Python程序设计

列表 - 进阶应用

刘安
苏州大学，计算机科学与技术学院

http://web.suda.edu.cn/anliu/

本节涉及到的知识点

- 列表遍历
  - enumerate和zip函数
- 列表复制
- 列表嵌套
- 列表排序

通过for循环遍历列表

- 列表也是一种可迭代对象：iter(L)函数获得相应的迭代器I，每执行next(I)函数一次获得一个值

- for循环可以自动实现迭代过程，每次访问列表的一个元素

```
>>> L = [1, 2, 3]
>>> I = iter(L)
>>> next(I)
1
>>> next(I)
2
>>> next(I)
3
>>> next(I)
Traceback (most recent call last):
  File "<pyshell#337>", line 1, in <module>
    next(I)
StopIteration
```

```
>>> for x in L:
        print(x)


1
2
3
```

循环变量与列表元素的关系

- 执行语句for x in L会把列表的元素依次赋值给循环变量x，假设当前赋值的元素是y
  - 如果y不可变，x获得y的副本，对x的修改不会影响y的值
  - 否则，x绑定到y，对x的修改就是对y的修改

```
>>> A = [1, 2, 3]
>>> for x in A:
        x = x + 1
```

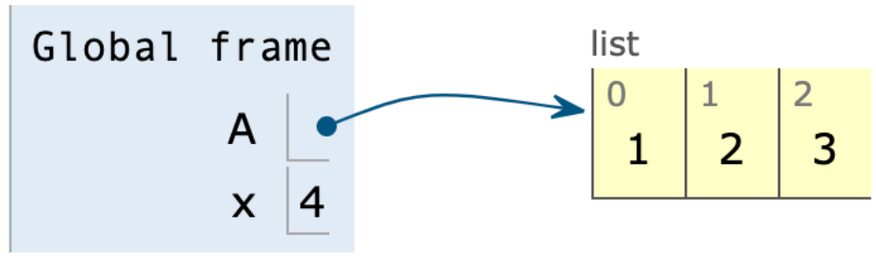
```
>>> A
[1, 2, 3]
```

```
>>> B = [[1], [2], [3]]
>>> for x in B:
        x.append(1)
```

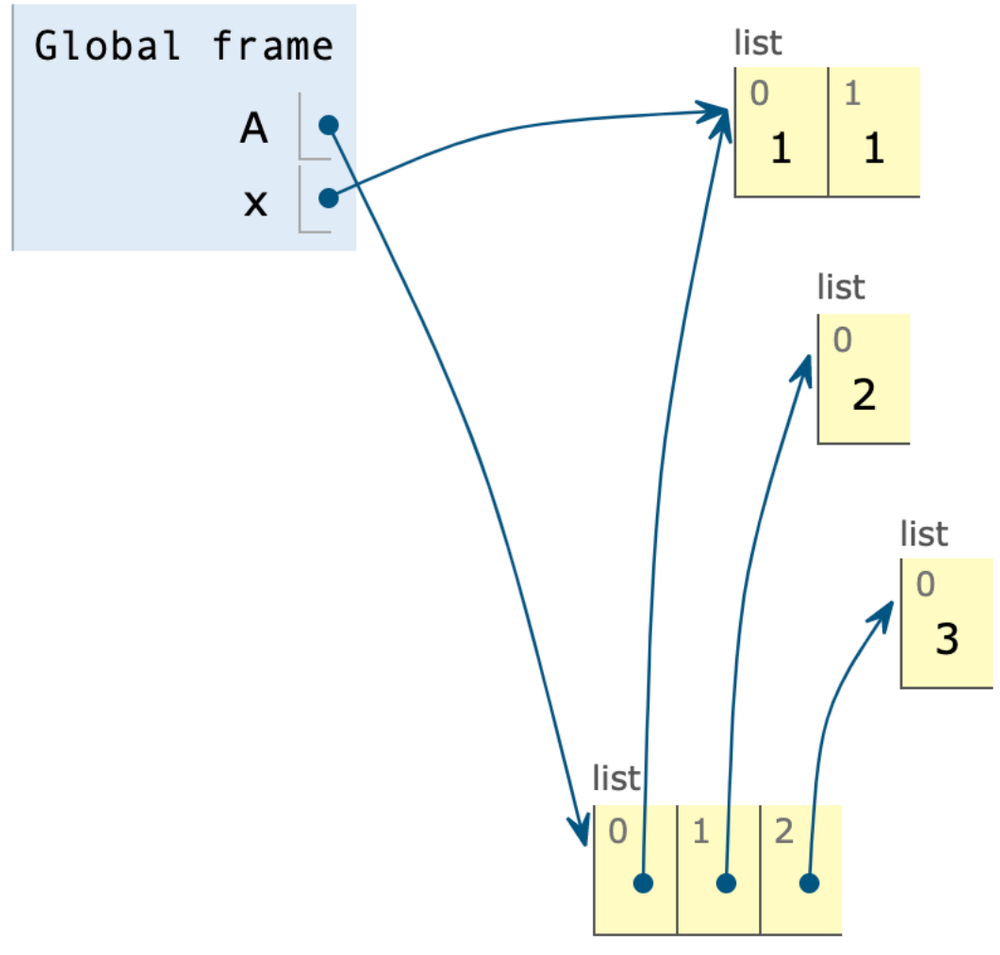
```
>>> B
[[1, 1], [2, 1], [3, 1]]
```

循环变量与列表元素的关系

```
1  A = [1, 2, 3]
2  for x in A:
3      x = x + 1
```




```
1  A = [[1], [2], [3]]
2  for x in A:
3      x.append(1)
```



line that has just executed

next line to execute

使用索引修改列表

- 对于不可变元素，循环变量获取的是其副本
- 如果想修改列表中元素的值，必须借助索引
- 使用一个变量来维护索引值，通常用range和len组合构造所有有效的索引

```
>>> L = [1, 2, 3]
>>> for i in range(len(L)): #将列表每个元素值加1
        L[i] += 1


>>> L
[2, 3, 4]
```

enumerate函数

- enumerate(iterable [,start])：返回一个可迭代对象，每次迭代返回一个(count, value)元组，其中count从start开始（如果没有指定，使用默认值0），value从iterable中的第0个元素开始

```
>>> L = [1, 2, 3]
>>> list(enumerate(L))
[(0, 1), (1, 2), (2, 3)]
>>>
>>> list(enumerate(L, 10))
[(10, 1), (11, 2), (12, 3)]
>>>
>>> list(enumerate('abc'))
[(0, 'a'), (1, 'b'), (2, 'c')]
```

enumerate函数在for循环中的应用

- 使用enumerate函数可以在for循环中同时记录元素及其索引

```
>>> L = [1, 2, 3]
>>> for t in enumerate(L):
        print(t)
```

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

```
>>> for i, x in enumerate(L):
        print(i, x, L[i])
```

```
0 1 1
1 2 2
2 3 3
```

右侧的for循环中实际上使用了序列赋值：i, x = t
这里t是一个含有2个元素的元组

zip函数

- zip(*iterables)：返回一个可迭代对象，每次迭代返回一个元组，元组的第i个元素来自参数iterables中的第i个可迭代对象，迭代在最短可迭代对象元素用尽的情况下结束

```
>>> list(zip([1, 2, 3], [4, 5, 6]))
[(1, 4), (2, 5), (3, 6)]
>>>
>>> list(zip(range(3), 'abc', [1, 2, 3]))
[(0, 'a', 1), (1, 'b', 2), (2, 'c', 3)]
>>>
>>> list(zip('abcde', '123'))
[('a', '1'), ('b', '2'), ('c', '3')]
```

zip函数在for循环中的应用

- 通过zip函数，可以在一个for循环中同时遍历多个列表

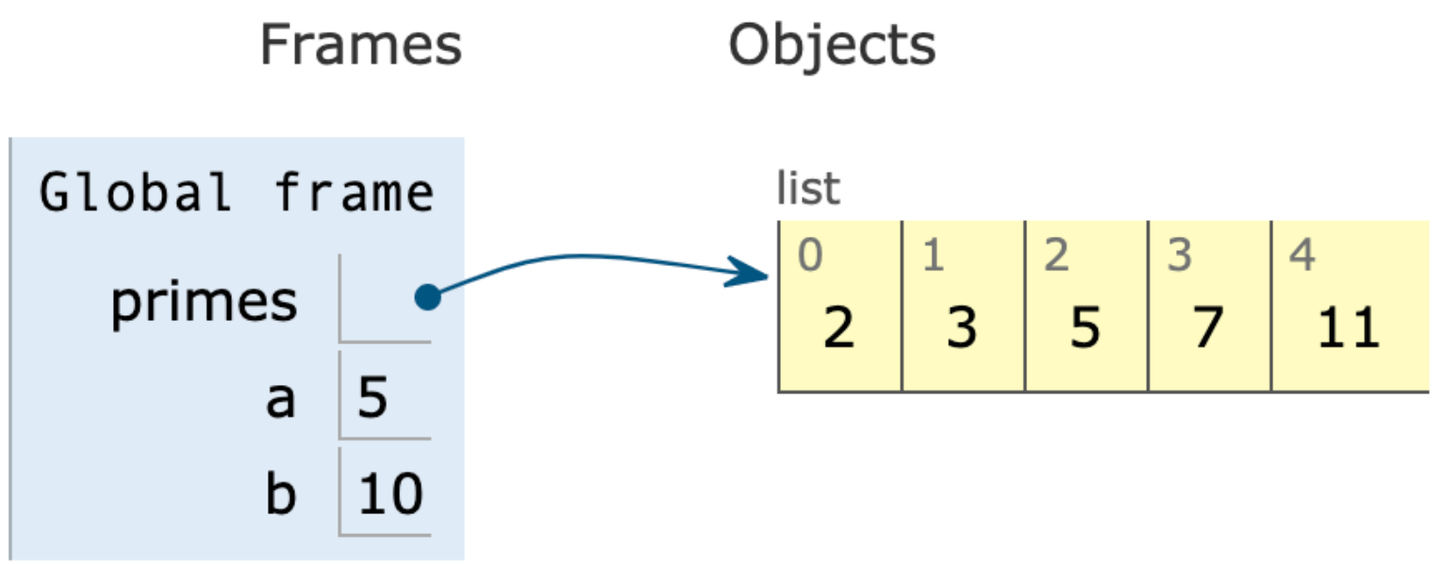
```
>>> names = ['Ada', 'Bob', 'Cal']
>>> prog = [90, 75, 80]
>>> alg = [85, 80, 90]
>>>
>>> for name, *score in zip(names, prog, alg):
        print(name, sum(score))


Ada 175
Bob 155
Cal 170
```

for循环中使用了序列解包，赋值后score是一个列表，包含两个元素

元素查找

- 编写一个函数，接受一个列表L和一个对象t，查找t是否在列表L中，如果在，返回t的索引，否则返回-1

```
>>> primes = [2, 3, 5, 7, 11]
>>> a = 5
>>> b = 10
>>> seq_search(primes, a)
2
>>> seq_search(primes, b)
-1
```



元素查找

- 编写一个函数，接受一个列表L和一个对象t，查找t是否在列表L中，如果在，返回t的索引，否则返回-1

```
1 def seq_search(L, t):
2     for i, e in enumerate(L):
3         if e == t:
4             return i
5     return -1
```

- 如果t和e相等，说明t在列表中，这时i恰好是t的索引，返回i
- 如果t不在L中，那么对于每一个e，条件e==t都为假，语句return i不会执行
- 当遍历L中所有元素之后，程序执行到第5行，这时返回-1表示t不在L中

元素查找

- 注意参数类型

```
def seq_search(L, t):
    for i, e in enumerate(L):
        if e == t:
            return i
    return -1
```

这里并没有指定L中元素的类型以及对象t的类型，Python就是这么强大 👍

```
>>> mixed = [3.14, 'hi', 100, [1, 2]]
>>> seq_search(mixed, 3.14)
0
>>> seq_search(mixed, 'hi')
1
>>> seq_search(mixed, [1, 2])
3
>>> seq_search(mixed, 1000)
-1
```

寻找插入点

- 编写一个函数，接受一个升序列表L和一个元素t，返回索引idx，使得将元素t添加至L[idx]上，列表L仍保持升序。注意列表有可能为空

```
def find_insert_index(L, t):
    for i, x in enumerate(L):
        if x > t:
            return i
    # t should be inserted at the end of L
    return len(L) # i+1 is also okay
```

bisect模块中的bisect函数利用二分查找在有序序列中搜索待插入的位置
该模块还提供了insort函数将一个元素插入到有序序列中，并保持序列的有序性
更多内容参见：https://docs.python.org/3/library/bisect.html

插入排序

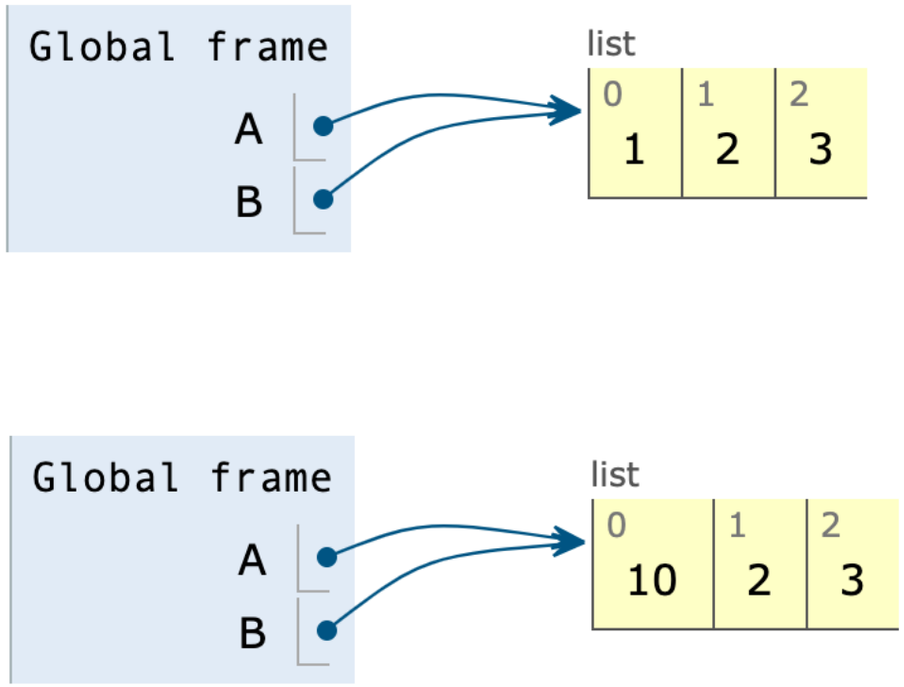
- 创建一个长度为10的列表A，每个元素是一个在[0,100]之间的随机数。然后使用find_insert_index函数和insert方法对A排序，结果放在一个新的列表L中

```
>>> import random
>>> A = [random.randint(0, 100) for i in range(10)]
>>> A
[52, 48, 75, 79, 55, 17, 4, 66, 13, 16]
>>> L = []
>>> for x in A:
        i = find_insert_index(L, x)
        L.insert(i, x)


>>> L
[4, 13, 16, 17, 48, 52, 55, 66, 75, 79]
```

列表复制 - 避免引用的副作用

- 赋值运算符是将变量绑定到对象
- 列表是可变对象
- 通过一个引用修改某列表时，别的引用也能看见这种变化

```
>>> A = [1, 2, 3]
>>> B = A
>>> B[0] = 10
>>> B
[10, 2, 3]
>>> A
[10, 2, 3]
```

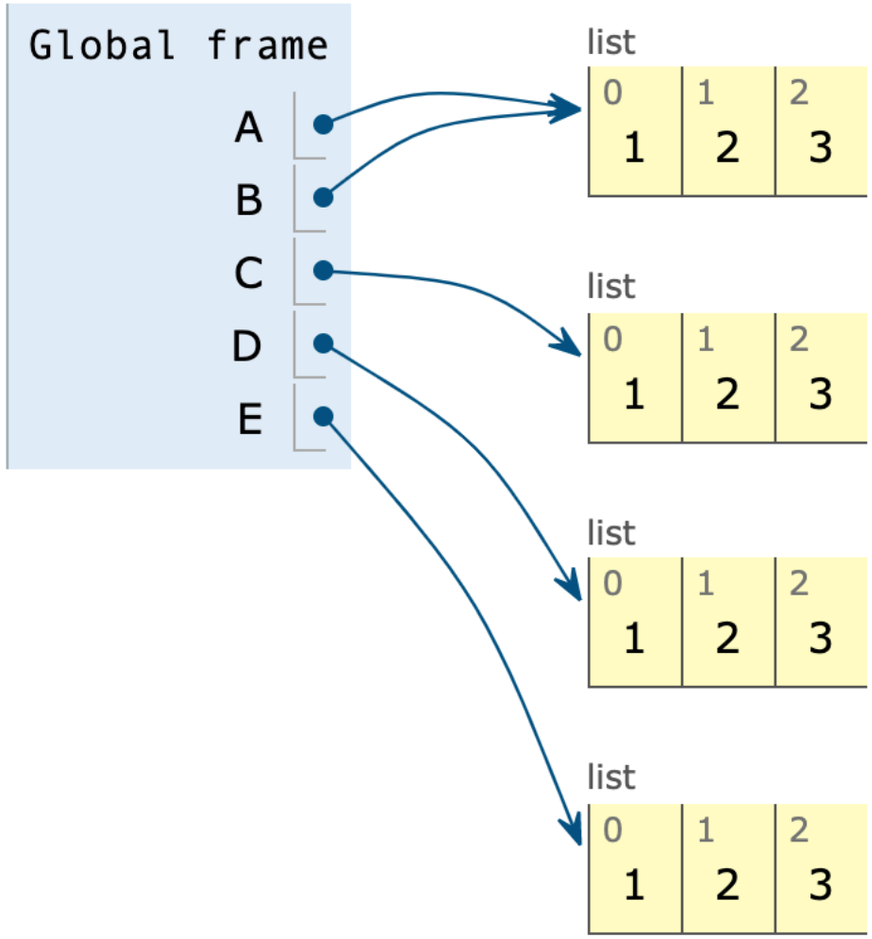


假设修改B影响A不是期望的结果，这时需要显式的复制操作

复制列表的方法

- list函数、分片操作、copy方法

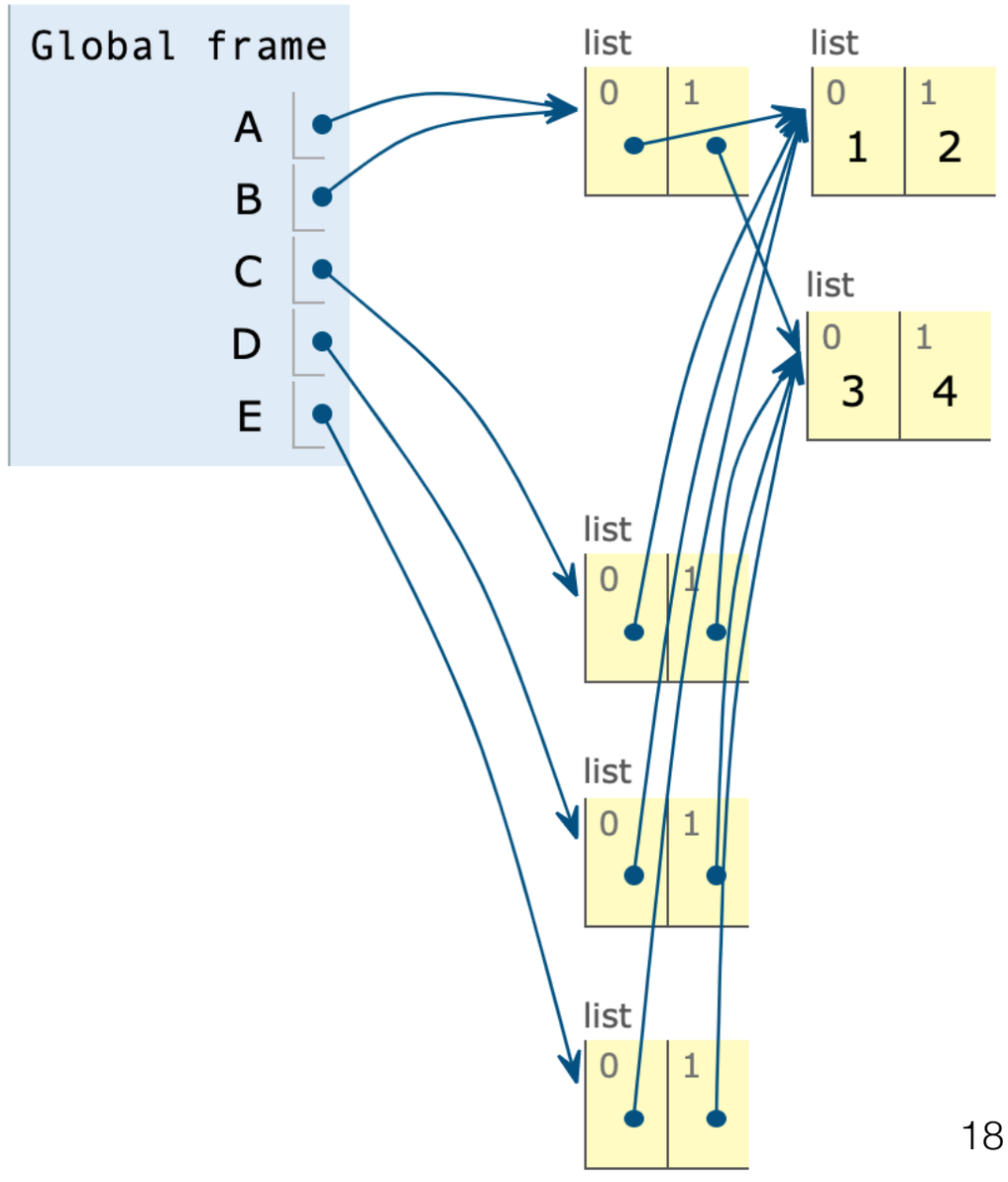
```
>>> A = [1, 2, 3]
>>> B = A
>>> C = list(A)
>>> D = A[:]
>>> E = A.copy()
>>>
>>> B[0] = 10
>>> A
[10, 2, 3]
>>> C
[1, 2, 3]
```



嵌套列表的复制

- 列表的元素还可以是对列表的引用
- list函数、分片、copy方法只能实现浅拷贝

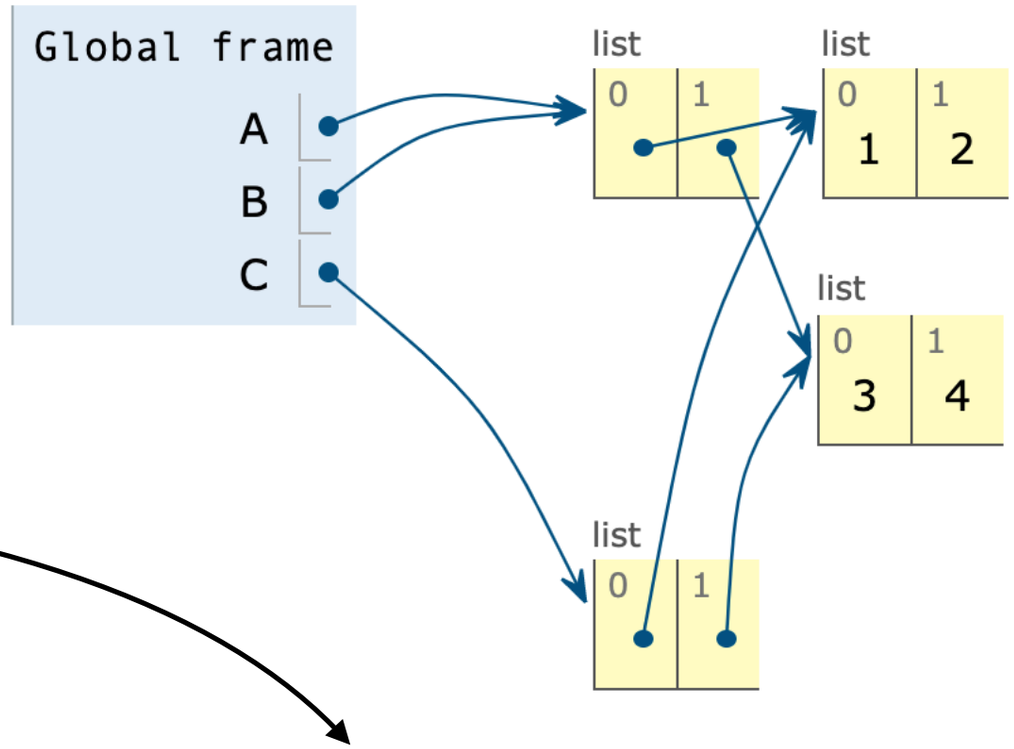
```
>>> A = [[1, 2], [3, 4]]
>>> B = A
>>> C = list(A)
>>> D = A[:]
>>> E = A.copy()
```

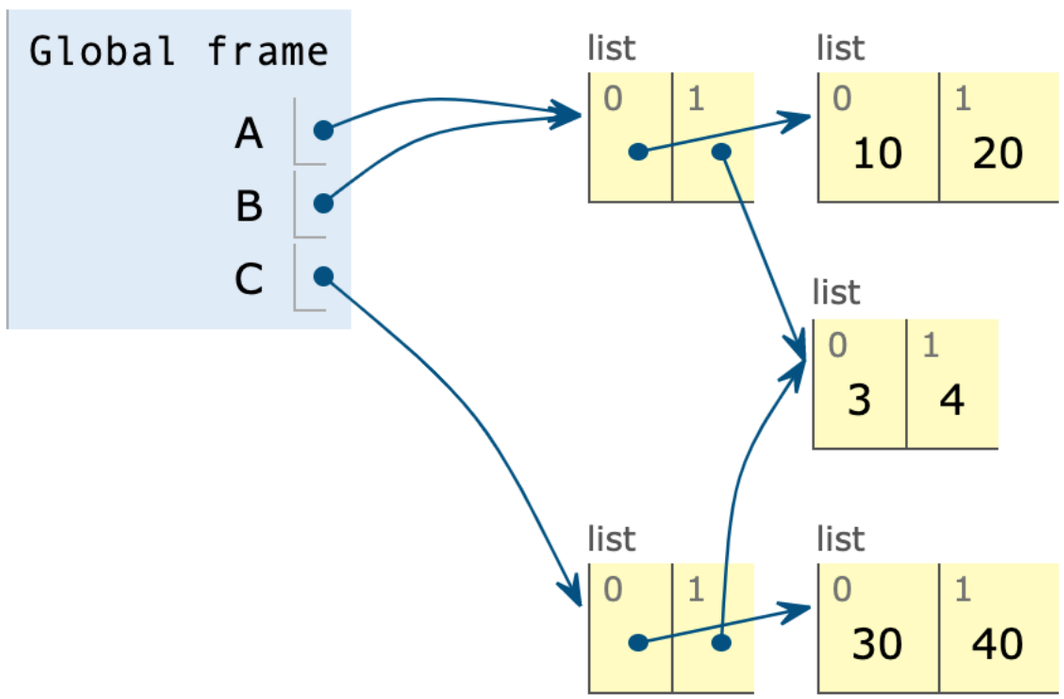


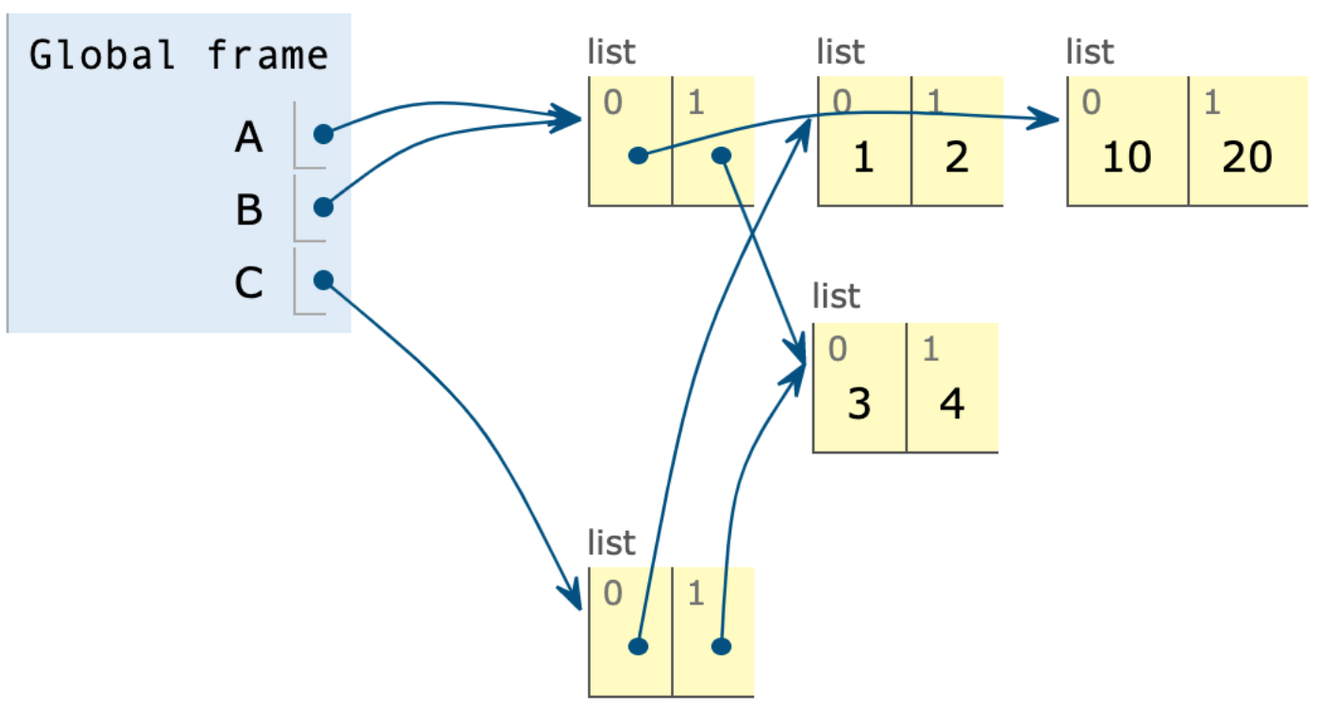
嵌套列表的复制

- 浅拷贝 - 只复制列表的顶层元素

```
>>> A = [[1, 2], [3, 4]]
>>> B = A
>>> C = A[:]
>>> B[0] = [10, 20]
>>> C[0] = [30, 40]
```



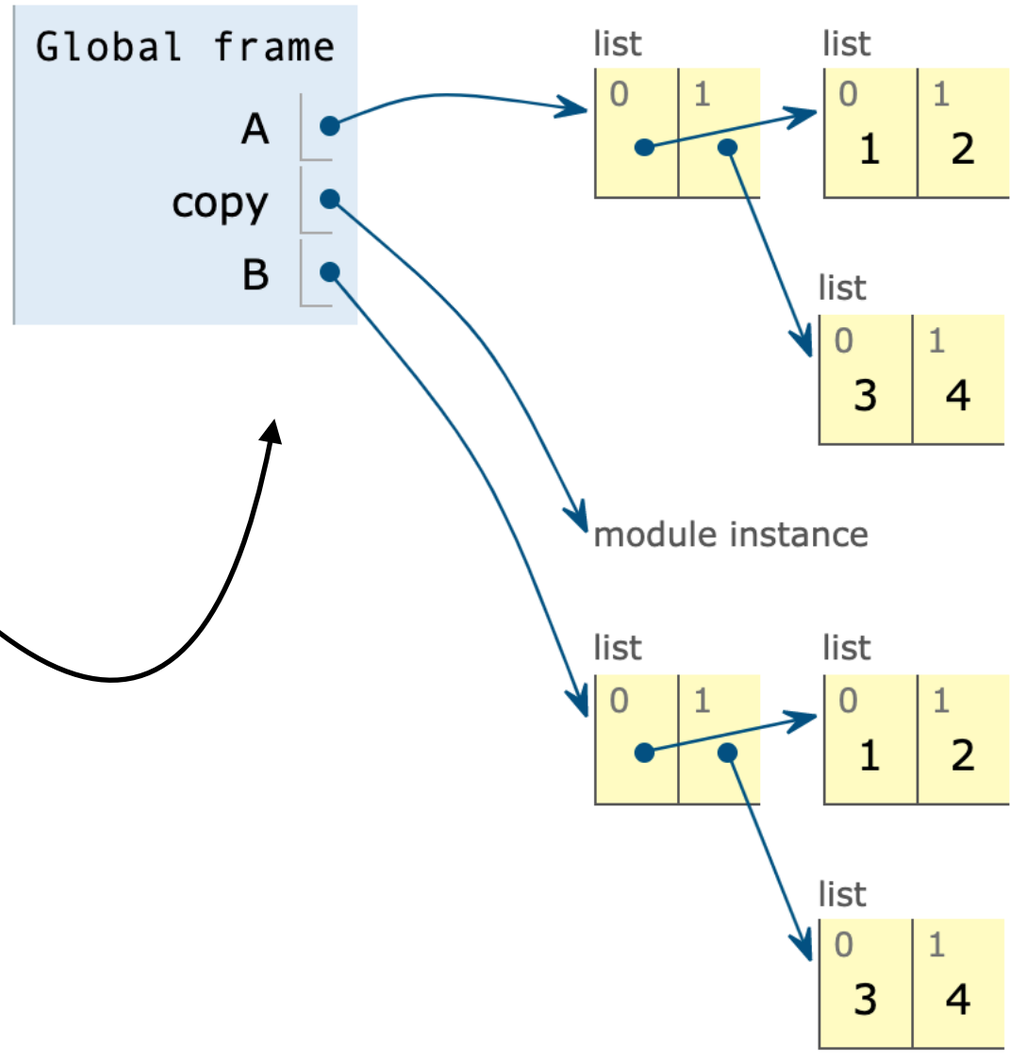




通过copy模块实现深拷贝

- 深拷贝的基本思想 - 递归的遍历对象来复制所有的元素
- copy模块中的deepcopy方法

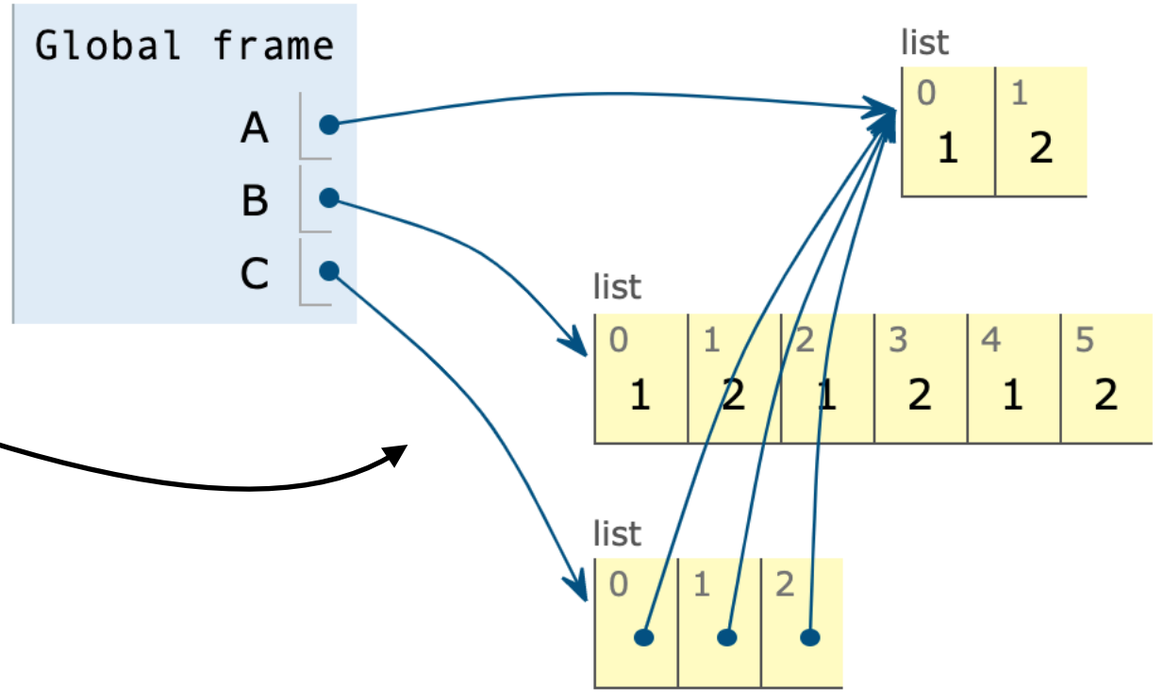
```
>>> import copy
>>> A = [[1, 2], [3, 4]]
>>> B = copy.deepcopy(A)
>>> B[0] = [10, 20]
>>> B
[[10, 20], [3, 4]]
>>> A
[[1, 2], [3, 4]]
```



注意重复操作*导致的引用

- 重复操作*也会导致共享引用
- 如有必要，使用之前介绍的复制方法来避免共享引用

```
>>> A = [1, 2]
>>> B = A * 3
>>> C = [A] * 3
>>> B
[1, 2, 1, 2, 1, 2]
>>> C
[[1, 2], [1, 2], [1, 2]]
>>> B[0] = 10
>>> C[0][0] = 100
>>> B
[10, 2, 1, 2, 1, 2]
>>> C
[[100, 2], [100, 2], [100, 2]]
```



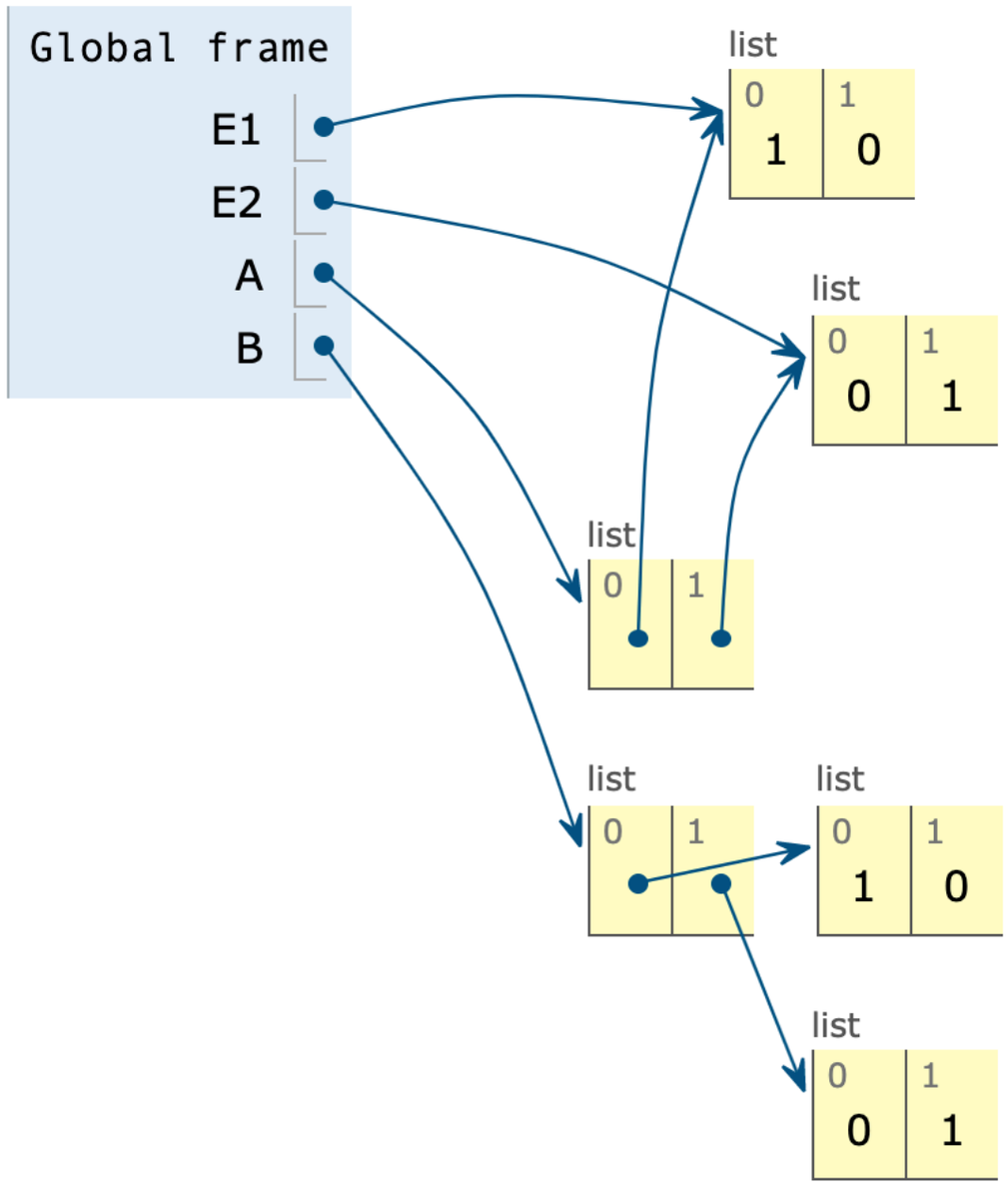
列表嵌套

- 列表的元素还是列表，称为嵌套列表
- 列表支持任意层次的嵌套
- 主要用来描述复杂的数据结构，比如矩阵
- 在处理时一定要注意列表引用带来的副作用
    - 初始化嵌套列表时避免共享引用

使用嵌套列表表示矩阵

- $A = [a_{ij}]_{m \times n}$
- $A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, M = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$

```
>>> E1 = [1, 0]
>>> E2 = [0, 1]
>>> A = [E1, E2]
>>> B = [list(E1), E2[:]]
>>> A
[[1, 0], [0, 1]]
>>> B
[[1, 0], [0, 1]]
>>> A[0]
[1, 0]
>>> B[1][1]
1
```



矩阵相加

- 编写一个函数，接受两个矩阵，返回它们的和
  - $A + B = [a_{ij} + b_{ij}]_{m \times n}$ $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 2 \\ 3 & 5 \end{bmatrix}$

```
1 def add_matrix(A, B):
2     m = len(A)
3     n = len(A[0])
4     C = [[None] * n for i in range(m)]
5     for i in range(m):
6         for j in range(n):
7             C[i][j] = A[i][j] + B[i][j]
8     return C
```

矩阵相乘

- 编写一个函数，接受两个矩阵，返回它们的乘积
  - $[a_{ij}]_{m\times n} \times [b_{ij}]_{n\times r} = [c_{ij}]_{m\times r}$ $\quad \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} \times \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} = \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}$
  - $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$

```python
def multiply_matrix(A, B):
    m = len(A)
    n = len(A[0])
    r = len(B[0])
    C = [[0] * r for i in range(m)]
    for i in range(m):
        for j in range(r):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
    return C
```

✍️ 这里可以是None吗?

矩阵转置

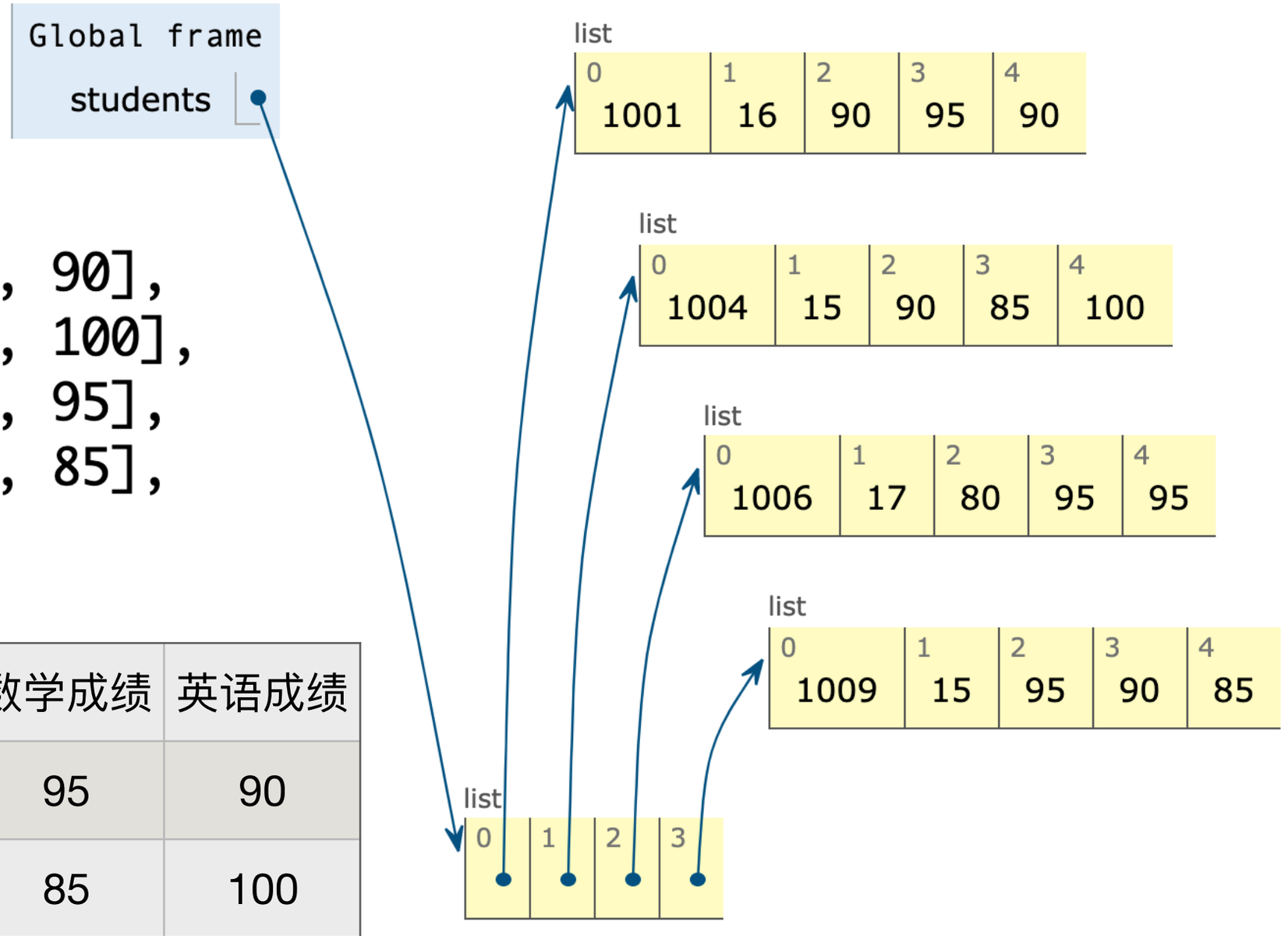
- 编写一个函数，接受一个矩阵，返回它的转置
  - m行n列的矩阵A的转置A^T是一个n行m列的矩阵，其中A^T的第i列就是A的第i行 $A = \begin{bmatrix} 1 & 3 & 2 \\ 5 & 0 & 1 \end{bmatrix}$ $A^T = \begin{bmatrix} 1 & 5 \\ 3 & 0 \\ 2 & 1 \end{bmatrix}$

```python
def transpose_matrix(A):
    m = len(A)
    n = len(A[0])
    B = [[None] * m for k in range(n)]
    for i in range(n):
        for j in range(m):
            B[i][j] = A[j][i]
    return B
```

使用嵌套列表存储复杂数据

```
>>> students = [
    [1001, 16, 90, 95, 90],
    [1004, 15, 90, 85, 100],
    [1006, 17, 80, 95, 95],
    [1009, 15, 95, 90, 85],
    ]
```

学号	年龄	语文成绩	数学成绩	英语成绩
1001	16	90	95	90
1004	15	90	85	100
1006	17	80	95	95
1009	15	95	90	85



列表排序

- L.sort()：将列表L中的元素原地排序，即修改了L的内容
  - 排序依据是列表中任意两个对象之间定义了<关系
  - 原地排序节省存储空间，适用于非常大的列表
    - 该方法返回None，不是排序后的列表
- sorted(iterable)：返回一个有序列表，其中的元素来自可迭代对象iterable
  - 不仅适用于列表，也适用不可变的可迭代对象比如元组和字符串（这些对象没有sort方法）

列表排序

```
>>> L = [random.randint(0, 100) for i in range(8)]
>>> L
[66, 81, 80, 32, 84, 24, 99, 26]
>>> L.sort() # 默认从小到大排序，即升序排列
>>> L
[24, 26, 32, 66, 80, 81, 84, 99]
>>> L = L.sort() # 常见的错误
>>> print(L) # 上述赋值让L绑定到None对象
None
>>> L = [random.randint(0, 100) for i in range(8)]
>>> sorted(L) # 默认从小到大排序，即升序排列
[5, 14, 31, 33, 38, 68, 92, 99]
>>> L # 列表L的值没有改变
[33, 68, 99, 5, 31, 92, 14, 38]
```

通过reverse参数控制升序和降序

```
>>> L = [random.randint(0,100) for i in range(8)]
>>> L
[43, 34, 65, 16, 45, 55, 79, 87]
>>> sorted(L, reverse = True)
[87, 79, 65, 55, 45, 43, 34, 16]
>>> sorted(L, reverse = False)
[16, 34, 43, 45, 55, 65, 79, 87]
```

reverse参数的值是True，
那么元素按照从大到小排序

```
>>> L.sort(reverse = True)
>>> L
[87, 79, 65, 55, 45, 43, 34, 16]
>>> L.sort(reverse = False)
>>> L
[16, 34, 43, 45, 55, 65, 79, 87]
```

reverse参数的值是FALSE，
那么元素按照从小到大排序

如果没有指定reverse参数的值，那么默认按照从小到大排序

对象的大小关系

- 回顾：可以通过sort方法和sorted函数对列表进行排序

```
>>> L = [5, 3, 7, 1, 9]
>>> sorted(L)
[1, 3, 5, 7, 9]
>>> L = ['p', 'y', 't', 'h', 'o', 'n']
>>> sorted(L)
['h', 'n', 'o', 'p', 't', 'y']
```

- 排序的关键：知道如何判定两个对象的大小关系
- 对于嵌套列表students
  - 如何判断列表的大小关系

```
>>> students = [
    [1001, 16, 90, 95, 90],
    [1004, 15, 90, 85, 100],
    [1006, 17, 80, 95, 95],
    [1009, 15, 95, 90, 85],
    ]
```

列表的大小关系

- 从左至右，逐个元素比较，直至遇到不相等的元素
- 对于列表A和B,
  - $A = B$，如果$A[i] = B[i]\ (0 \le i \le n-1)$
  - $A < B$，如果$\exists k, A[k] < B[k], A[i] = B[i]\ (0 \le i \le k-1)$
  - $A > B$，如果$\exists k, A[k] > B[k], A[i] = B[i]\ (0 \le i \le k-1)$

```
>>> [1, 2, 3, 4] < [1, 2, 4, 5]
True
>>> [1, 2, 3, 4] > [1, 2, 2, 4, 5]
True
```

列表的大小关系

- 对于列表A和B,
  - $A = B$, 如果$A[i] = B[i]\ (0 \le i \le n-1)$
  - $A < B$, 如果$\exists k, A[k] < B[k], A[i] = B[i]\ (0 \le i \le k-1)$
  - $A > B$, 如果$\exists k, A[k] > B[k], A[i] = B[i]\ (0 \le i \le k-1)$

```
>>> students
[[1001, 16, 90, 95, 90], [1004, 15, 90, 85, 100],
[1006, 17, 80, 95, 95], [1009, 15, 95, 90, 85]]
>>> sorted(students)
[[1001, 16, 90, 95, 90], [1004, 15, 90, 85, 100],
[1006, 17, 80, 95, 95], [1009, 15, 95, 90, 85]]
```

如何根据学生的年龄排序呢?

单关键字排序

- sort方法和sorted函数都接受一个key参数，其值是一个函数对象，默认值为None
- 该函数只有一个参数，通常是待比较的对象
- 该函数只有一个返回值，即对象在指定关键字上的值，根据该值确定对象的大小关系

```
1 def sort_helper(s):
2     return s[1]
```

- 参数s是一个列表，代表一个学生的信息，比如[1001，16，90，95，90]。返回值是s[1]，列表的第1个元素存储学生的年龄，所以这里根据学生的年龄来决定列表的大小

单关键字排序

- 根据学生的年龄升序排序

```
>>> def sort_helper(s):
        return s[1]

>>> sorted(students, key = sort_helper)
[[1004, 15, 90, 85, 100], [1009, 15, 95, 90, 85],
[1001, 16, 90, 95, 90], [1006, 17, 80, 95, 95]]
```

- 根据学生的语文成绩升序排序

```
>>> def sort_helper(s):
        return s[2]

>>> sorted(students, key = sort_helper)
[[1006, 17, 80, 95, 95], [1001, 16, 90, 95, 90],
[1004, 15, 90, 85, 100], [1009, 15, 95, 90, 85]]
```

排序的稳定性

- 具有相同关键字的对象在排序前后的相对顺序不变

```
>>> students
[[1001, 16, 90, 95, 90], [1004, 15, 90, 85, 100],
[1006, 17, 80, 95, 95], [1009, 15, 95, 90, 85]]
```

```
>>> def sort_helper(s):
        return s[1]
```

学号1004的学生和学号1009的学生年龄相同
排序前后1004号都在1009号之前

```
>>> sorted(students, key = sort_helper)
[[1004, 15, 90, 85, 100], [1009, 15, 95, 90, 85],
[1001, 16, 90, 95, 90], [1006, 17, 80, 95, 95]]
```

```
>>> def sort_helper(s):
        return s[2]
```

学号1001的学生和学号1004的学生语文成绩相同
排序前后1001号都在1004号之前

```
>>> sorted(students, key = sort_helper)
[[1006, 17, 80, 95, 95], [1001, 16, 90, 95, 90],
[1004, 15, 90, 85, 100], [1009, 15, 95, 90, 85]]
```

多关键字排序

- 对象之间的大小关系由多个具有主次关系的关键字决定
  - 首先比较关键字K_1，如果相等，再比较关键字K_2，如果相等，继续比较下一个关键字，直至最后一个关键字K_n
- 多关键字排序的关键在于利用排序的稳定性
  - 先根据最次关键字K_n排序，然后再根据次关键字K_{n-1}排序，最后根据主关键字K_1排序
  - 根据主关键字排序的时候，不会改变次关键字决定的顺序
  - Python提供的sort方法和sorted函数都是稳定的排序算法

多关键字排序

- 根据学生的语文成绩降序排序，如果成绩相同，根据年龄升序排序

- 首先根据次关键字年龄进行升序排序

```
>>> L = sorted(students, key = sort_helper_age)
>>> L
[[1004, 15, 90, 85, 100], [1009, 15, 95, 90, 85],
[1001, 16, 90, 95, 90], [1006, 17, 80, 95, 95]]
```

- 然后根据主关键字语文成绩进行降序排序

```
>>> L = sorted(L, key = sort_helper_chn, reverse =
True)
>>> L
[[1009, 15, 95, 90, 85], [1004, 15, 90, 85, 100],
[1001, 16, 90, 95, 90], [1006, 17, 80, 95, 95]]
```

学生排序

- 编写一个函数，接受一个存储多个学生信息的嵌套列表，返回排序后的列表。排序规则是：首先根据学生的总成绩降序排序，如果总成绩相同，根据语文成绩降序排序，如果语文成绩还相同，根据年龄升序排序

```
>>> students
[[1001, 16, 90, 95, 90], [1004, 15, 90, 85, 100],
[1006, 17, 80, 95, 95], [1009, 15, 95, 90, 85]]
>>> L = sort_students(students)
>>> L
[[1004, 15, 90, 85, 100], [1001, 16, 90, 95, 90],
[1009, 15, 95, 90, 85], [1006, 17, 80, 95, 95]]
```

学生排序

```
def sort_helper_age(s):
    return s[1]

def sort_helper_chn(s):
    return s[2]

def sort_helper_total(s):
    return s[2] + s[3] + s[4]

def sort_students(L):
    L.sort(key = sort_helper_age)
    L.sort(key = sort_helper_chn, reverse = True)
    L.sort(key = sort_helper_total, reverse = True)
    return L
```