，但不会改变实参表



函数的表参数与共享 (2)

- 当函数某个形参关联的实参是一个表对象，如果函数体里
  - 通过该形参执行修改表的有关操作，则会实际修改所关联 (实参) 表的内容
  - 原因：形参和实参关联/共享了同一个表对象 (可变对象)



函数的表参数与共享 (2)

- 当函数某个形参关联的实参是一个表对象，如果函数体里
  - 通过该形参执行修改表的有关操作，则会实际修改所关联 (实参) 表的内容
  - 原因：形参和实参关联/共享了同一个表对象 (可变对象)

➔ 当函数接受可变参数时，应谨慎考虑是否需要把函数内部的修改/操作效果作用到实参可变对象 (如果不，应该先建立拷贝)

- 参数的默认值
  - **Python** 规定：解释器在处理函数定义时，对参数列表里的默认值表达式逐个求值 (且只求值一次)，并记录这些值
    - 在函数调用时，有需要则使用所记录的值
  - 如果参数的默认值是可变对象，则可能引发共享问题 (演示)

➔ 应避免使用可变对象作为函数参数的默认值！