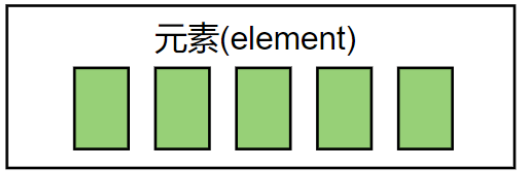
有序对象

主要内容

- 有序对象(通用方法）：字符串、range对象、列表和元组
- 可变有序对象：列表
- 有序对象的切片
- 序列解包
- 用于序列的常用内置函数：zip和enumerate
- 函数式编程
- 多维列表

列表和元组

- 容器(collections)对象：可以容纳多个对象的对象，其每个组成部分称为**元素(element)**
- 有序对象(sequence)：容器对象的元素按照一定顺序排列，可以访问指定位置的元素
- 列表(list)是Python中内置**可变的有序序列**，元组(tuple)是python中内置**不可变的有序序列**
- **列表字面量**定义：开中括号表示列表的开始，配对的闭中括号表示列表的结束，列表的元素之间以逗号隔开，比如[1, 2, 3]
- **元组字面量**定义：比如(1, 2, 3)
  - 与列表字面量类似，开圆括号和闭圆括号表示元组的开始和结束
  - 在不引起歧义时，圆括号也可省略
  - **注意：**如果创建只有一个元素的元组，需要在元素后面加上一个逗号“,”，比如 (1,)



- 列表和元组中的元素：
  - 可以是任意类型的数据(对象)，比如整数、浮点数、字符串等基本类型，也可是列表、元素、字典、集合以及其他自定义类型
  - 每个元素的类型也可各不相同

列表和元组字面量

- 内置函数 **len(obj)**返回容器对象obj中的元素的个数

```
>>> s = []
>>> s
[]
>>> len(s)
0
>>> [1, 2, 3]
[1, 2, 3]
>>> ['if', 'for', 'while']
['if', 'for', 'while']
>>> s = [[1, 2, 3], [4, 5, 6]]
>>> s
[[1, 2, 3], [4, 5, 6]]
>>> len(s)
2
>>> ['tony', ('python', 'math'), 18]
['tony', ('python', 'math'), 18]
```

```
>>> t = ()
>>> t
()
>>> len(t)
0
>>> (1, 2, 3)
(1, 2, 3)
>>> t = (2, [1, 2, 3], [4, 5, 6])
>>> t
(2, [1, 2, 3], [4, 5, 6])
>>> len(t)
3
>>> 1, 2
(1, 2)
>>> x = 1, 2, (3, 4)
>>> x
(1, 2, (3, 4))
>>> x = (3)
>>> x
3
>>> x = 3,  # or (3, )
>>> x
(3,)
```

有序序列

0	1	2	3	4	[0,5)
10	20	30	40	50	len(s)=5
-5	-4	-3	-2	-1	[-5,0)

列表、元组、range对象、字符串都是有序序列

- 通过下标访问有序对象中对应位置的元素： **seq[index]**
- 第一个元素下标为0，第二个元素下标为1，以此类推，直到len(seq)-1
- 最后一个元素下标为-1，倒数第二个元素下标为-2，以此类推，直到-len(seq)
- 负数下标 -i 对应于 len(seq)-i
- 合法下标为 [-len(seq), len(seq)-1]
- 如果下标越界，程序会报错(Exception **IndexError**)
- 字符串也是有序序列对象，也可通过下标访问，表示**第几个字符组成的字符串**

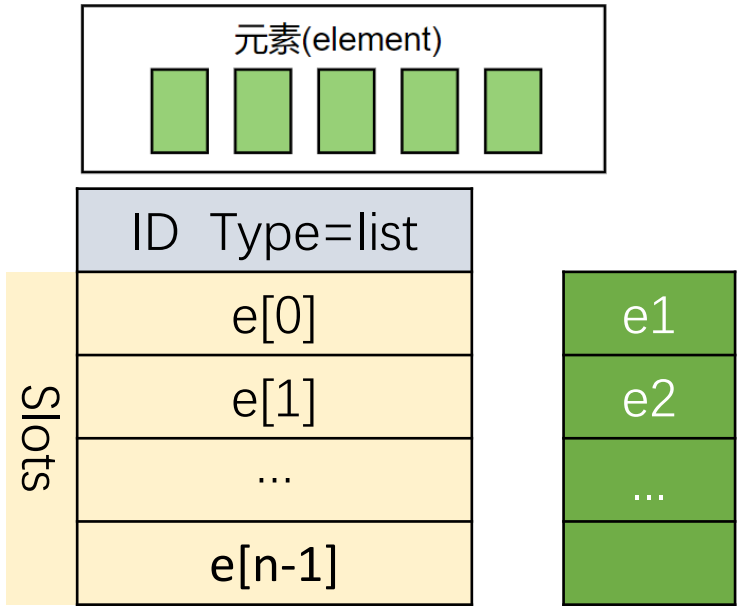
```
>>> x = [1, 2, 3, 4, 5]
>>> x[0]
1
>>> x[-1]
5
>>> x[len(x)]
...
IndexError: list index out of range
```

```
>>> t = (1, 2, 3, 4, 5)
>>> t[0]
1
>>> t[-5]
1
>>> hexdigits = '0123456789abcdef'
>>> digit = 15
>>> hexdigits[digit]
'f'
```

列表和元组

- 列表和元组作为容器对象，其元素可以是任何类型的对象
- 列表和元组的内部实现非常相似：
  - 列表和元组有多个slot，每个slot保存的是对应位置的对象(元素)的引用(相当于保存各个对象的ID)
  - 列表是可变对象，指的是可以改变列表对象的各个slot，即可以增加和删除元素，可以**修改元素(赋值)**
  - 元组是不可变对象，指一旦创建，无法改变元组对象的slot，即无法增加、删除、修改(赋值)元素
- 列表元素的修改： 赋值语句 LHS = RHS
  - LHS为变量var时,将名字空间中的名字与对象空间中的对象(RHS给出的表达式运算后的结果)绑定,即该变量指向RHS给出的对象
  - LHS可为属性形式，比如`import math; math.pi = 3.14`
  - **LHS还可以是下标形式obj[expr],** 即某些可变容器对象(如列表,字典等)中的指定位置中的元素指向RHS表达式所对应的对象

```
x = [1, 2, 3];  x[0] = -1
```



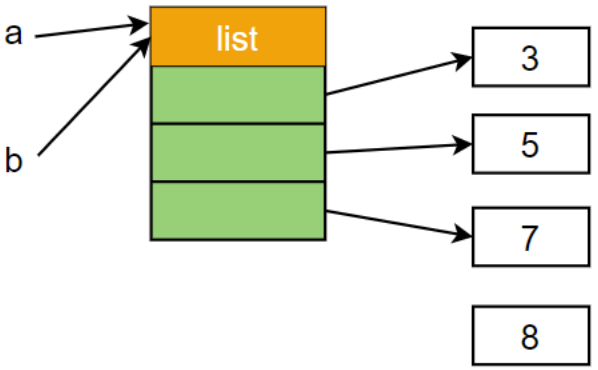
有序对象：下标形式

- 有序对象的下标形式seq[idx]，类似于以前介绍的变量
  - 其出现在表达式中，表示访问该元素所指向的对象
  - 如果seq为列表，则其出现在赋值语句的左边LHS时，表示修改该元素，相当于该列表对象的相应元素指向一个新的对象
    - 为变量（保存对象的引用）赋值时，并不是直接修改变量所指的对象的值，而是使**变量指向新的值(对象)**
    - 为列表对象的元素赋值时，并不是直接修改元素所指向的对象的值，而是使**元素指向新的值(对象)**
  - 元组、字符串等都是不可变对象，因此下标形式seq[idx]不允许出现在赋值语句的左边

```
>>> a = [3, 5, 7]
>>> b = a
>>> a[1]
5
>>> id(a[1])
1733441856
```

```
>>> a[1] = 8
>>> id(a[1])
1733441904
>>> a
[3, 8, 7]
>>> b
[3, 8, 7]
```

```
>>> t = (1, 2, 3)
>>> t[0]
1
>>> t[0] = -1
... TypeError: 'tuple'
object does not support
item assignment
```

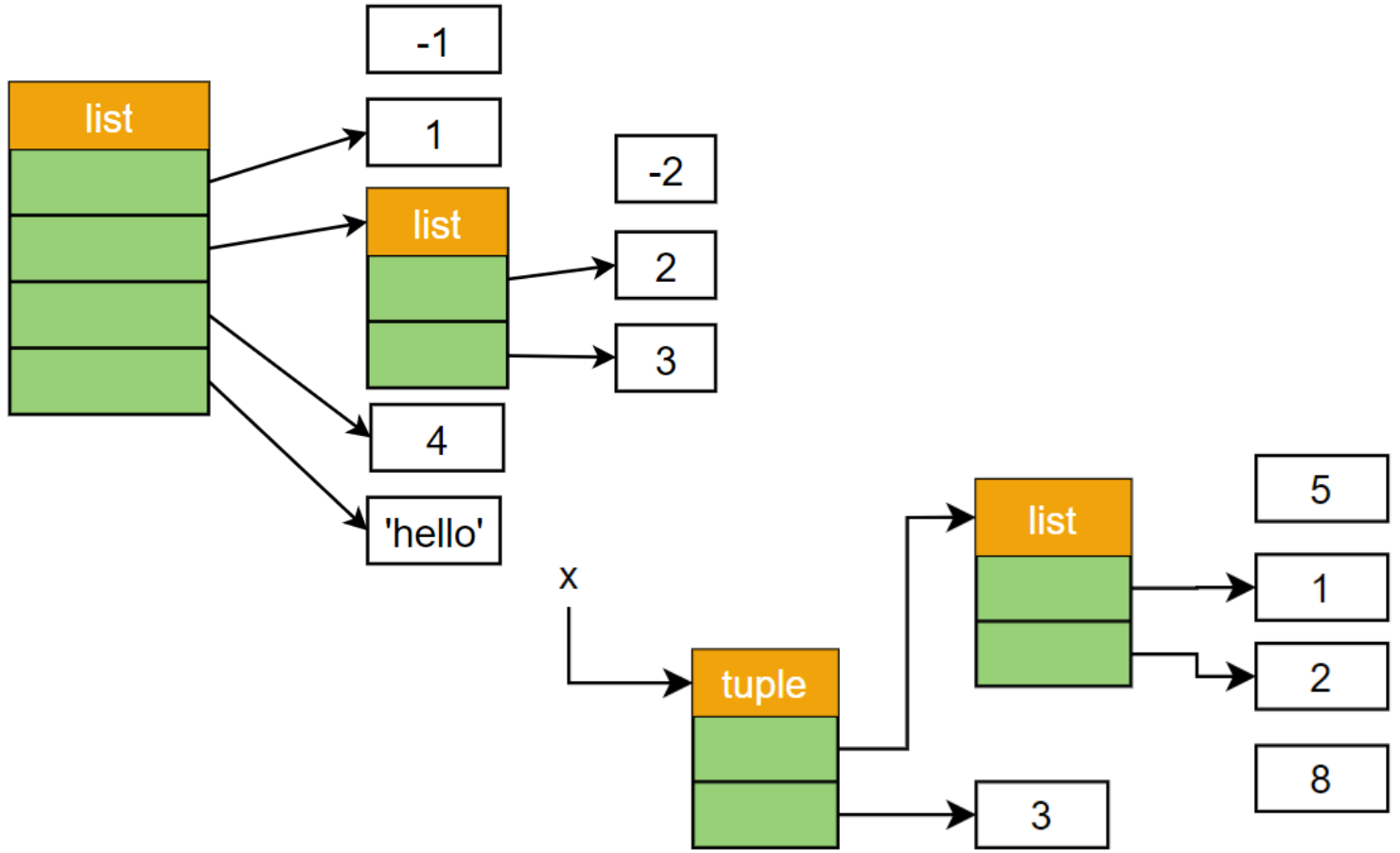


有序对象：下标形式

- 列表和元组中的元素的类型没有限制，其元素可以是一个可通过下标访问的有序对象
  - 多级下标形式：seq[index1][index2]
  - 连续的下标访问采用**左结合**的方式
  - 通过seq[index1]得到seq下标为index1的元素, 比如为e
  - seq[index1][index2]相当于e[index2], 即访问seq (下标为index1)元素所指向的容器对象中的相应(下标为index2)元素
- 列表为**可变对象**, 指其元素可变，即可(通过赋值语句)**修改列表的元素，可增加或删除列表的元素**
  - 通过赋值语句修改列表的元素指的是改变列表元素所指向的对象, 而不是改变列表元素所指向的对象的值
- 元组和列表的元素可指向可变或不可变对象
  - 如果某个元素是一个可变的容器对象（如列表）
    - 可以通过两层下标或相应方法来修改该元素所指向对象的值
    - 但这点与外层容器对象的可变性无关
- 元组不可变，指的是**其元素不可变**（不能增加元素，不能删除元素，不能通过赋值来改变元素，即指向其他对象）
  - **元组的元素指向的对象的可变并没有限制，**可以是可变对象，这样元组对象的值是可变的

有序对象：下标形式

```
>>> d = [1, [2, 3], 4, 'hello']
>>> len(d)
4
>>> d[1]
[2, 3]
>>> d[1][0]
2
>>> d[1][0] = -2
>>> d
[1, [-2, 3], 4, 'hello']
>>> d[0] = -1
>>> d[3]
'hello'
>>> d[3][0] = 'H'
... TypeError: 'str' object does
not support item assignment
```




```
>>> x = ([1, 2], 3)
>>> x[0][0] = 5; x.append(8)
>>> x
([5, 2, 8], 3)
>>> x[0] = x[0] + [10]
TypeError: 'tuple' object does not support item
assignment
```

创建列表或元组

- 采用字面量方法创建一个列表或元组：[1, 2, 3] 或(1, 2, 3)
- 采用构造函数法创建列表或元组: list([iterable])或tuple([iterable])
  - list或tuple函数的参数可为空，这时产生空列表或空元组
  - list或tuple函数的参数可为可迭代（iterable）对象(比如列表、range对象、字符串、元组、字典、集合等)
    - 创建一个新的列表或元组对象，该对象的每个元素指向可迭代对象中各个元素指向的对象
    - tuple函数的参数为元组时，由于元组是不可变对象，因此返回该元组
- 在已有列表对象时，还可以通过该对象的copy()方法，创建一个新的列表对象，该对象的每个元素指向原列表对象的对应位置的元素

```
>>> a=(1, 2)
>>> tuple(a) is a
True
```

```
>>> x = list() #创建空列表
>>> x
[]
>>> list([1, 3, 5, 7, 9])
[1, 3, 5, 7, 9]
>>> list(range(1, 10, 2))
[1, 3, 5, 7, 9]
```

```
>>> list((1, 3, 5, 7, 9))
[1, 3, 5, 7, 9]
>>> list('hello world')
['h', 'e', 'l', 'l', 'o', ' ',
'w', 'o', 'r', 'l', 'd']
>>> s = list(range(5))
>>> s.copy()
[0, 1, 2, 3, 4]
```

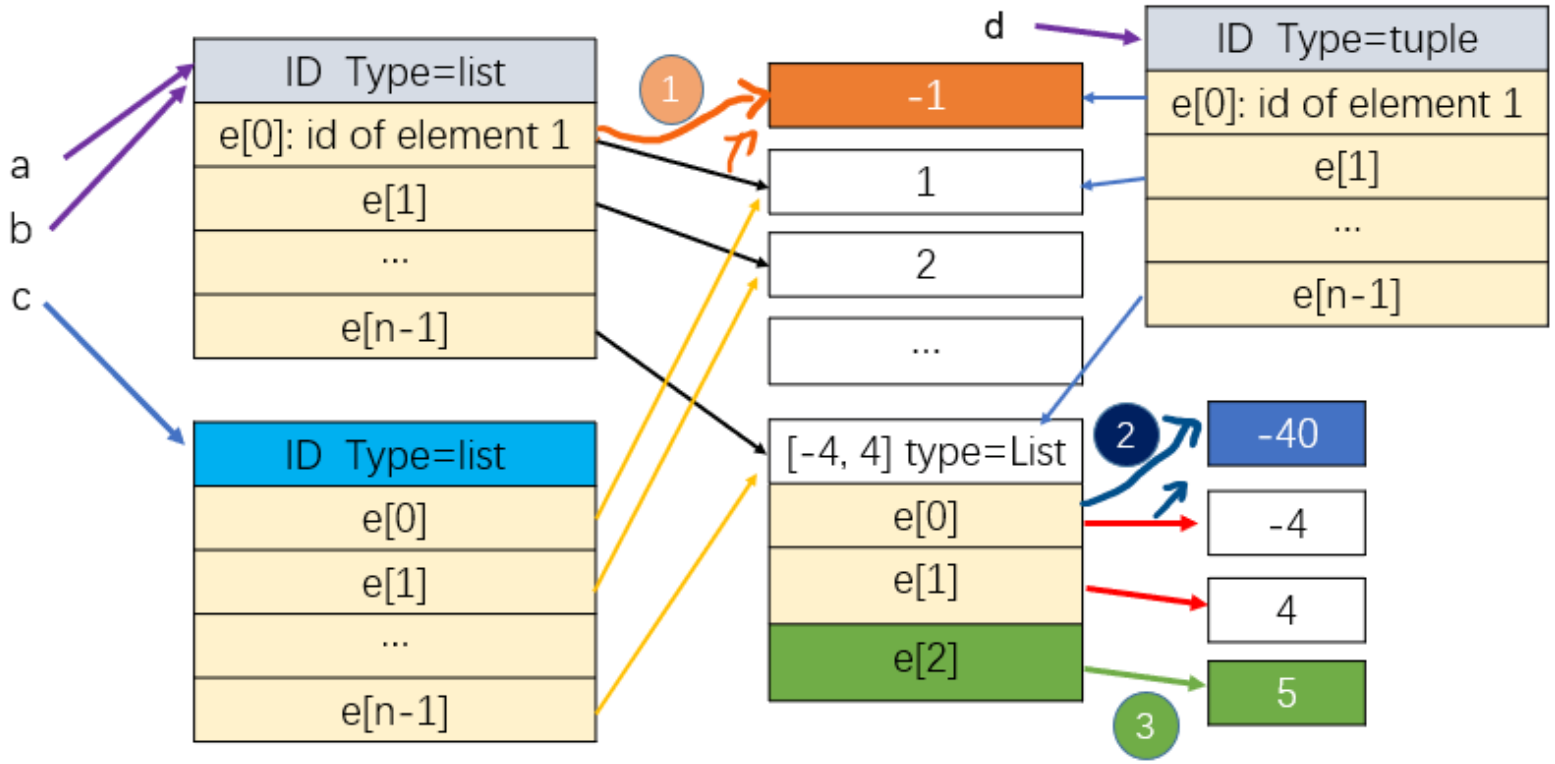
```
>>> tuple()
()
>>> tuple(range(5))
(0, 1, 2, 3, 4)
>>> tuple([1, 2, 3])
(1, 2, 3)
>>> tuple('123')
('1', '2', '3')
```

浅拷贝(shallow copy)

- 调用构造函数基于可迭代对象构造一个新的序列对象时，新的序列对象的元素与原可迭代对象中的对应元素指向同一个对象，即构造时采用**浅拷贝**(shallow copy)
- python对于容器对象的许多操作（字面量以及将介绍的加法、乘法和切片等）缺省采用浅拷贝
- copy模块的deepcopy**方法进行深拷贝**，返回同种类型的对象，确保新对象与原对象隔离，即改变某个对象不会影响另一个对象

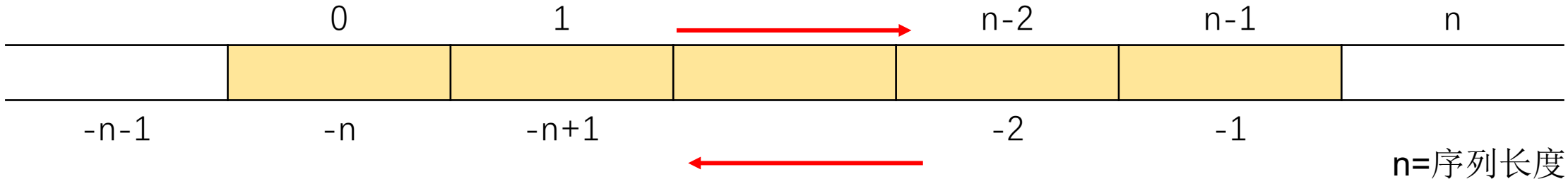
```
e = (1, 2, 3, [-4, 4])
a = [1, 2, 3, [-4, 4]]
b = a
c = list(a)
d = tuple(a)
a[0] = -1          # 1
c[3][0] = -40      # 2
c[3].append(5)     # 3
```

```
>>> import copy
>>> s2 = copy.deepcopy(s)
>>> s2
[0, 1, [3, 4, 5]]
>>> s2[-1] is s[-1]
False
```



有序对象(列表、元组和字符串等）的访问

- seq[index]:使用**下标**访问列表元素，如果下标越界，抛出IndexError，如果下标不为整数或切片，抛出TypeError



- seq.index(value, [start, [stop]]) 获取指定值value在指定范围[start, stop]内首次出现的下标。缺省为从有序对象的第一个元素，到最后一个元素（包括）结束。也可以指定从start开始到stop(不包括)结束。如果无法找到值相等的元素，抛出异常ValueError

```
>>> a = [3, 4, 5, 7, 9, 11, 9, 7, 7, 6, 5]
>>> a[3]
7
>>> a.index(7)
3
>>> a.index(7, 4)
7
>>> a.index(7, 8, -1)
8
```

```
>>> a.index(13)
... ValueError: 13 is not in list
>>> t = (1, 2, 3, 2, 3)
>>> t.index(3)
2
```

有序对象(列表、元组和字符串等）的访问

- **seq.count(value)** 统计指定元素value(相等关系)在列表对象中出现的次数，不会抛出异常
- 成员关系判断运算符 in和not in
  - value in seq 判断一个值是否存在于有序对象中(即是否有元素==value)，结果为True或False
  - value not in seq：如果value不出现在有序对象中，结果为True，否则False
  - 运算符in也可用于其他容器对象，包括字典和集合等
  - for循环中使用关键字in对可迭代对象的元素进行遍历，for obj in iterable:

```
>>> a = [3, 4, 5, 7, 9, 11, 9, 7, 7, [6, 5]]
>>> a.count(7)
3
>>> a.count(0)
0
>>> 7 in a
True
>>> 7 not in a    # 不建议使用 not 7 in a
False
>>> 0 not in a
True
```

```
>>> [6, 5] in a
True
>>> t = (1, 2, 3, 2, 3)
>>> t.count(3)
2
>>> t.count(0)
0
>>> 3 in t
True
>>> 0 not in t
True
```

巧用成员关系判断运算符 in

```
x == value1 or x == value2 or x == value3 or x == value4
```

等价于： `x in (value1, value2, value3, value4)`

```
if x == 3 or x == 5 or x == 7:
    print('...')

if x in (3, 5, 7):
    print('...')

text = input('Please input...')
if text.lower() in ('bye', 'quit', 'exit'):
    print('bye!')
```

但是如果知道x为整数，且判断x是否为1到n的整数时:

- 绝对不要使用 x in range(1, n+1)
- 应该使用 1 <= x <= n

字符串的元素访问和计数

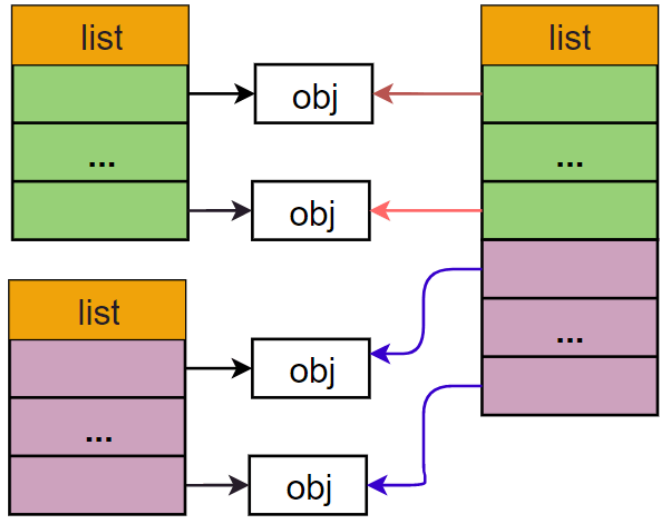
```
>>> 'pppppp'.count('pp')
3
```

- 字符串是有序序列，可以通过下标访问指定位置的元素(即由对应位置的字符组成的字符串），可支持in, count, index，只是其参数除了是单个字符组成的字符串外，还可是长度超过1的**子字符串**
- strobj.index(sub[,start[,end]]) 返回子串sub(而不仅仅是单个字符)在指定范围内**首次出现**的位置（下标），**如果不存在则抛异常ValueError**
- strobj.count(sub[,start[,end]]) 返回子串sub出现的次数，找到子串后，在之后的下一个位置继续...
- sub in S　sub not in S 判断子串sub是否在S中出现
- rindex(sub[,start[,end]]) 返回子串sub在指定范围内**最后一次**出现的位置，**如果不存在则ValueError**
- find(sub[,start[,end]]) rfind(sub[,start[,end]]) 与index/rindex类似，返回子串sub在指定范围内首次或最后一次出现的位置，**如果不存在则返回-1**

```
>>> s = "apple,peach,banana,peach,pear"
>>> s.index('peach')
6
>>> s.index('ppp')
Traceback (most recent call last):
...
ValueError: substring not found
>>> s.rindex('peach')
19
>>> s.count('an')
2
```

```
>>> s.find("peach")
6
>>> s.find("peach", 6 + 5)
19
>>> s.find("peach", 11, 24)
19
>>> s.rfind('p')
25
>>> s.rfind('ppp')
-1
```

有序对象的加法和乘法运算



- 运算符“+”
  - 同种类型的有序对象相加，即列表+列表、元组+元组、字符串+字符串
  - 创建一个新的同种类型的有序对象，由两个有序对象的元素按照先后顺序组成
- 运算符“*”
  - 有序对象(列表、元组、字符串等）与整数相乘
  - 生成一个新的有序对象，相当于原有序对象中的元素按照顺序重复出现多次
  - 对于元组或列表而言，新的元组或列表中会有多个元素指向同一个对象(原元组或列表中的对应位置的元素所指向的对象)

```
>>> s = a = [3, 4, 5] + [7]
>>> a
[3, 4, 5, 7]
>>> a = a + [[7]]
>>> a
[3, 4, 5, 7, [7]]
>>> s is a
False
>>> a + 7
... TypeError: can only concatenate list (not "int") to list
>>> a = [1, 2, 3]
>>> a = a * 2
>>> a
[1, 2, 3, 1, 2, 3]
```

```
>>> a = [1, 2, 3]
>>> a = a * 2
>>> a
[1, 2, 3, 1, 2, 3]
>>> t = (1, 2, [3])
>>> t1 = t * 2
>>> t1
(1, 2, [3], 1, 2, [3])
>>> t[-1][0] = 0
>>> t
(1, 2, [0])
>>> t1
(1, 2, [0], 1, 2, [0])
```

有序对象的比较运算

- 关系运算符(大于、小于、大于等于、小于等于、等于、不等于）可比较数值或序列的大小
- 等于和不等于几乎可以用于任何类型，而且不同类型的对象也可以比较相等关系
- **数字类型**（bool、整数、浮点等）的对象之间可以比较，或者是相同类型的对象才有可能比较
- 列表/元组/字符串等有序序列的（**同类**）对象可以比较，按顺序比较各个元素的大小，如果前面比较确定了大小关系，则结束。如果其中某个对象没有对应的元素，则无元素表示其最小
- 字符之间的比较按照其在(Unicode)字符集中出现的先后顺序。 空格… 0…9 …A…Z… a…z …
- 算术运算符和关系运算符的优先级？ 算术运算符的优先级更高，即先算再比较

```
>>> 4 != 'ab'
True
>>> [1, 2] == (1, 2)
False
>>> [1, 2, 3] < [1, 2, 4]
True
>>> (1, 2, 3, 4) < (1, 2, 4)
True
>>> [1, 2] < [1, 2, -1]
True
```

```
>>> [1, 2, 3] == [1.0, 2.0, 3.0]
True
>>> 'a'>'A'
True
>>> 'ABC' < 'C' < 'Pascal' < 'Python'
True
>>> [1, 2, ['aa', 'ab']] < [1, 2, ['abc', 'a'], 4]
True
>>> [1,2] + [3] == [1,2,3]
True
```

主要内容

- 有序对象：字符串、列表和元组
- **可变有序对象：列表**
- 有序对象的切片
- 序列解包
- 用于序列的常用内置函数：zip和enumerate
- 函数式编程
- 多维列表

列表对象的方法

	方法	说明
增加	list.append(x)	将元素x添加至列表尾部
	list.extend(L)	将列表L中所有元素添加至列表尾部
	list.insert(index, x)	在列表指定位置index处添加元素x
删除	list.remove(x)	在列表中删除首次出现的指定元素
	list.pop([index])	删除并返回列表对象指定位置的元素
	list.clear()	删除列表中所有元素，但保留列表对象，该方法在Python2中没有
查找	list.index(x)	返回值为x的首个元素的下标
	list.count(x)	返回指定元素x在列表中的出现次数
其他	list.reverse()	对列表元素进行原地逆序
	list.sort()	对列表元素进行原地排序
	list.copy()	返回列表对象的浅拷贝

```
del list_var
del list_var[index]
```

```
list_var[index]
x in list_var
x not in list_var
```

切片slicing：获得列表中的多个元素

19

列表对象的方法：增加元素

```
>>> a = a.append(5)
>>> print(a)
None
```

不要这样调用！！

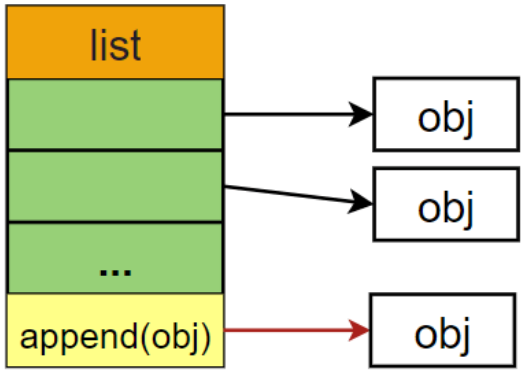
```
a = a.append(obj)
a = a.extend(iterable)
a = a.insert(idx, obj)
```

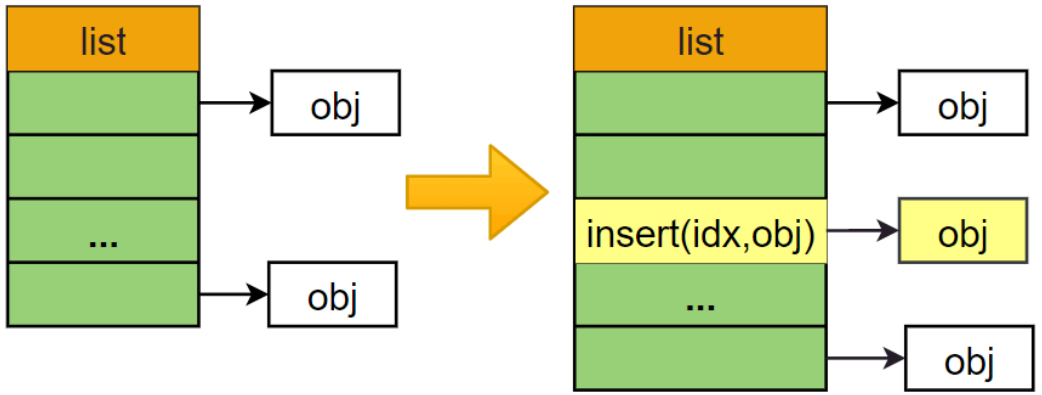
- append(obj): 将obj附加到列表对象，成为最后一个元素
- extend(iterable)： 将另一个**可迭代对象**的所有元素添加至该列表对象尾部
- insert(index, object)：将元素添加至列表的指定位置，在原index对应的位置插入新元素，原来位置以及后面的元素相应往后移动一个位置。 **index必须为整数**，否则throw TypeError
- **原地操作**列表， 返回值为None

```
>>> a = [3, 4, 5, 7]
>>> id(a)
89539448
>>> a.append(9)
>>> a
[3, 4, 5, 7, 9]
>>> id(a)
89539448
>>> a.append([10])
>>> a
[3, 4, 5, 7, 9, [10]]
>>> a.extend(1)
... TypeError: 'int' object is not iterable
```

```
>>> a = [1, 2]
>>> a.extend([3, 4])
>>> a.extend([[5, 6]])
>>> a
[1, 2, 3, 4, [5, 6]]
>>> a.extend(range(2))
>>> a
[1, 2, 3, 4, [5, 6], 0, 1]
>>> a  = [3, 4, 5, 7, 9]
>>> a.insert(3, 6)
>>> a
[3, 4, 5, 6, 7, 9]
>>> a.insert(0, 2) #最前面插入
>>> a
[2, 3, 4, 5, 6, 7, 9]
>>> a.insert(len(a), -1) # 附加
```

```
>>> a
[2, 3, 4, 5, 6, 7, 9, -1]
>>> a.insert(-1, -2) # -2 成为倒数第二个
>>> a
[2, 3, 4, 5, 6, 7, 9, -2, -1]
```





列表对象的方法：增加元素

复合赋值语句

list_obj += iterable 等价于 list_obj.extend(iterable)，**原地附加**多个元素

list_obj *= 4 与*运算类似，只是**原地重复**，重复元素实际上是多个元素**指向同一对象**

```
>>> s1 = list(range(4))
>>> alias = s1
>>> s1 += [4, 5]
>>> s1 += 'ok'
>>> s1
[0, 1, 2, 3, 4, 5, 'o', 'k']
>>> s1 is alias
True
>>> s1 *= 3
>>> s1
[0, 1, 2, 3, 4, 5, 'o', 'k', 0, 1, 2, 3, 4, 5, 'o', 'k', 0, 1, 2,
3, 4, 5, 'o', 'k']
>>> s1 is alias
True
```

列表对象的方法：增加元素

#	列表元素增加方法	别名	效果
1	append(obj)	附加	原地修改
2	extend(iterable)	扩展	原地修改
3	insert(index, obj)	插入	原地修改
4	list1 += iterable	复合赋值	原地修改
5	list1 *= n	复合赋值	原地重复
6	list1 + list2	拼接	**新建列表**
7	list1 * n	复制	**新建列表**

求n以内的平方数

square_numbers.py

```
import math
def square_numbers(n=100):
    numbers = []
    for i in range(n+1):
        k = int(math.sqrt(i))
        if k * k == i:
            numbers.append(i)
    return numbers
```

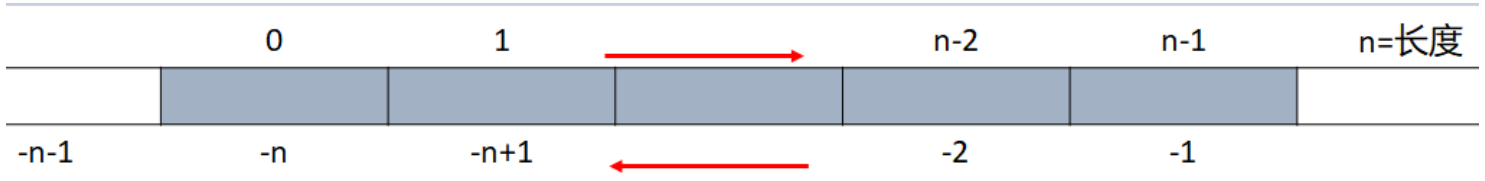
注意：

insert()方法会涉及到插入位置之后所有元素的移动，这会影响处理速度

建议：除非有必要，否则应尽量避免在列表中间位置插入元素的操作，而是优先考虑使用append()等方法在尾部附加

```
def square_numbers2(n=100):
    numbers = []
    for i in range(int(math.sqrt(n)) + 1):
        numbers.append(i*i)
    return numbers
```

列表对象的方法：删除元素



- 删除指定位置的元素：del seq[idx]
  - **下标越界时，抛出异常IndexError: list assignment index out of range**
  - **下标不为整数或切片对象时，抛出异常 TypeError: list indices must be integers or slices, not ...**
- 删除指定位置的元素，并且得到该元素：seq.pop(idx)
- 根据值删除第一个出现（即值相等）的元素：seq.remove(value)
  - 如果列表中有元素的值与value的值相等（seq[index] == value），则删除第一个值相等的元素，**返回None**。
    如果列表中不存在要删除的元素，则抛出异常ValueError

```
>>> a = list(range(3, 16, 2))
>>> a
[3, 5, 7, 9, 11, 13, 15]
>>> del a[1], a[-1]
>>> a
[3, 7, 9, 11, 13]
>>> del a[5]
... IndexError: list
assignment index out of range
>>> a.pop()
13
>>> a.pop(1)
7
```

```
>>> a.pop(-1)
11
>>> a = [3, 5, 7, 9, 7, [4, 5]]
>>> a.remove(7)
>>> a
[3, 5, 9, 7, [4, 5]]
>>> a.remove(7)
>>> a
[3, 5, 9, [4, 5]]
>>> a.remove(7)
... ValueError: list.remove(x): x not in list
```

```
>>> a.remove([4, 5])
>>> a
[3, 5, 9]
```

列表对象的方法：删除元素

- 列表的clear()方法，删除列表对象的所有元素，即变为空列表

```
>>> s = [1, 2, 3]
>>> s.clear()
>>> s
[]
```

方法	作用	返回	备注
del list_obj[index]	删除指定位置元素	无返回值	IndexError,TypeError
pop([index])	删除指定位置元素并返回该元素	返回元素	IndexError,TypeError
remove(value)	删除首次出现的指定元素，返回None	返回None	没有相应元素时ValueError
clear()	删除所有元素	返回None	空列表

注意：

1. insert()方法会涉及到插入位置之后所有元素的移动，这会影响处理速度
2. 列表删除方法pop()弹出非尾部元素时，也有类似问题

建议：除非有必要，否则应尽量避免在列表中间或头部位置位置插入和删除元素的操作，而是优先考虑使用append()方法尾部附加和pop()方法尾部删除

列表元素的删除+for循环

- 如果要删除列表中所有出现的某个元素怎么办？ for循环遍历，发现元素时删除该元素

```python
def remove_all(list_, value):
    for item in list_:
        if item == value:
            list_.remove(value)
    return list_
```

remove_all.py

```python
def remove_all_0(list_, value):
    for i in range(len(list_)):
        print(i)
        if list_[i] == value:
            del list_[i]
    return list_
```

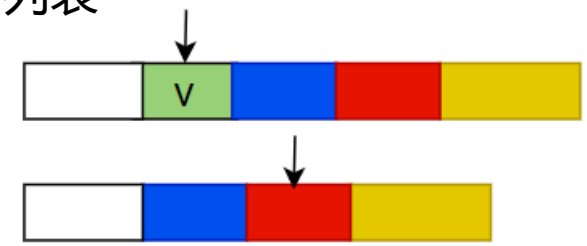
```python
x = [1,2,1,2,1,2,1,2,1]
print(remove_all(x,1))
# [2, 2, 2, 2]
x = [1,2,1,2,1,1,1]
print(remove_all(x,1))
# [2, 2, 1] ???
```

每轮循环从列表中取下一个元素，
但循环体中修改了列表

列表的迭代器实现：

- 一般假设原有的可迭代对象并不会改变，至少下标不变
- 迭代器：维护当前下标pos
- 如果pos < len(x)，则下一个元素为x[pos]；pos += 1

pos = 0	[1,2,1,2,1,1,1]	i = x[pos] → 1	remove
pos = 1	[2,1,2,1,1,1]	i = x[pos] → 1	remove
pos = 2	[2,2,1,1,1]	i = x[pos] → 1	remove
pos = 3	[2,2,1,1]	i = x[pos] → 1	remove
pos = 4	[2,2,1]	pos >=len(x)	stop



基本观察：

1. 每当插入或删除一个元素之后，该元素位置后面所有元素的索引就都改变了
2. 迭代器实现一般假设原有的可迭代对象并不会改变，至少下标不变

例子中：一旦remove(value)或del list_[i]，后面的元素往前移动一个位置，导致会跳过后面一个元素!

列表元素的删除+for循环

x = [1,2,1,2,1,1,1]　　复制 xx = [1,2,1,2,1,1,1]

```
def remove_all_2(list_, value):
    for item in list_.copy():   # list_[:]
        if item == value:
            list_.remove(value)
    return list_
```

```
def remove_all_3(list_, value):
    for i in range(len(list_) - 1, -1, -1):
        if list_[i] == value:
            del list_[i]
    return list_
```

基本观察：

1. 每当插入或删除一个元素之后，该元素位置后面所有元素的索引就都改变了
2. 迭代器实现一般假设原有的可迭代对象并不会改变，至少下标不变

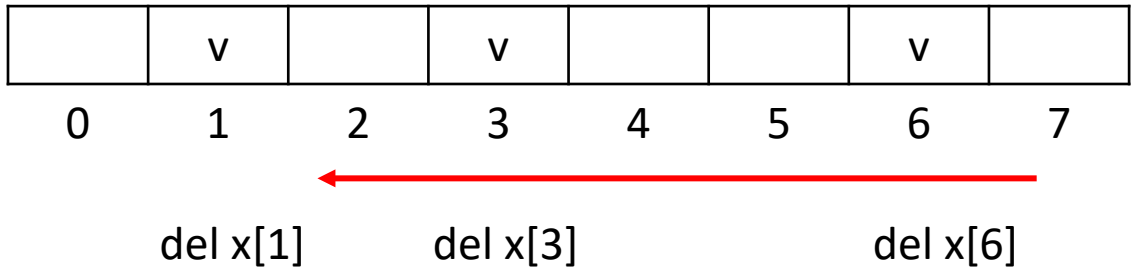
怎么办？ 优先第一种

1. for循环中，可迭代对象不是原始的列表，而是使用**克隆的列表（如列表的切片）替代**，即使原始列表因元素删除或增加而变化，切片时已经生成的列表不会变化
2. 使用正确的顺序，例如从后往前依次判断。**通过下标**从后面删除时，被删除元素之前的元素的下标不会改变
3. 不使用for循环，而是采用while，并且在循环体中考虑下标或索引改变的情况

注意while的条件表达式每轮循环都会重新计算，但是for循环 in后面的表达式只会计算一次

it_ = iter(list_[:])

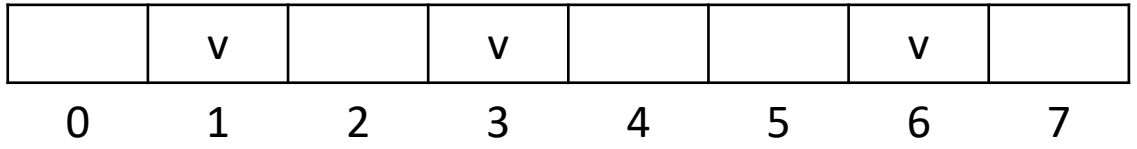
loop item = next(it_) ….



列表元素的删除+for循环

```
def remove_all_4(list_, value):
    i = 0
    while i < len(list_):
        if list_[i] == value:
            del list_[i] #不需要更新i
        else:
            i = i + 1
    return list_
```

- 初始化 i = 0，表示从头开始检查
- 循环体中：
  - 正常情况下 i = i + 1， 检查下一个元素
  - 当前元素为value时，移走该value，下一个元素的下标仍然是i

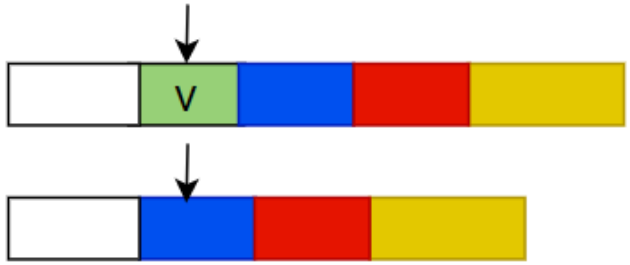


基本观察：

1. 每当插入或删除一个元素之后，该元素位置后面所有元素的索引就都改变了
2. 迭代器实现一般假设原有的可迭代对象并不会改变，至少下标不变

怎么办？ 优先第一种

1. for循环中，可迭代对象不是原始的列表，而是使用**克隆的列表（如列表的切片）替代**，即使原始列表因元素删除或增加而变化，切片时已经生成的列表不会变化
2. 使用正确的顺序，例如从后往前依次判断。**通过下标**从后面删除时，被删除元素之前的元素的下标不会改变
3. 不使用for循环，而是采用while，并且在循环体中考虑下标或索引改变的情况



列表的排序

(1) 使用列表对象的**sort方法**进行**原地**排序。 help(list.sort)

L.sort(key=None, reverse=False) -> 返回None， 默认基于元素间的大小关系排序，升序排列

- reverse缺省为False，表示升序，即从小到大；如果传递参数reverse=True时为降序排列
- key缺省为None，表示基于元素间的大小关系排序。也可传递函数对象，作为排序的基准

(2) 使用**内置函数sorted**对可迭代对象(列表/字符串/元组等)进行排序并**返回新列表**

sorted(iterable, key=None, reverse=False)

```
>>> s = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> import random
>>> random.shuffle(s)
>>> s
[13, 5, 6, 11, 9, 7, 15, 4, 3, 17]
>>> s1 = s[:]
>>> s.sort() #默认为升序排列
>>> s
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> s1.sort(reverse=True)   #降序排列
>>> s1
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
```

```
>>> s = [10, 5, 23, 20]
>>> s1 = sorted(s)
>>> s1
[5, 10, 20, 23]
>>> s2 = sorted(s, reverse=True)
>>> s2
[23, 20, 10, 5]
>>> sorted('python')
['h', 'n', 'o', 'p', 't', 'y']
```

列表的逆序

- 使用列表对象的**reverse方法**将元素原地逆序(即列表中元素出现的相反顺序），注意不等同于L.sort(reverse=True)
- 使用**内置函数reversed**对可迭代对象(如列表)的元素进行逆序排列并返回**reversed对象**
  - sorted函数返回新的列表
  - reversed函数返回的不是列表，而是一个可迭代对象，更是一个**迭代器**

```
t  = [3, 4, 6, 7, 5, 9, 11, 17, 13, 15]
t.reverse()
print(t)
# [15, 13, 17, 11, 9, 5, 7, 6, 4, 3]
```

```
m = reversed(t)
print(type(m))
print(list(m))

for item in m:
    print(item)

print('-' * 40)

for item in reversed(t):
    print(item, end=', ')
```

右边代码的输出：

```
<class 'list_reverseiterator'>
[3, 4, 6, 7, 5, 9, 11, 17, 13, 15]
----------------------------------------
3, 4, 6, 7, 5, 9, 11, 17, 13, 15,
```

主要内容

- 有序对象：字符串、列表和元组
- 可变有序对象：列表
- **有序对象的切片**
- 序列解包
- 用于序列的常用内置函数：zip和enumerate
- 函数式编程
- 多维列表

有序对象的切片(slice)

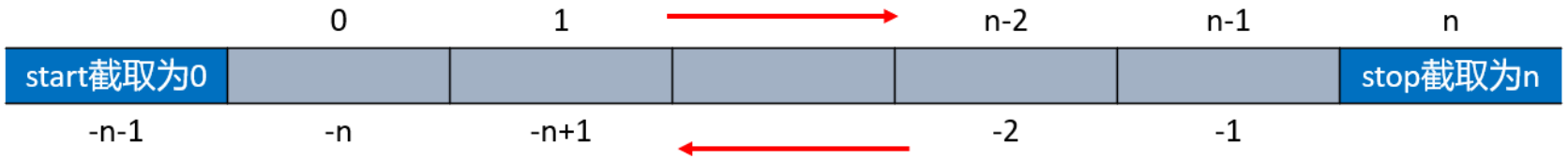
- 可通过**整数下标**来访问有序对象对应位置的元素
- 可通过**整数下标与赋值语句**结合修改列表中对应位置的元素
- 可通过**整数下标与del语句**结合删除列表中对应位置的元素
- 整数下标(指定的某个位置）→ 切片下标（指定的多个位置）
  - 可通过**切片下标**访问有序对象中多个位置的元素
  - 可通过**切片下标与赋值语句**结合修改列表中多个位置的元素
  - 可通过**整数下标与del语句**结合删除列表中多个位置的元素
- 切片格式:
  - start:stop 从start开始到stop为止（不包括）的元素
  - start:stop:step： 指定步长，而不是缺省的连续元素

```
>>> s = list(range(10))
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s[-1]
9
>>> s[-1] = -s[-1]
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, -9]
>>> s[0:4]
[0, 1, 2, 3]
>>> s[0:4] = [0] * 4
>>> s
[0, 0, 0, 0, 4, 5, 6, 7, 8, -9]
```

有序对象的切片(slice):基本切片

seq[start:stop]：有序对象seq中那些下标在[start,stop)的元素（可0个），组成一个新的同种类型的有序对象

- start和stop都可省略，表示采用缺省值。如seq[:5] seq[5:] seq[:]
- start缺省为0，而stop缺省为**列表结束即len(s)**
- start<stop (下标意义上的比较，而不是简单的整数大小比较）才可访问到相应的元素，否则对应的元素序列为空
  - 合法的下标范围为 [-len(seq), len(seq))
  - 切片的下标超出范围时，会截取到合适的下标
  - 如果start<-len(seq)，则相当于0，或相当于-len(seq)
  - 如果stop超过len(seq)，则相当于len(seq)

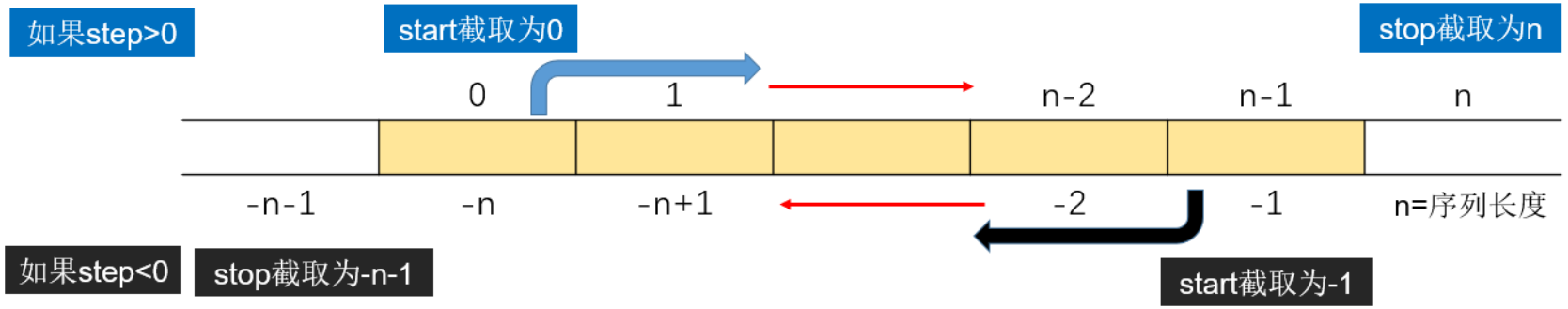



```
>>> s = list(range(10))
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s[3:6]
[3, 4, 5]
>>> s[:5]
[0, 1, 2, 3, 4]
>>> s[5:]
[5, 6, 7, 8, 9]
>>> s[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s[1:1]
[]
>>> s[1:-1]
[1, 2, 3, 4, 5, 6, 7, 8]
>>> s[6:20]
[6, 7, 8, 9]
>>> s[-20:3]
[0, 1, 2]
>>> s[20:3]
[]
>>> m = 'abcdefg'
>>> m[1:-1]
'bcdef'
>>> t = tuple(range(4))
>>> t
(0, 1, 2, 3)
>>> t[1:-1]
(1, 2)
```

有序对象的切片(slice):扩展切片

seq[start:stop:step]: step可以省略，表示缺省为1。描述了有序对象中从start开始, 每隔step的元素，直到stop（不包括）为止

- step可以为负值，表示从有序对象中start开始，从后面往前访问相应位置的元素，直到stop （不包括）为止。这些元素之间相隔指定间隔step
- 注意到列表的合法下标范围为 [-len(s), len(s))，切片的下标超出范围时，会截取到合适的下标
- 如果step>0, 则start缺省为列表开头(start=0), stop缺省为列表长度或者最后一个元素再后一个位置，stop=len(seq)
- 如果step<0, 则start缺省为列表最后一个元素，start=len(seq)-1或-1，stop缺省为列表第一个元素再前一个位置, stop=-len(seq) - 1




```
>>> s = list(range(10))
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> s[::2]
[0, 2, 4, 6, 8]
>>> s[1::2]
[1, 3, 5, 7, 9]
>>> s[2:-2:2]
[2, 4, 6]
>>> s[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> s[-1:-4:-1]
[9, 8, 7]
>>> s[-1::-2]
[9, 7, 5, 3, 1]
>>> s[-2::-2]
[8, 6, 4, 2, 0]
```

有序对象的切片(slice): slice对象（拓展）

- 切片下标除了冒号形式外，也可以是一个slice对象。内置函数slice与range的用法类似

slice(stop)

slice(start, stop[, step])

如果某个值要采用缺省值，传递参数None

s[:stop]	s[slice(stop)]
s[start:stop]	s[slice(start, stop)]
s[start:stop:step]	s[slice(start, stop, step)]

- 什么时候会用到slice对象呢？在代码比较复杂的时候，slice对象可提高可读性

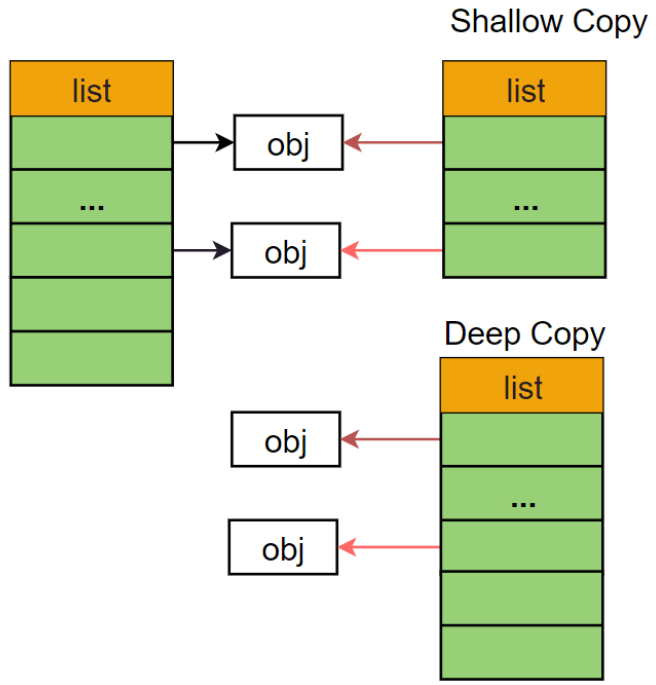
```
>>> s = list(range(8))
>>> clone_slice = slice(None, None)
>>> last_three_slice = slice(-3, None)
>>> reverse_slice = slice(None, None, -1)
>>> t1 = s[clone_slice]
>>> t2 = s[last_three_slice]
>>> t3 = s[reverse_slice]
```

```
>>> s
[0, 1, 2, 3, 4, 5, 6, 7]
>>> s[:]
[0, 1, 2, 3, 4, 5, 6, 7]
>>> s[-3:]
[5, 6, 7]
>>> s[::-1]
[7, 6, 5, 4, 3, 2, 1, 0]
```

有序对象的切片(slice):切片的使用

- **出现在表达式**中时:
  - 下标为整数时，seq[idx] 表示访问有序对象中对应位置的元素
  - 下标为切片时，seq[slice]表示访问有序对象中多个位置的元素，最终得到的是由那些位置的元素组成的新的同种类型的有序对象
  - 注意：切片得到的新的有序对象的元素与原有序对象的对应元素指向同一个对象，即浅拷贝
- 如果有序对象是可变的，即有序对象seq为列表时:
  - **下标与del语句结合**:
    - 下标为整数时，del seq[idx]表示删除列表中对应位置的元素
    - 下标为切片时，del seq[slice]表示删除列表中多个对应位置的元素
  - **下标与赋值语句结合**:
    - 下标为整数时，seq[idx] = value，表示列表对应位置的元素改变为value
    - 下标为切片时，seq[slice] = iterable，表示改变（修改或删除）列表多个位置的元素

使用下标访问时下标越界会抛出异常，切片操作则不会，下标超出范围时，仅返回能遍历到的元素或空序列，代码具有更强的健壮性




```
>>> s = [[1, 2], [3, 4]]
>>> s1 = s[:]
>>> import copy
>>> s2 = copy.deepcopy(s)
>>> s1[0] is s[0]
True
>>> s2[0] is s[0]
False
```

通过切片修改列表

使用切片原地修改列表内容：添加多个新元素

`seq[i:i] = iterable`

赋值语句左边为列表的切片形式，且start/stop为同一个位置，右边为iterable对象，即在列表对应的位置i处插入iterable对象中的各个元素

- 可以在尾部添加多个元素: seq[len(seq):]
- 可以在头部插入多个元素: seq[0:0]
- 可以在中间(位置i)插入多个元素: seq[i:i]

使用切片原地修改列表内容，替换或者删除元素

`seq[start:stop] = iterable`

`seq[start:stop:step] = iterable`

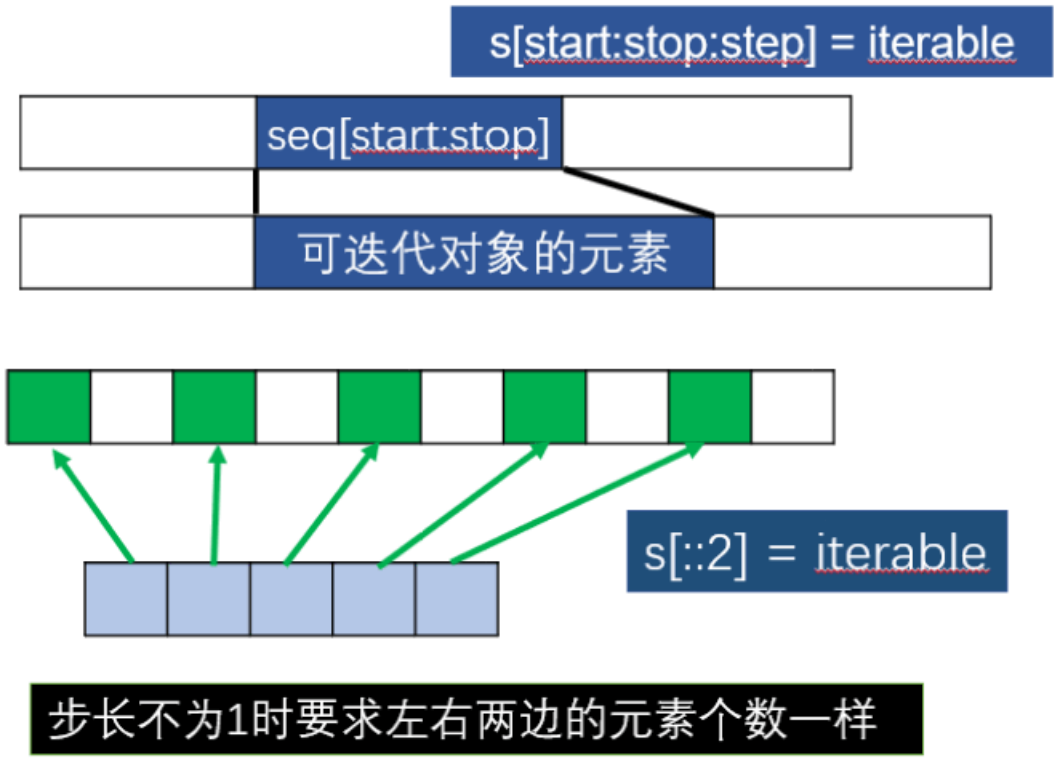
赋值语句左边的切片下标描述的位置**不为空**，相当于将指定位置的元素**替换为**赋值语句右边iterable对象中的各个元素

- 右边iterable对象长度为0，比如为空列表时，相当于删除对应的元素
- step为1时，即连续位置，不要求两边元素个数一样

```
>>> s = [3, 5, 7]
>>> s[len(s):] = [9, 11] #尾部
>>> s
[3, 5, 7, 9, 11]
>>> s[0:0] = [-1, 1]    #头部
>>> s
[-1, 1, 3, 5, 7, 9, 11]
>>> s[3:3] = [51, 53]  #指定位置
>>> s
[-1, 1, 3, 51, 53, 5, 7, 9, 11]
>>> s[-1:-1] = [10]
>>> s
[-1, 1, 3, 0, 51, 53, 5, 7, 9, 10, 11]
>>> s[:3] = ()  # 删除前面3个元素
>>> s
[0, 51, 53, 5, 7, 9, 10, 11]
>>> s[:3] = [1, 2] # 替代前面3个元素
>>> s
[1, 2, 5, 7, 9, 10, 11]
>>> s[:3] = range(4)
>>> s
[0, 1, 2, 3, 7, 9, 10, 11]
```

通过切片修改列表

- 列表的切片和赋值语句结合时，切片步长不为1时，要求左右两边的元素个数一样
  - 切片和赋值语句结合只能删除连续多个元素
- 列表的切片和del语句结合： del seq[slice]
  - 对于slice中的步长没有限制




```
>>> s = list(range(9))
>>> s
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> s[::2] = [0] * ((len(s) + 1) // 2)
>>> s
[0, 1, 0, 3, 0, 5, 0, 7, 0]
>>> s[1::2] = [1] * (len(s) // 2)
>>> s
[0, 1, 0, 1, 0, 1, 0, 1, 0]
>>> s[:2] = []
>>> s
[0, 1, 0, 1, 0, 1, 0]
>>> s[::2] = []
...
ValueError: attempt to assign sequence
of size 0 to extended slice of size 4
>>> del s[::2]
>>> s
[1, 1, 1]
```

通过切片修改列表：示例

编写函数，接收一个列表s和一个整数k作为参数，将s中元素**原地循环左移k位**，移出的元素转移到s后部.
要求：不得使用其它列表保存中间值.

假设原列表为：[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]，参数k为5，则处理后列表为：[6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 1, 2, 3, 4, 5]

- 算法思路：将s中下标k之前（不包括）的元素逆序，下标k及其后的元素逆序，最后将整个列表逆序

lshift.py

```
def lshift(s, k):
    s[:k] = reversed(s[:k])
    s[k:] = reversed(s[k:])
    s.reverse()
```

```
def lshift2(s, k):
    s[:k] = s[k-1::-1])
    s[k:] = s[:k-1:-1])
    s.reverse()
```

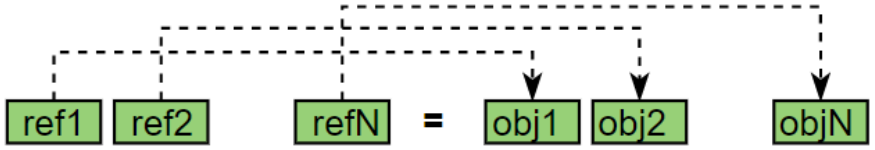
A1A2…Ak	B1B2…Bn
Ak…A2A1	Bn…B2B1
B1B2…Bn	A1A2…Ak

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
[6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 1, 2, 3, 4, 5]

主要内容

- 有序对象：字符串、列表和元组
- 可变有序对象：列表
- 有序对象的切片
- **序列解包**
- 用于序列的常用内置函数：zip和enumerate
- 函数式编程
- 多维列表

序列解包(sequence unpacking)



- 序列解包也称为iterable unpacking，将**可迭代对象**拆分成各个元素，赋值给多个**对象引用**(如变量)
  - 对象引用可以是**变量名**，可以是**属性**，可以是通过**下标**描述的容器对象中的元素，可以是通过**切片**描述的列表中的一个或多个元素
- 赋值语句(LHS=RHS)中左边LHS为通过**元组或列表形式描述的多个变量引用**时，表示序列解包
  - LHS通过圆括号和方括号(不引起歧义时可省略）组织对象引用，通过逗号分割引用
  - RHS可以是任何可迭代对象，包括tuple、list、dict、range、str等，逐个按顺序（从左到右) 取该可迭代对象的元素赋予左边对应位置的对象引用
  - **基本序列解包**，要求左边LHS中的对象引用个数与RHS中的元素**个数相同**
  - **扩展序列解包**，允许RHS的元素个数大于左边对象引用的个数，可使用**带星号的对象引用（*seq）**来收集RHS中的**多余元素**

```
>>> (x, y, z) = (False, 3.5, 'exp')
>>> x, y, z = (False, 3.5, 'exp')
>>> x, y, z
(False, 3.5, 'exp')
```

```
>>> a, b, c = [1, 2, 3]
>>> a, b, c
(1, 2, 3)
>>>[a, b, c] = 'abc'
>>> a, b, c
('a', 'b', 'c')
```

```
>>> a, b, c = range(3)
>>> a, b, c
(0, 1, 2)
>>> a, b, c = range(4)
Traceback (most recent call last):
ValueError: too many values to unpack (expected 3)
```

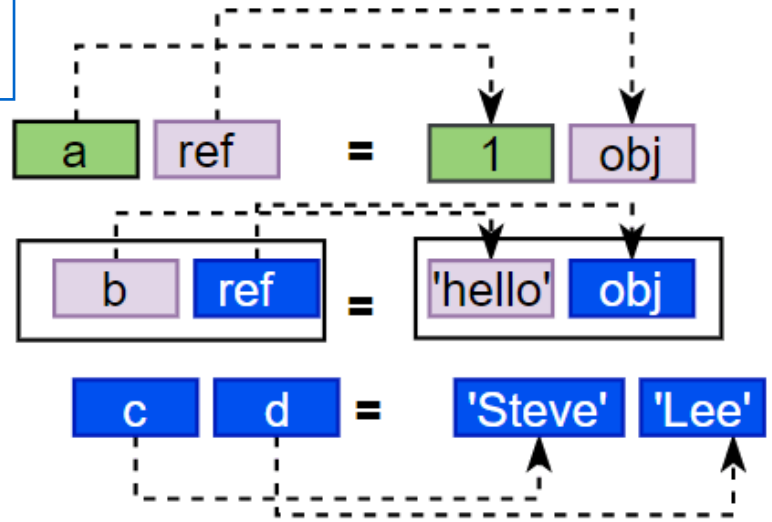
序列解包(sequence unpacking)

- 对象引用可以是变量名，可以是属性，可以是通过下标描述的容器对象中的元素，可以是通过切片描述的列表中的一个或多个元素
- 对象引用为**切片**时，必须与**可迭代对象对应**

```
>>> import math
>>> list1 = list(range(12))
#list1 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> x, math.y, list1[-1], list1[0:5] = 3, 4, 0, range(-5,0)
>>> x, math.y
(3, 4)
>>> list1
[-5, -4, -3, -2, -1, 5, 6, 7, 8, 9, 10, 0]
```

- 序列解包还可以嵌套

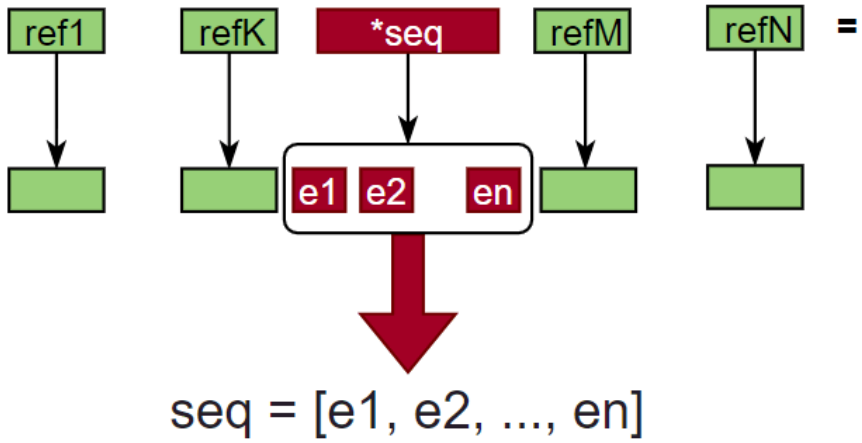
```
>>> a, [b, (c, d)] = 1, ['hello', ('Steve', 'Lee')]
>>> a, b, c, d
(1, 'hello', 'Steve', 'Lee')
```



扩展序列解包

扩展序列解包用于赋值语句LHS=RHS，相比原来的序列解包，

- 引入**带星号的对象引用（*seq）**
- LHS中**最多允许出现一次带星号的对象引用**，该引用前后的变量——对应赋予右边的可迭代对象的元素之后，**剩余的元素都转变为list**然后赋值给该引用seq




```
>>> a, *b, c = range(1, 7)
>>> a, b, c
(1, [2, 3, 4, 5], 6)
```

name, email, phone_numbers = record[0], record[1], record[2:] # 这种方式麻烦，而且要能够支持下标和切片等

```
data = [ 'Tom', 'math', '16010045', '13905750074',(1999, 12, 21) ]
name, major, _, phone, birth = data

record = ('Dave', 'dave@example.com', '773-555-1212', '847-555-1212')
name, email, *phone_numbers = record
print(name, email, phone_numbers)
```

iterable_unpacking.py

序列解包使用实例

问题：任意输入三个英文单词，按字典顺序输出这三个单词

选择排序算法： 首先选择最小的，剩下的里面再选择最小的，直到剩下一个元素为止

```
text = input('请输入三个英文单词，以空格分割==>')
x, y, z = text.split()[:3]
if x > y:
    x, y = y, x
if x > z:
    x, z = z, x
if y > z:
    y, z = z, y
#x, y, z = sorted((x, y, z))
print(x, y, z)
```

words_sort.py

第一个if语句，x保存原来x,y中的最小值
第二个if语句，x保存三个单词中的最小值
第三个if语句，剩下两个单词y,z进行排序，y保留小的

`text.split()[:3]`

根据空格分割后得到一个字符串列表，然后通过切片取前面3个元素

假设用户输入 how are you

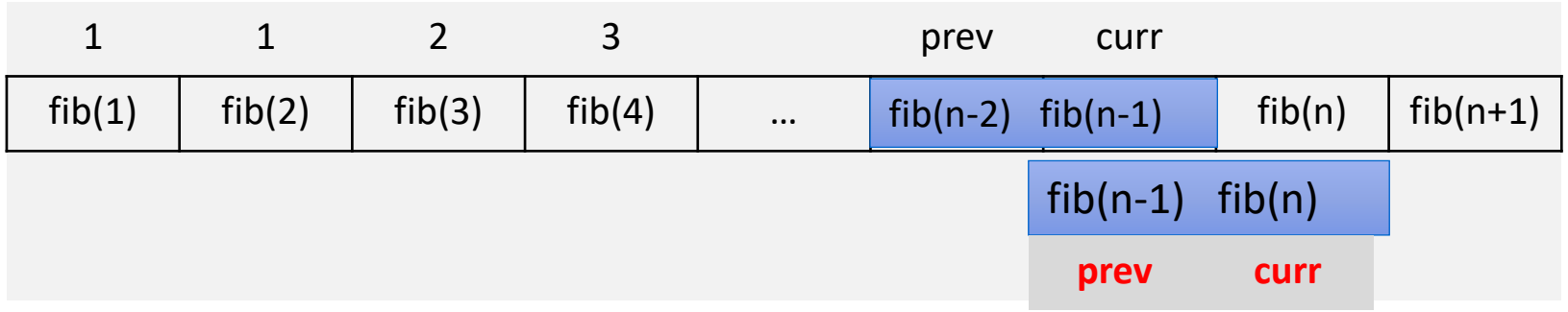
```
x, y, z = ['how', 'are', 'you']
```

x, y = y, x 将变量x和y对调，等价于

```
tmp = x
x = y
y = tmp
```

序列解包使用实例

编写函数，接收整数参数t，返回斐波那契数列中大于t的第一个数



fib.py

```
def fib(t):
    """ 返回大于t的最小fibonacci数 """
    prev, curr = 1, 1
    while curr <= t:
        prev, curr = curr, prev + curr
    return curr
```

序列解包使用实例

编写函数，求两个正整数的最大公约数和最小公倍数

接收两个正整数(比如24,60)作为参数，返回一个元组，其中第一个元素为最大公约数，第二个元素为最小公倍数

gcd.py

```
def gcd1(m, n):
    if m > n:
        m, n = n, m
    p = m * n
    while m != 0:
        r = n % m
        n = m
        m = r
    return (n,int(p / n))
print(gcd1(30, 20))
```

```
def gcd2(m, n):
    p = m * n
    while m != 0:
        m,n = n % m, m
    return n, p // n
```

- 最大公约数(最大公因数或最大公因子) greatest division factor：两个或者多个自然数的共有的因数中最大的一个
- 最小公倍数(Least Common Multiple)：两个或者多个自然数的共有的倍数中除0之外的最小的一个
- 最大公因数和最小公倍数之间的性质：两个自然数的乘积等于这两个自然数的最大公约数和最小公倍数的乘积
- 质因数分解法：把每个数分别分解质因数
  - 最大公约数：把各数中的全部公有质因数提取出来连乘所得的积
  - 最小公倍数：把各数中的全部公有的质因数和独有的质因数提取出来连乘所得的积
- **辗转相除法（欧几里德法）求最大公约数**

  gcd(m,n) = gcd(n%m, m)

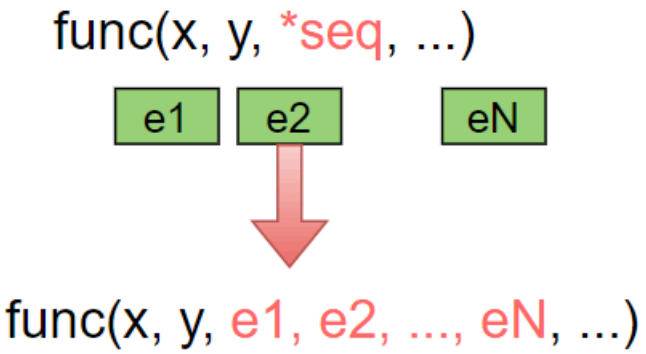
  …

  (m=0, gcd)

序列解包

- 序列解包不仅仅用于**赋值语句**
- 序列解包可用于**函数调用**中 func(x, y, *seq)
  - ***seq** 将可迭代对象seq的各个元素拆分后作为**位置实参**

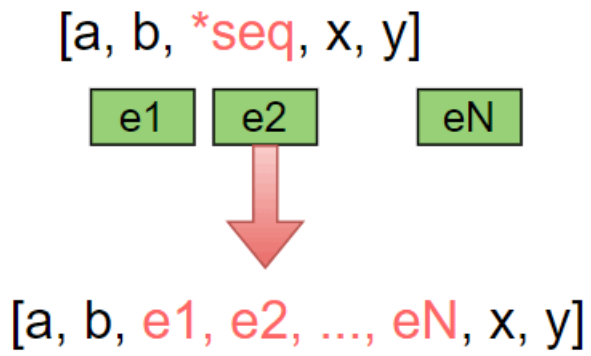
```
s = list(range(10))
print(*s, sep=' + ', end=' = ')
print(sum(s))  # 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45
```



- 序列解包可用于序列**字面量定义**中 (python 3.5引入)
  - *seq 表示将可迭代对象seq的各个元素拆分为多个元素

```
>>> *range(4), 4            # 元组字面量定义
(0, 1, 2, 3, 4)
>>> [*range(4), 4]          # 列表字面量定义
[0, 1, 2, 3, 4]

>>> {*range(4), 4}          # 集合字面量定义
{0, 1, 2, 3, 4}
```



序列解包总结

使用场景	描述
x, y, z = range(3)	**赋值语句**的基本序列解包，支持嵌套序列解包
x, y, z, *seq = range(10)	**赋值语句**的扩展序列解包，支持嵌套序列解包。首先匹配其他位置的对象引用，剩余的元素作为seq所指向的新列表中的元素
func(x, y, *seq)	**函数调用时**的序列解包，将可迭代对象seq的各个元素拆分后作为位置实参
[*range(4), 4]	元组、列表、集合**字面量定义时**的序列解包，将可迭代对象的各个元素拆分后作为新的对象中的元素
func(x, y, *seq, **map)	函数调用时的序列解包，表示将map对象(如字典)的各个元素拆分后作为关键实参 (var=value)
{'x': 1, **{'y': 2}}	字典字面量定义时的序列解包，将map对象的各个元素(key:value)拆分后作为新的字典对象中的元素

```
print(*range(4))                    # 1
k = *range(4)                       # 2 SyntaxError
'%s %s %s %s' % range(4)            # 3 TypeError
'%s %s %s %s' % tuple(range(4))     # 4
'%s %s %s %s' % (*range(4), )       # 5
```

序列解包使用实例

编写程序，判断今天是今年的第几天

1. 首先得到今天对应的年份、月份和当月第几天
2. 今年的第几天=前面的月份天数之和+当月第几天

- 2月的天数随闰年而不同，怎么判断闰年？
  - 能被400整除
  - 或者能被4整除，但不能被100整除

```
>>> import time
>>> time.localtime()
time.struct_time(tm_year=2021, tm_mon=10,
tm_mday=19, tm_hour=20, tm_min=25,
tm_sec=13, tm_wday=1, tm_yday=292,
tm_isdst=0)
```

```
def test_yday():
    import time
    date = time.localtime()
    print('%4d-%02d-%02d是今年的%d天' %
(*date[:3], yday(date)))
```

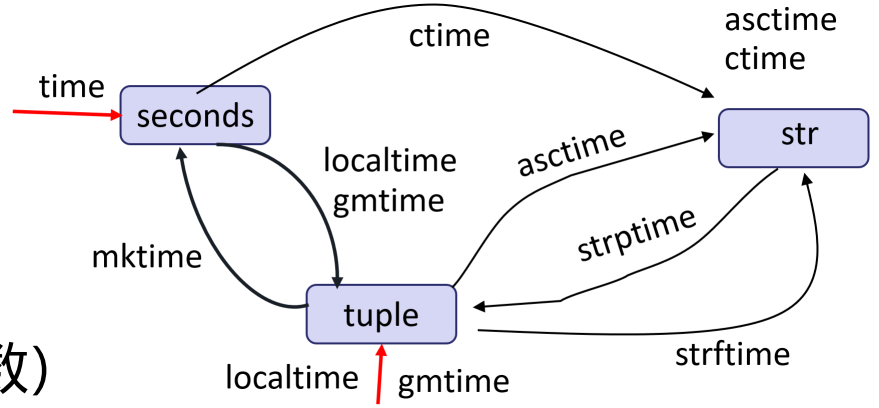
```
def yday(date):
    day_month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)
    year, month, day, *_ = date
    yday_ = sum(day_month[:month-1]) + day
    if month > 2:
        if is_leap_year(year):
            yday_ += 1
    return yday_
```

day_of_year.py

```
def is_leap_year(year):
    return year % 400 == 0 or year % 4 == 0 and year % 100 != 0
```

time模块

拓展的内容



- 得到当前时刻，进行不同格式的时间之间的转换
- 两种格式 + 字符串
  - 从epoch(即标准时1970年1月1日）开始到现在的秒数（浮点数）
  - 包含9个整数的tuple: 年/月/日/小时/分钟/秒/星期几(0-6, 0表示星期一)/当年第几天(1-366)/是否夏时制

time()	**返回从epoch开始到现在的秒数**
localtime([seconds])	将从epoch开始的秒数格式转换为本地时间的tuple格式，缺省为现在
gmtime([seconds])	将从epoch开始的秒数格式转换为UTC时间的tuple格式，缺省为现在
ctime([seconds])	将从epoch开始的秒数按照系统缺省方式转换为字符串，缺省为现在
mktime(tuple)	将本地时间的tuple格式转换为秒数
asctime([tuple])	将本地时间的tuple格式按系统缺省方式转换为字符串，缺省为现在，即相当于传递参数localtime()
strftime(format[,tuple])	按照format描述的格式将本地时间的tuple格式转换为字符串，缺省为现在
strptime(string, format)	按照format所描述的格式分析字符串string中的时间，返回tuple格式
sleep(seconds)	睡眠指定的秒数(浮点数)

time模块

拓展的内容

```
>>> import time
>>> start = time.time()
>>> start
1585987981.4362593
>>> stop = time.time()
>>> stop - start
190.97639966011047
>>> time.localtime()
time.struct_time(tm_year=2020, tm_mon=4, tm_mday=4, tm_hour=16, tm_min=13,
tm_sec=21, tm_wday=5, tm_yday=95, tm_isdst=0)
>>> time.gmtime()
time.struct_time(tm_year=2020, tm_mon=4, tm_mday=4, tm_hour=8, tm_min=13, tm_sec=31,
tm_wday=5, tm_yday=95, tm_isdst=0)
>>> start_tuple = time.localtime(start)
>>> time.mktime(start_tuple)
1585987981.0
>>> time.asctime(start_tuple)
'Sat Apr  4 16:13:01 2020'
>>> time.ctime(start)
'Sat Apr  4 16:13:01 2020'
```

time模块

datetime模块在time模块的基础上提供日期时间等信息的访问，支持时间运算

拓展的内容

格式化字符	描述	格式化字符	描述
%Y	四位年份	%a	星期几的简写，如Wed
%m	月份[1,12]	%A	星期几的全名，如Wednesday
%d	当月第几天	%b	月份的简写
%H	小时[0,23]	%B	月份的全名
%I	小时[1,12]	%c	缺省的日期和时间格式
%M	分钟[0,59]	%u	星期几[1,7]
%S	秒数[0,59]	%w	星期几[0,6]
%p	上午或下午描述 PM/AM	%U	今年的第几个星期 [0, 53]
%z	包含时区偏移	%j	今年的第几天[1,366]
%s	距离Epoch的秒数		

```
>>> now = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())
>>> now
'2018-04-09 23:20:26‘
>>> time.strptime(now, "%Y-%m-%d %H:%M:%S")
time.struct_time(tm_year=2018, tm_mon=4, tm_mday=9, tm_hour=23, tm_min=20, tm_sec=26,
tm_wday=0, tm_yday=99, tm_isdst=-1)
```

主要内容

- 有序对象：字符串、列表和元组
- 可变有序对象：列表
- 有序对象的切片
- 序列解包
- **用于序列的常用内置函数：zip和enumerate**
- 函数式编程
- 多维列表

用于序列对象的常用内置函数

- len(seq)：返回容器对象seq中的元素个数，适用于列表、range、元组、字典、集合、字符串等
- max和min：两种语法
  - max(arg1, arg2, *args, key=func)、min(arg1, arg2, *args, key=func): 传递的参数中的最大或最小值
  - max(iterable, default=obj, key=func)、min(iterable default=obj, key=func)：返回迭代对象中的最大或最小**元素**，如果为空，返回obj
- sum(iterable, start=0)：对**数值型可迭代对象**的元素进行求和运算，最后加上start（缺省为0）。元素为非数值型时抛出异常TypeError

```
>>> s = [1, 4, -5, -7, 6]
>>> len(s)
5
>>> max(1, 4, -5, -7, 6)
6
>>> min(1, 4, -5, -7, 6)
-7
>>> max(s)
6
>>> min(s)
-7
>>> max([1,2], [2,4])
[2, 4]
>>> sum(s)
-1
```

用于序列对象的常用内置函数:zip

- zip(iter1,iter2,...):返回一个**zip对象，**该对象是一个迭代器(iterator)对象
- 传递的参数为可迭代对象(列表、字符串、元组、range等，甚至包括字典)
- 每次调用next(zip_object)会返回一个元组，该元组的元素为各个参数中对应位置的对象，第一次为所有参数的第一个元素，第二次为所有参数的第二个元素...
- 如果这些可迭代对象的长度不一致，则该迭代器的元素个数为**这些长度的最小值**

```
>>> aList = [1, 2, 3]
>>> bList = [4, 5, 6]
>>> cList = zip(a, b)
>>> cList
<zip object at 0x0000000003728908>
>>> list(cList)
 [(1, 4), (2, 5), (3, 6)]
>>> list(cList)          # why???
[]
```

zip函数的参数为可迭代对象

itera	iterb	iterc
itera_1	iterb_1	iterc_1
itera_2	iterb_2	iterc_2
itera_3	iterb_3	iterc_3
itera_4		iterc_4
itera_5		

返回可迭代的zip对象

zip(itera, iterb, iterc)
itera_1, iterb_1, iterc_1
itera_2, iterb_2, iterc_2
itera_3, iterb_3, iterc_3

zip对象可以进一步用list/tuple等函数转换为列表/元组对象

zip(iter1,iter2,...)返回的**zip对象，**是一个迭代器(iterator)对象

- 调用list(...)后将zip对象中的元素取完了
- 再次调用时还是试图取zip对象的下一个元素

zip实例：同时对多个序列进行迭代

```
for item in zip(xpts, ypts):
    x, y = item
```

```
for x, y in zip(xpts, ypts):
    pass
```

- 每次取zip对象的下一个元素，即来自于xpt和ypts中对应位置的元素组成的元组，假设为 xpts_i, ypts_i
- 序列展开: x, y = xpts_i, ypts_i

zip_enumerate.py

```
xpts = (1, 5, 4, 2, 10, 7)
ypts = [101, 78, 37, 15, 62, 99, 100]
for x, y in zip(xpts, ypts):
    print(x, y, sep='\t')
```

```
1	101
5	78
4	37
2	15
10	62
7	99
```

itertools模块包含了许多迭代器，其中zip_longest(iter1, iter2, ..., fillvalue=None) 返回zip_longest对象

- 与zip类似，只是迭代器的元素个数为所有可迭代对象参数的元素个数的最大值
- 如果某个可迭代对象参数没有更多的元素了，相当于下一个元素为fillvalue，缺省为None

```
import itertools
s1 = 'abcde'
s2 = list(range(3))
for x, y in itertools.zip_longest(s1, s2, fillvalue=0):
    print(x, y, sep='\t')
```

```
a	0
b	1
c	2
d	0
e	0
```

用于序列操作的常用内置函数:enumerate

- enumerate(iterable[, start])：返回enumerate(枚举)对象，是一个迭代器对象，其每个元素为包含下标和对应可迭代对象的元素的元组。第一个元素的下标从start开始，缺省为0

zip_enumerate.py

```
d_list = [5, 6, 7]
for item in enumerate(d_list):
    print(item)

for idx, value in enumerate(d_list, 1):
    print(idx, value)

data = zip([1, 3, 5], [2, 4, 6])
# Error!
# for n, x, y in enumerate(data):
for n, (x, y) in enumerate(data):
    print(n, ':', x, y)
```

```
(0, 5)
(1, 6)
(2, 7)

1 5
2 6
3 7

0 : 1 2
1 : 3 4
2 : 5 6
```

```
for i in range(len(d_list)):
    print(i, d_list[i])
```

可迭代对象iter1

0	e1
1	e2
2	e3
3	e4
4	e5

enumerate(iter1)

0, e1
1, e2
2, e3
3, e4
5, e5

enumerate(iter1, start) 从0开始的下标+start

idx,val = (index, value) #序列解包

```
for n, (x, y) in enumerate(data): pass
# enumerate(data)的元素为: index, (iter1_i, iter2_i)
# 采用嵌套序列解包:
# n, (x, y) = index, (iter1_i, iter2_i)
```

enumerate使用示例

- 编写函数，参数为多个整数组成的有序对象，查找其中的最小值及其下标(可能有多个)。返回一个元组，其中第一个元素为最小值，其余元素为最小值在有序对象中的下标（最小值出现的位置）

index_min.py

```
import random
def index_min(s):
    m = min(s)
    result = (m,)
    for index, value in enumerate(s):
        if value == m:
            result = result + (index,)
    return result
```

```
x = [random.randint(1, 20) for i in range(50)]
print(x)
print(index_min(x))
print(index_min2(x))
```

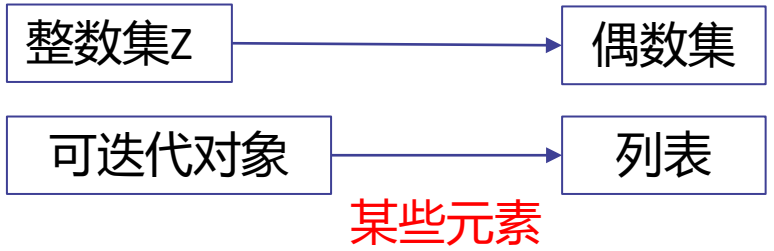
```
def index_min2(s):
    min_value = min(s)
    result = [min_value]
    for index, value in enumerate(s):
        if value == min_value:
            result.append(index)
    return tuple(result)
```

建议使用list，而不是tuple来保存中间结果

主要内容

- 有序对象：字符串、列表和元组
- 可变有序对象：列表
- 有序对象的切片
- 序列解包
- 用于序列的常用内置函数：zip和enumerate
- **函数式编程：**
  - 列表解析式
  - 生成器表达式
  - map/filter/reduce
  - 生成器函数
- 多维列表

列表解析式（list comprehension）



- 集合论：{ x | p(x)}
  - 偶数 {x | x = 2k, k ∈Z}
  - 平方数 {x | x= k ** 2, k ∈Z} 或者 { x ** 2 | x ∈Z}
- 列表推导式/解析式（list comprehension）是利用其他可迭代对象(比如列表)中的各个元素或者某些元素创建新列表的一种方法，非常简洁，代码具有强可读性

```
[expr for value in iterable]                    [2*k for k in range(100)]
[expr for value in iterable if condition]       [2*k for k in range(100) if k % 10 == 0]
```

基本要素：

- **可迭代对象**中的元素，通过**表达式进行进一步运算**，对应新列表中的元素
- if子句**过滤掉**可迭代对象中的一些元素
- **列表推导式中引入的变量相当于本地变量，仅在列表推导式中有意义**

```
s = [x * x for x in range(10)]
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
s = [x * x for x in range(10) if x % 2]
# [1, 9, 25, 49, 81]
```

```
s = []
for x in range(10):
    s.append(x * x)
```

```
s = []
for x in range(10):
  if x % 2:
      s.append(x * x)
```

列表解析式：复杂表达式

列表解析式中的表达式可是函数调用、元组列表等字面量定义、复杂表达式，还可以是列表解析式

- **函数调用**

```
>>> vec = [-1, -4, 6, 7.5, -2.3, 9, -11]
>>> [abs(x) for x in vec]
[1, 4, 6, 7.5, 2.3, 9, 11]
```

- **元组定义，每个元素为(数，数的平方)**

```
>>> [(x, x ** 2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

- if else**三元表达式**

```
>>> [v ** 2 if v % 2 == 0 else v + 1 for v in [2, 3, 5, 4, -1] if v > 0]
[4, 4, 6, 16]
```

- 列表解析式

```
>>> [[i * j for j in range(1, 4)] for i in range(1, 4)]
[[1, 2, 3], [2, 4, 6], [3, 6, 9]]
```

列表解析式：多个for/if子句

语法： [expr for value in iterable] 或 [expr for value in iterable if condition]

第一个for(加上可选的if)之后可以跟0或多个for(加上可选的if)

```
[expr for value1 in iterable1 if condition1
        for  value2 in iterable2  if condition2
               …
        for  valueN in iterableN if conditionN]
```

从一个迭代对象中取value1，在此基础上从第二个迭代对象中取value2，如此...从最后一个迭代对象中取valueN，这些value进行运算作为新列表中的元素

- expr: 可以使用所有for变量
- 后面的for子句和if子句可以使用前面的for变量

产生包含了所有两位偶数的列表

```
s1 = list(range(10, 100, 2))
s2 = [10 * x + y for x in range(1, 10)
                 for y in range(0, 10, 2)]
s3 = [10 * x + y for x in range(0, 10) if x != 0
                 for y in range(10) if y % 2 == 0]
```

```
s4 = [ ]  # s3对应的循环实现
for x in range(0, 10):
    if x != 0:
        for y in range(10):
            if y % 2 == 0:
                s4.append(10 * x + y)
```

列表解析式：素数

判断p是否为素数

- 一个数是素数，除了1和自身外，没有其他因子
- 首先得到p除以可能的因子（[2, sqrt(n)]） 的余数组成的列表
- 如果其中有0，说明不是素数。如果都不为0，则为素数

```
prime = 0 not in [p % d for d in range(2, int(math.sqrt(p) + 1))]
```

使用列表推导式生成100以内的所有素数：

```
>>> import math
>>> [p for p in range(2, 100) if 0 not in
                [p % d for d in range(2, int(math.sqrt(p) + 1))]]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79,
83, 89, 97]
```

生成器表达式

- **生成器表达式(Generator Expression)**的语法与列表推导式基本相同，唯一的区别就是生成器推导式使用**圆括号**，列表推导式使用方括号
- 生成器表达式运算的结果并不是一个元组，而是一个生成器对象
- 生成器对象是一个**迭代器对象**，当然也是一个可迭代对象，可用for循环访问，也可转换为列表等

```
>>> [x * x for x in range(4)]
[0, 1, 4, 9]
>>> g = (x * x for x in range(4))
>>> g
<generator object <genexpr> at 0x00FFADB0>
>>> dir(g)
[...'__iter__',..., '__next__',...]
>>> next(g)
0
>>> next(g)
1
>>> next(g)
4
```

```
>>> next(g)
9
>>> next(g)
Traceback (most recent call last):
  File "#python..", line 1, in <module>
    next(g)
StopIteration
>>> iter(g) is g
True
>>> list(g)
[]
>>> list(x * x for x in range(4))
[0, 1, 4, 9]
```

生成器表达式

勾股数 (pythogorian triplets)：存在{x,y,z}, 0<x<y<z, 使得x^2+y^2=z^2

求前10个勾股数

pyt_firstN.py

```
def pyt_firstN_list(nums):
    pyt = [(x,y,z) for z in range(1, 100)
           for y in range(1,z)
           for x in range(1,y) if x*x + y*y == z*z ]
    firstN_pyt = pyt[:nums]
    print(*firstN_pyt)
```

循环次数为o(n^3)， n从100改成1000?

```
pyt = [(x,y,z) for z in range(1, 1000) for y in range(1,z)
                                       for x in range(1,y) if x*x + y*y == z*z ]
```

```
def pyt_firstN_generator(nums):
    pyt = ((x,y,z) for z in range(1, 10000)
           for y in range(1, z)
           for x in range(1, y)
              if x*x + y*y == z*z )
    firstN_pyt = [next(pyt) for x in range(nums)]
    print(*firstN_pyt)
```

内置函数map和filter

在Python语言的发展过程中，是首先有map/filter等内置函数，后来引入列表解析式和生成器表达式

`map(func, iterable) filter(func, iterable)`

- 将一个函数作用到一个可迭代对象上，返回一个**迭代器对象**
- map函数返回的迭代器的元素是对iterable对象的元素调用相应函数func的结果，即每个元素为func(element)
- filter函数返回的迭代器的元素是这样得到的：
  - iterable对象的元素作为参数调用func，如果func(element)返回True，则该元素为迭代器的元素
  - 如果filter函数的第1个参数为None，则iterable对象的元素真值判断为True时作为迭代器的元素

```
map(func, iterable)    <======>    (func(i) for i in iterable)
filter(func, iterable) <======>    (i for i in iterable if func(i))
filter(None, iterable) <======>    (i for i in iterable if i)
```

```
>>> list(map(str, range(5)))
['0', '1', '2', '3', '4']
>>> s = 1, 2, '', 0, -1, []
>>> list(filter(None, s))
[1, 2, -1]
```

```
>>> pi_digits = 3, 1, 4, 1, 5, 9, 2, 6, 5
>>> def is_odd(x):
        return x % 2

>>> list(filter(is_odd, pi_digits))
[3, 1, 1, 5, 9, 5]
```

lambda表达式

- lambda表达式用来定义一个仅包含一行代码的匿名函数，运算结果为一个匿名**函数对象**
  - 没有函数名字，不需要额外的取名烦恼（可能重复），避免了可能的名字冲突
  - 函数体只包含一个表达式，且该表达式的计算结果为函数的返回值
    - 只能是表达式，但注意函数调用也是表达式，因此可以调用print、 map、 list等。
    - 不允许包含if/for/while等其他复杂的语句
- lambda表达式也是函数的一种，遵循函数的参数定义，可以使用缺省值参数、可变长度参数等
- 遵循函数的传递规则，可以使用位置参数，也可使用关键字参数传递来调用该函数
- 遵循函数的作用域规则，参数为局部变量

```
def  __anonymous_function__(arguments):
    return expr
```

```
lambda arguments: expr
```

```
def is_odd(x):
    return x % 2
pi_digits = 3, 1, 4, 1, 5, 9, 2, 6, 5
```

```
s1 = list(filter(is_odd, pi_digits))
s2 = list(filter(lambda x: x % 2, pi_digits))
# [3, 1, 1, 5, 9, 5]
```

- lambda表达式的优先级最低，大部分情况下expr不用加上括号，但注意如果表达式中有**省略了圆括号的元组定义时**，容易出现问题。 比如f = **lambda** x=3: x, 0

reduce

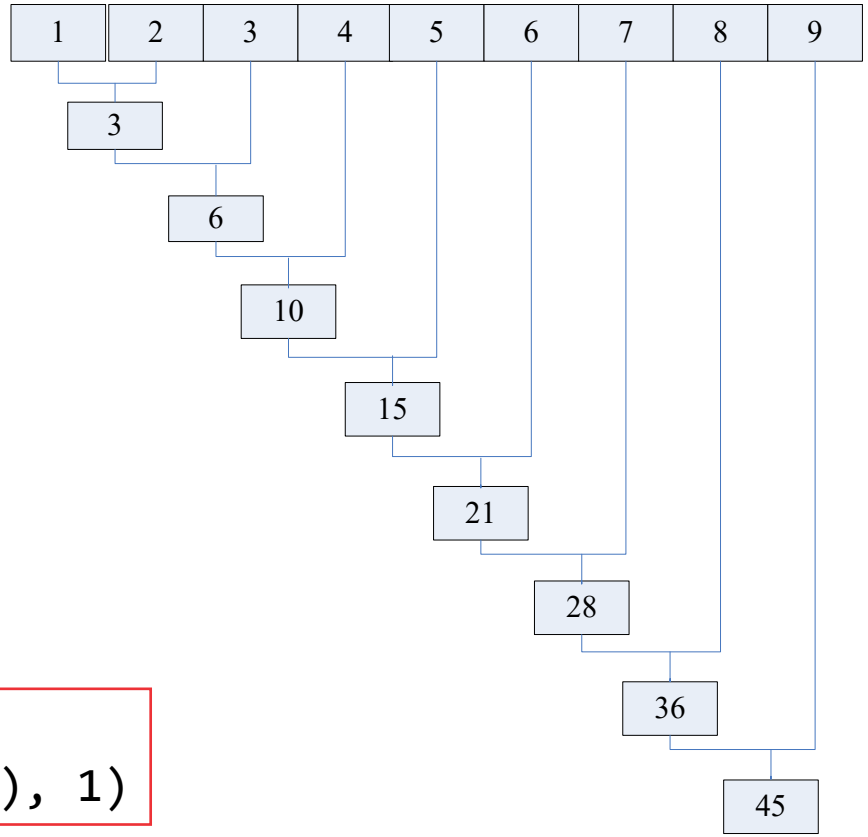
```
>>> reduce(lambda x,y:x+y, [], 0)
0
```

拓展的内容

模块functools的reduce(func, iterable [,initial)

- 将可迭代对象的所有元素进行归约，将它们归并在一起
- func的第一个参数为前面规约的结果，第二个参数为当前元素，返回将当前元素归约后的结果
- 如果只有2个参数：初始归约结果为第1个元素，从第2个元素开始调用func(sum, element)，直到最后一个元素
- 如果传递了initial，则初始规约结果为initial

```
>>> from functools import reduce
>>> seq=[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> def add(x, y):
        return x + y
>>> reduce(add, range(10))
45
>>> reduce(lambda x,y:x + y, seq)
45
>>> reduce(add, map(str, range(10)))
'0123456789'
```




```
def factorial(n):
    return reduce(lambda x, y: x * y, range(1, n + 1), 1)
```

map和reduce示例

拓展的内容

- 有多个字符串组成的可迭代对象，找出其中最长单词的长度
- 有多个字符串组成的可迭代对象，找出其中最长的单词

longest.py

```
import string
import random
from functools import reduce

num_words = random.randint(1, 100)
words = [ ''.join(random.choices(string.ascii_letters, k=random.randint(0, 30))) for i in
range(num_words)]
print('%d words' % len(words))

max_len = reduce(max, map(len, words))
max_len = max(map(len, words))

longest = reduce(lambda word1, word2: word1 if len(word1) > len(word2) else word2, words)
print(longest)
```

生成器函数

拓展的内容

- 生成器表达式：（x ** 2 for x in range(10))
  - 不是一次返回全部数据，而是一个iterator对象，通过next内置函数获得下一个元素
- 生成器函数可创建生成器对象
  - **生成器函数**是函数的一种，普通函数原来通过return语句返回值，现在通过yield expression返回值
  - 调用生成器函数的结果是得到一个生成器对象
  - 每次调用next时得到一个元素，开始执行生成器函数函数体或从上次暂停处恢复执行
    - 执行生成器函数中给出的代码直到遇到yield表达式或者从生成器函数中返回
    - 如果遇到yield表达式，暂停生成器函数的执行，并返回yield表达式所给出的表达式
    - 语句执行到从生成器函数中返回，则抛出异常StopIteration
    - **恢复执行时，yield表达式的结果为None**

```
def f0(iterable):
    s = []
    for item in iterable:
        s.append(item ** 2)
    return s
```

generate_func.py

```
def f(iterable):
    for item in iterable:
        yield item ** 2

g = f(range(10))
print(g)
print(next(g))
print(next(g))
print(next(g))
```

生成器函数：斐波那契数列

- fib(n): 输出小于n的斐波那契数
- fib2(n): 产生斐波那契数列

fib_generator.py

```
def fib(n):
    prev, curr = 1, 1
    while prev < n:
        print(prev, end=' ')
        prev, curr = curr, prev + curr
    print()
```

```
def fib2():
    prev, curr = 1, 1
    while True:
        yield prev
        prev, curr = curr, prev + curr
```

返回前10个数： 1 1 2 3 5 8 13 21 34 55

```
a = fib2()
for i in range(10):
    print(next(a), end=' ')
print()
```

返回1000以内的fibonacci数

```
count = 0
for i in fib2():
    if i > 1000:
        break
    print(i, end='\t')
    count += 1
    if count % 5 == 0 :
        print()
if count % 5:
    print()
```

```
1       1       2       3       5
8       13      21      34      55
89      144     233     377     610
987
```

生成器函数

拓展的内容

- yield from iterable：调用next时返回可迭代对象的下一个元素
- itertools模块中包含了多个迭代器，包括count/cycle/repeat/chain等

generate_func.py

```
def f1():
    yield from 'welcome to the python world.'

def f2():
    for i in 'welcome to the python world.':
        yield i

def count(start=0, step=1):
    while True:
        yield start
        start += step

for i in count():
    if i % 10 == 0:
        print(i)
    if i > 100:
        break
```

生成器函数：调用者和生成器之间的交互

拓展的内容

生成器对象的其他方法可以进行更多的交互： **feedback = yield expr**

- __next__(): 调用者不传递对象给生成器， **恢复执行时yield expr的结果**为None
- send(value): 调用者传递value给生成器，**恢复执行时yield expr的结果**为value。生成器函数体还没有执行时调用send开始执行，只能传递None，即send(None)
- throw(type, value, traceback)，后面两个参数可选，**恢复执行时yield expr**会抛出类型为type的异常
- close()，**恢复执行时yield expr**会抛出GeneratorExit异常，生成器在捕获该异常时应该结束

generate_func.py

```
import random
def producer():
    while True:
        value = random.randint(0, 100)
        print('\n[PRODUCER]: produce', value)
        try:
            feedback = yield value
            print('[PRODUCER]: got feedback', feedback)
        except GeneratorExit:
            print('[PRODUCER]: done')
            break
        except Exception as e:
            print('[PRODUCER]: caught exception', type(e), e)
```

```
def consumer(p):
    value = p.send(None)
    for i in range(5):
        print('[CONSUMER]: got ', value)
        feedback = '奇数' if value % 2 else '偶数'
        value = p.send(feedback)
    print('[CONSUMER]: got ', value)
    value = p.throw(TypeError, 'spam')
    print('[CONSUMER]: got ', value)
    p.close()
p = producer()
consumer(p)
```

主要内容

- 有序对象：字符串、列表和元组
- 可变有序对象：列表
- 有序对象的切片
- 序列解包
- 用于序列的常用内置函数：zip和enumerate
- 函数式编程：
  - 列表解析式
  - 生成器表达式
  - map/filter/reduce
  - 生成器函数
- **多维列表**

多维列表

- 可以通过列表的列表来表示二维矩阵(matrix)： matrix = [row1, row2, ... row_n]
  - matrix的元素为每一行对应的列表，该列表中的元素为该行的各列
  - len(matrix)为矩阵的行数，而len(matrix[0])为矩阵的列数
  - 通过多级下标(行下标和列下标)访问矩阵中的元素，matrix[i][j]表示第i+1行和第j+1列的元素

```
matrix = [
 [1, 2, 3, 4, 5],
 [6, 7, 0, 0, 0],
 [0, 1, 0, 0, 0],
 [1, 0, 0, 0, 8],
 [0, 0, 9, 0, 3],
]
```

	[0]	[1]	[2]	[3]	[4]
[0]	1	2	3	4	5
[1]	6	7	0	0	0
[2]	0	1	0	0	0
[3]	1	0	0	0	8
[4]	0	0	9	0	3

matrix[0]→ [1, 2, 3, 4, 5]
matrix[1]→ [6, 7, 0, 0, 0]
matrix[2]→ [0, 1, 0, 0, 0]
matrix[3]→ [1, 0, 0, 0, 8]
matrix[4]→ [0, 0, 9, 0, 3]

matrix[0][0] → 1
matrix[4][4] → 3

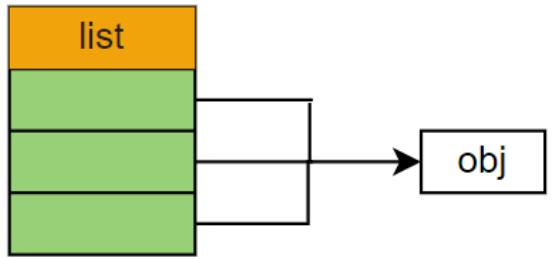
矩阵初始化

- **产生指定维度的随机矩阵**

matrix_usage.py

```
import random
def generate_random_matrix(rows=3, cols = 3):
    matrix = [] # Create an empty list
    for row_num in range(rows):
        row = []
        for col_num in range(cols):
            row.append(random.randint(0, 100))
        matrix.append(row)
    return matrix
```

```
def generate_3d_array(size1, size2, size3):
    array = []
    for i in range(size1):
        array2 = []
        for j in range(size2):
            array3 = []
            for k in range(size3):
                array3.append(random.randint(0, 100))
            array2.append(array3)
        array.append(array2)
    return array
```




```
matrix = [[random.randint(0, 100) for i in range(cols)] for i in range(rows)]
```

- **产生全0矩阵**

```
def generate_zeros_matrix(rows=3, cols=3):
    """ 构造一个全0矩阵
    """
    matrix = [[0] * cols for i in range(rows)]
    # matrix = [[0] * cols] * rows
    return matrix
```

```
# 错误的代码
matrix = [[0] * 4] * 3

row = [0] * 4
matrix = [row, row, row]
```

二维矩阵的访问

- 显示矩阵

```
def print_matrix(matrix):
    for row in matrix:
        for value in row:
            print(value, end=" ")
        print() # Print a new line
```

```
# 嵌套列表的平铺，即新列表由最内层的列表元素组成
>>> matrix = [ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> [col for row in matrix for col in row]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

```
def print_matrix_2(matrix): # 通过下标访问
    for row_num in range(len(matrix)):
        for col_num in range(len(matrix[row_num])):
            print(matrix[row_num][col_num], end=" ")
        print() # Print a new line
```

- 获得矩阵的第i行： matrix[i - 1]
- 获得矩阵的第j列：

```
def get_column(matrix, column=0):
    vec = []
    for row in matrix:
        vec.append(row[column])
    return vec
```

```
>>> [row[1] for row in matrix] # 第2列
[2, 6, 10]
```

二维矩阵的转置

- 矩阵的转置：把mxn阶矩阵(即m行n列)的行和列互换，形成n*m阶的转置矩阵。即原矩阵的行变为转置矩阵的列，原矩阵的列变为转置矩阵的行。
- 采用zip函数实现

zip(*matrix)函数调用的序列解包，可迭代对象matrix的元素展开为位置实参，相当于 zip(matrix[0], matrix[1], …, matrix[len(matrix)-1])

```
matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
matrix_transpose = list(zip(*matrix))
print(matrix_transpose)
matrix_transpose2 = [list(item) for item in matrix_transpose]
matrix_transpose3 = [list(item) for item in zip(*matrix)]
return matrix_transpose3
```

```
[1,   2,   3,   4],
[5,   6,   7,   8],
[9,   10,  11,  12]
```

```
[1,  5,  9 ],
[2,  6,  10],
[3,  7,  11],
[4,  8,  12]
```

```
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

- 使用列表解析式的嵌套实现矩阵转置

```
>>> matrix = [ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> [[row[i] for row in matrix] for i in range(len(matrix[0]))]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

表达式为一个列表推导式
[row[i] for row in matrix]

0	1	2	3
[row[0] for row in matrix]	[row[1] for row in matrix]	[row[2] for row in matrix]	[row[3] for row in matrix]

锯齿状列表：示例

- 杨辉三角形, n表示行数

yanghui_triangle.py

```
def yanghui_triangle(n):
    if not isinstance(n, int) or n < 1:
        raise ValueError('A positive integer is expected.')
    triangle = [[1]]
    if n == 1:
        return triangle
    line = []
    for i in range(1, n):
        new_line = [1]
        for j in range(0, len(line) - 1):
            new_line.append(line[j] + line[j + 1])
        new_line.append(1)
        triangle.append(new_line)
        line = new_line
    return triangle
```

下一行相比上一行多了一个数，最前面和最后为1，中间的各项都是上一行中连续两项之和

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1]
[1, 5, 10, 10, 5, 1]
[1, 6, 15, 20, 15, 6, 1]
[1, 7, 21, 35, 35, 21, 7, 1]
[1, 8, 28, 56, 70, 56, 28, 8, 1]
[1, 9, 36, 84, 126, 126, 84, 36, 9, 1]
```