東南大學 SOUTHEAST UNIVERSITY 人工智能学院

Python编程

授课教师：王洪松

邮箱：hongsongwang@seu.edu.cn

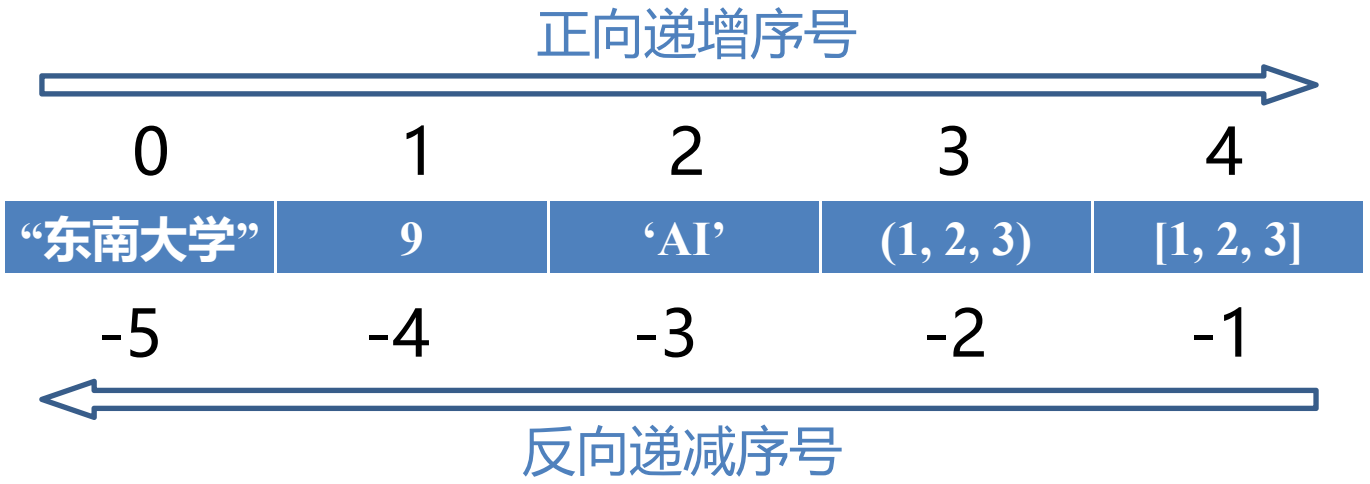
Python语法

- 使用缩进来定义代码块，使用换行符来表示语句的结束
- 变量在赋值时自动声明，不需事先声明变量名及其类型
- 对象是Python语言中最基本的概念
- Python没有自增自减运算符，python有//, **, 没有++和--
- id(object)函数是返回对象object在其生命周期内位于内存中的地址
- is 比较的是两个实例对象的内存地址是否相同，==比较它们的数值是否相同
- Python采用基于值的内存管理方式，如果为不同变量赋予相同值，这个值在内存中只有一份，多个变量指向同一块内存地址
- Python变量名对英文字母的大小写敏感

Python序列结构
列表

Python序列结构

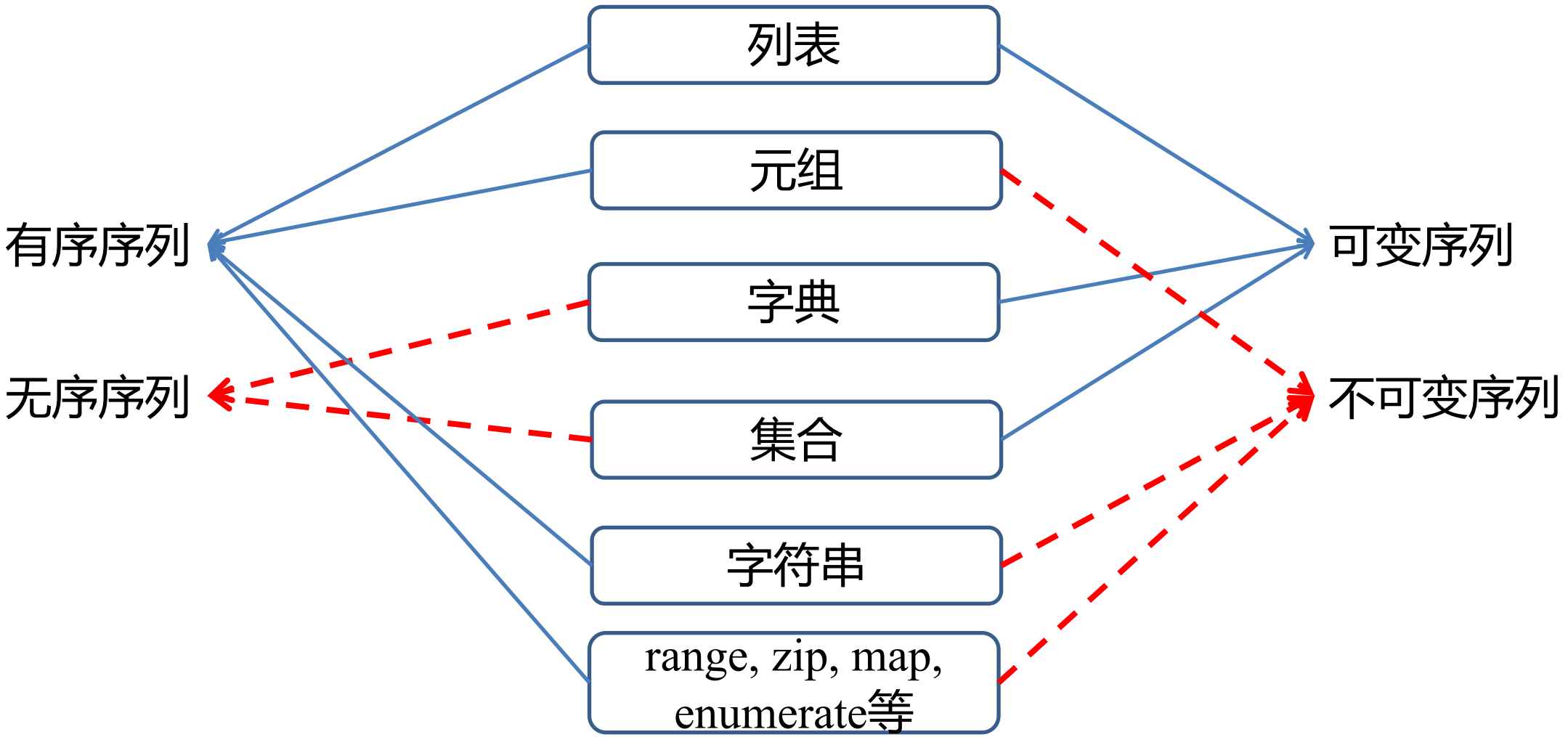
- Python序列类似于其他编程语言中的数组，但支持更多功能
- Python中最基本的序列结构
  - 列表、元组、 字典（映射）、集合、字符串、range等对象
- 列表、元组、字符串支持下标索引
  - 第一个元素下标为0，第二个元素下标为1，以此类推
  - 支持双向索引
    - **最后一个元素-1，倒数第二个元素下标为-2，以此类推**



Python序列结构

	列表	元组	字典	集合
类型名称	list	tuple	dict	set
定界符	[]	()	{}	{}
是否可变	是	否	是	是
是否有序	是	是	否	否
是否支持下标	是（序号）	是（序号）	是（键）	否
元素分隔符	逗号,	逗号,	逗号,	逗号,
元素形式限制	无	无	键: 值	可哈希
元素取值限制	无	无	“键”可哈希	可哈希
元素可重复	是	是	“键”不重复 “值”可重复	否
元素查找速度	非常慢	慢	非常快	非常快
增加和删除元素速度	尾部操作快 其他位置慢	不允许	快	快

Python序列结构



Python序列结构

- 列表概述
  - 列表是Python中内置有序、可变序列，列表的所有元素放在一对中括号[]中，并使用逗号分隔开
  - 当列表元素增加或删除时，列表对象自动进行扩展或收缩内存，保证元素之间没有空隙
  - Python中一个列表中的数据类型可以不同，可分别为整数、浮点数、字符串等基本类型，也可以是列表、元组、字典以及其他自定义类型的对象

```
#coding=utf-8
list_1 = [1, 2, 3, 4, 5]
list_2 = ['a', 'b', 'c', '东南', '大学']
list_3 = [1, 'a', [1, '东南'], '大学']
```

Python序列结构

- 列表常用方法

方法	说明
list.append(x)	将元素x添加至列表list尾部
list.extend(L)	将列表L中所有元素添加至列表list尾部
list.insert(index, x)	在列表list指定位置index处添加元素x，该位置后面的所有元素后移一个位置
list.remove(x)	在列表list中删除首次出现的指定元素x，该元素之后的所有元素前移一个位置
list.pop([index])	删除并返回列表list中下标为index（默认-1）的元素
list.index(x)	返回列表list中第一个值为x的元素的下标，若不存在x抛异常
list.clear()	删除列表list中所有元素，但保留列表对象
list.count(x)	返回指定元素x在列表list中的出现次数
list.reverse()	对列表list所有元素进行逆序
list.sort(key=None, reverse=False)	对列表list所有元素排序，key用于指定排序依据，reverse决定升序(False)还是降序(True)，不可比较则抛出异常
list.copy	返回列表list的浅复制

Python序列结构

- 列表的创建
  - 使用“=”将一个列表赋值给变量
  - 使用list()函数将元组、range对象等转换为列表
- 列表的删除
  - 当不需要某个列表时，使用del命令删除整个列表

```
list_1 = [1, 3, 5, 7, 9]
#[1, 3, 5, 7, 9]
list_2 = list((1, 3, 5, 7, 9))
#[1, 3, 5, 7, 9]
list_3 = list(range(1, 10, 2))
#[1, 3, 5, 7, 9]
del list_2
print(list_2)
```

```
Traceback (most recent call last):
  File "C:/Users/xlkong/PycharmProjects/training/d1.py", line 8, in <module>
    print(list_2)
NameError: name 'list_2' is not defined
```

Python序列结构

- range对象
  - range(start, stop[, step])
    - start: 计数从 start 开始。默认是从 0 开始
      - 例如：range(5)等价于range（0, 5）
    - stop: 计数到 stop 结束，但不包括 stop
      - 例如：range(0, 5)是[0, 1, 2, 3, 4]没有5
    - step：步长，默认为1。
      - 例如：range(0, 5)等价于 range(0, 5, 1)
  - range()函数生成的值是一个range对象
    - range对象的行为和特征很像列表，但不是列表，这么做是为了节约空间
    - 在迭代的情况下返回指定索引的值

Python序列结构

- 列表元素增加的方式
  - + 运算符
  - append()方法
  - extend()方法
  - insert()方法
  - * 运算符

```
list_1 = [1, 3, 5, 7, 9]
list_2 = list_1 + [11]
#[1, 3, 5, 7, 9, 11]
```

实际上，这并不是真的为列表添加元素，而是创建了一个新列表，并将原列表和新元素依次复制到新列表的内存空间。

效率低

Python序列结构

- 列表元素增加的方式
  - + 运算符
  - append()方法
  - extend()方法
  - insert()方法
  - * 运算符

```
list_1 = [1, 3, 5, 7, 9]
list_1.append(11)
#[1, 3, 5, 7, 9, 11]
```

*append()方法在当前列表尾部追加元素，**原地修改列表**。*
“原地”是指不改变列表在内存中的首地址。

效率高

Python序列结构

- 列表元素增加的方式
  - + 运算符
  - append()方法
  - extend()方法
  - insert()方法
  - * 运算符

```
list_1 = [1, 3, 5, 7, 9]
list_1.extend([11])
#[1, 3, 5, 7, 9, 11]
```

*extend()方法将另一个迭代对象的所有元素加至该列表对象尾部，不改变列表对象的内存首地址，属于**原地操作***

效率高

Python序列结构

- 列表元素增加的方式
  - + 运算符
  - append()方法
  - extend()方法
  - insert()方法
  - * 运算符

```
list_1 = [1, 3, 5, 7, 9]
list_1.insert(5, 11)
#[1, 3, 5, 7, 9, 11]
```

insert(index, x)在列表list指定位置index处添加元素x，该位置后面的所有元素后移一个位置

列表的insert()可以在列表的任意位置插入元素，但由于列表的自动内存管理功能，insert()方法会引起插入位置之后所有元素的移动，影响处理速度

Python序列结构

- 列表元素增加的方式
  - + 运算符
  - append()方法
  - extend()方法
  - insert()方法
  - * 运算符

```
list_1 = [1, 3, 5]
list_2 = list_1*2
#[1, 3, 5, 1, 3, 5]
```

使用乘法“”来扩展列表对象，将列表与整数相乘，生成一个新列表，新列表是原列表中元素的重复*

Python序列结构

- 列表元素删除的方式
  - del命令
  - pop()方法
  - remove()方法

```
list_1 = [1, 3, 5, 7, 9]
del list_1[0]
del list_1[-1]
print(list_1) #[3, 5, 7]
```

del命令既可以删除列表中指定位置的元素，也可以直接删除整个列表

Python序列结构

- 列表元素删除的方式
  - del命令
  - pop()方法
  - remove()方法

```
list_1 = [1, 3, 5, 7, 9]
list_1.pop()
#[1, 3, 5, 7]
list_1.pop(1)
#[1, 5, 7]
list_1.pop(3)
#IndexError: pop index out of range
```

删除并返回指定位置（默认是最后一个）的元素，如果给定的索引超出了列表的范围则抛出异常

Python序列结构

- 列表元素删除的方式
  - del命令
  - pop()方法
  - remove()方法

```
list_1 = [1, 3, 5, 3, 5, 7, 9]
list_1.remove(1)
#[3, 5, 3, 5, 7, 9]
list_1.remove(5)
#[3, 3, 5, 7, 9]
list_1.remove(1)
#ValueError: list.remove(x): x not in list
```

删除首次出现的指定元素，如果列表中不存在匹配的元素，则抛出异常

Python序列结构

- 列表是Python中内置有序、可变序列，支持双向索引
- 当列表元素增加或删除时，列表对象自动进行扩展或收缩内存

列表创建
- =
- list()

列表元素增加
- + 运算符
- append()方法
- extend()方法
- insert()方法
- * 运算符

列表元素删除
- del命令
- clear()方法
- pop()方法
- remove()方法

列表删除
- del命令

列表元素操作
- count()方法
- index()方法
- sort()方法
- sorted()方法
- reverse()方法
- reversed()方法
- 切片操作

Python序列结构

- 示例：列表元素删除

```
list_1 = [1, 2, 1, 2, 1, 2]
for i in list_1:
    if i == 1:
        list_1.remove(i)
print(list_1)
```

- 程序目的是为了移除列表中的元素1，对否?

Python序列结构

- 示例：列表元素删除

```
list_1 = [1, 2, 1, 2, 1, 2]
for i in list_1:
    if i == 1:
        list_1.remove(i)
print(list_1)
#[2, 2, 2] 结果正确!
```

```
list_1 = [1, 1, 2, 1, 2, 1, 2]
for i in list_1:
    if i == 1:
        list_1.remove(i)
print(list_1)
#[2, 2, 1, 2] 结果错了!
```

Python序列结构

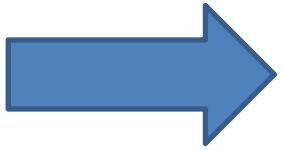
- 示例：列表元素删除

- 列表数据的差异
  - 第一组数据中没有连续的“1”，而第二组数据中存在连续的“1”
- 出错是因为没考虑列表的自动内存管理功能
  - 在删除列表元素时，Python会自动对列表内存进行收缩并移动列表元素以保证所有元素之间没有空隙，增加元素同理
  - 每当插入或删除一个元素后，该元素位置后面的所有元素的索引均发生变化

Python序列结构

- 看看发生了什么?

```
list_1 = [1, 1, 2, 1, 2, 1, 2]
for i in list_1:
    if i == 1:
        list_1.remove(i)
print(list_1)
#[2, 2, 1, 2] 结果错了!
```



```
# coding=utf-8
list_1 = [1, 1, 2, 1, 2, 1, 2]
j = 0
for i in list_1:
    j += 1
    print(f'第{j}次检测：元素{i}')
    if i == 1:
        list_1.remove(i)
        print(f'移除元素1，列表变为{list_1}')
    else:
        print(f'列表未改变，当前列表为{list_1}')
```

Python序列结构

- 实际的移除元素过程

list_1 = [1, 2, 1, 2, 1, 2]

第1次检测：元素1
移除元素1，列表变为[2, 1, 2, 1, 2]
第2次检测：元素1
移除元素1，列表变为[2, 2, 1, 2]
第3次检测：元素1
移除元素1，列表变为[2, 2, 2]

list_1 = [1, 1, 2, 1, 2, 1, 2]

第1次检测：元素1
移除元素1，列表变为[1, 2, 1, 2, 1, 2]
第2次检测：元素2
列表未改变，当前列表为[1, 2, 1, 2, 1, 2]
第3次检测：元素1
移除元素1，列表变为[2, 1, 2, 1, 2]
第4次检测：元素1
移除元素1，列表变为[2, 2, 1, 2]

Python序列结构

- 实际的移除元素过程

list_1 = [1, 2, 1, 2, 1, 2]

第1次检测：元素1
移除元素1，列表变为[2, 1, 2, 1, 2]
第2次检测：元素1
移除元素1，列表变为[2, 2, 1, 2]
第3次检测：元素1
移除元素1，列表变为[2, 2, 2]

  - 观察
    - remove()移除首次出现的元素
    - **内存空间自动收缩**
    - **remove()移除后，for循环次数-1**
    - 理解**可变**

list_1 = [1, 1, 2, 1, 2, 1, 2]

第1次检测：元素1
移除元素1，列表变为[1, 2, 1, 2, 1, 2]
第2次检测：元素2
列表未改变，当前列表为[1, 2, 1, 2, 1, 2]
第3次检测：元素1
移除元素1，列表变为[2, 1, 2, 1, 2]
第4次检测：元素1
移除元素1，列表变为[2, 2, 1, 2]

Python序列结构

- 案例：列表元素删除
  - 正确写法

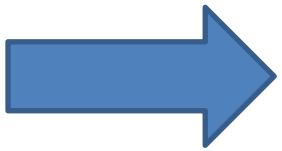
```
list_1 = [1, 1, 2, 1, 2, 1, 2]
for i in list_1[::]:
    if i == 1:
        list_1.remove(i)
print(list_1)
#[2, 2, 2]结果正确
```

```
list_1 = [1, 1, 2, 1, 2, 1, 2]
for i in range(len(list_1)-1, -1, -1):
    if list_1[i] == 1:
        del list_1[i]
print(list_1)
#[2, 2, 2]结果正确
```

小知识：Python代码调试

- 调试方法1：print()

```
list_1 = [1, 1, 2, 1, 2, 1, 2]
for i in list_1:
    if i == 1:
        list_1.remove(i)
print(list_1)
#[2, 2, 1, 2] 结果错了！
```



```
# coding=utf-8
list_1 = [1, 1, 2, 1, 2, 1, 2]
j = 0
for i in list_1:
    j += 1
    print(f'第{j}次检测：元素{i}')
    if i == 1:
        list_1.remove(i)
        print(f'移除元素1，列表变为{list_1}')
    else:
        print(f'列表未改变，当前列表为{list_1}')
```

小知识：Python代码调试

- 调试方法2：assert
  - 断言，声明其布尔值必须为真的判定，如果发生异常就说明表达示为假
  - assert condition [, ‘提示语’]

```
a = 0
assert a>2, 'a应大于2'
```

```
a = 0
if not a>2:
    raise AssertionError('a应大于2')
```

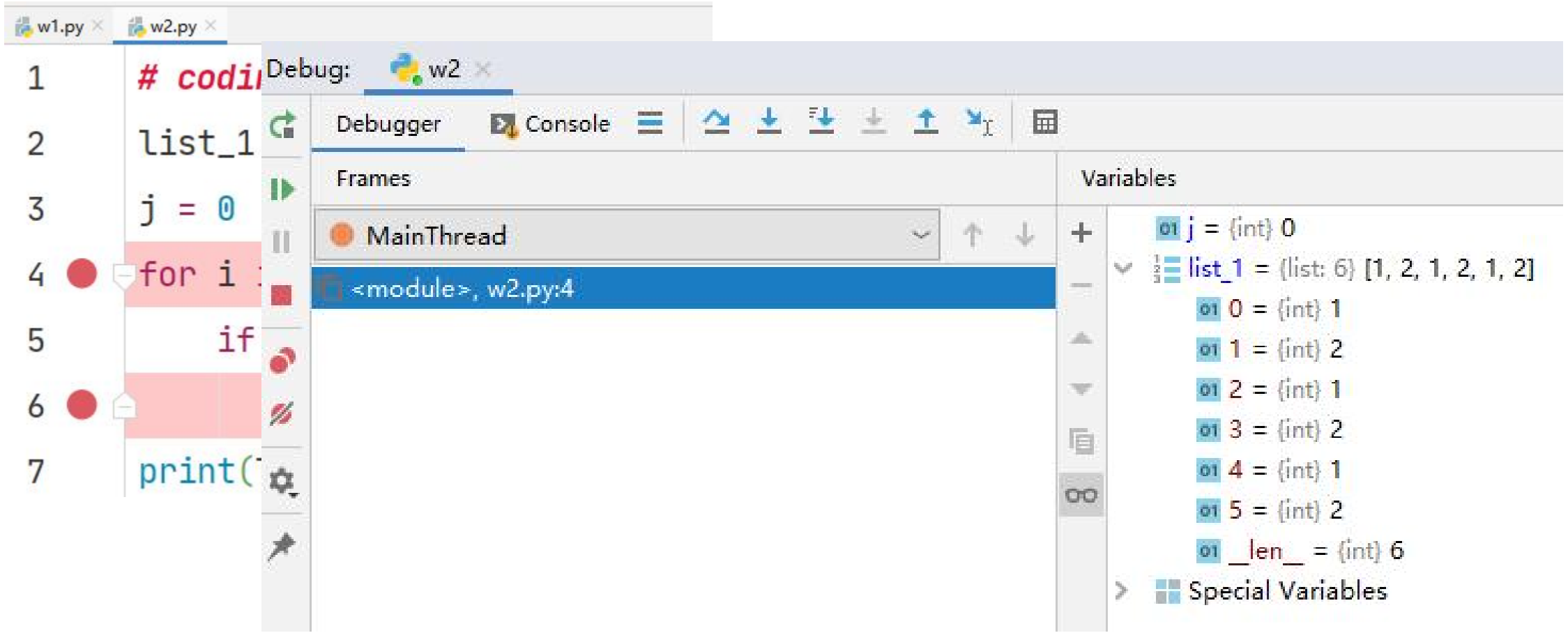
```
Traceback (most recent call last):
  File "C:/Users/xlkon/PycharmProjects/tempp/w2.py",
    assert a>2, 'a应大于2'
AssertionError: a应大于2
```

小知识：Python代码调试

- 调试方法3：pdb
  - 1）进入命令行Debug模式，python -m pdb xxx.py
  - 2）h：（help）帮助
  - 3）w：（where）打印当前执行堆栈
  - 4）d：（down）执行跳转到在当前堆栈的深一层（个人没觉得有什么用处）
  - 5）u：（up）执行跳转到当前堆栈的上一层
  - 6）b：（break）添加断点
    - b 列出当前所有断点，和断点执行到统计次数
    - b line_no：当前脚本的line_no行添加断点
    - b filename:line_no：脚本filename的line_no行添加断点
    - b function：在函数function的第一条可执行语句处添加断点

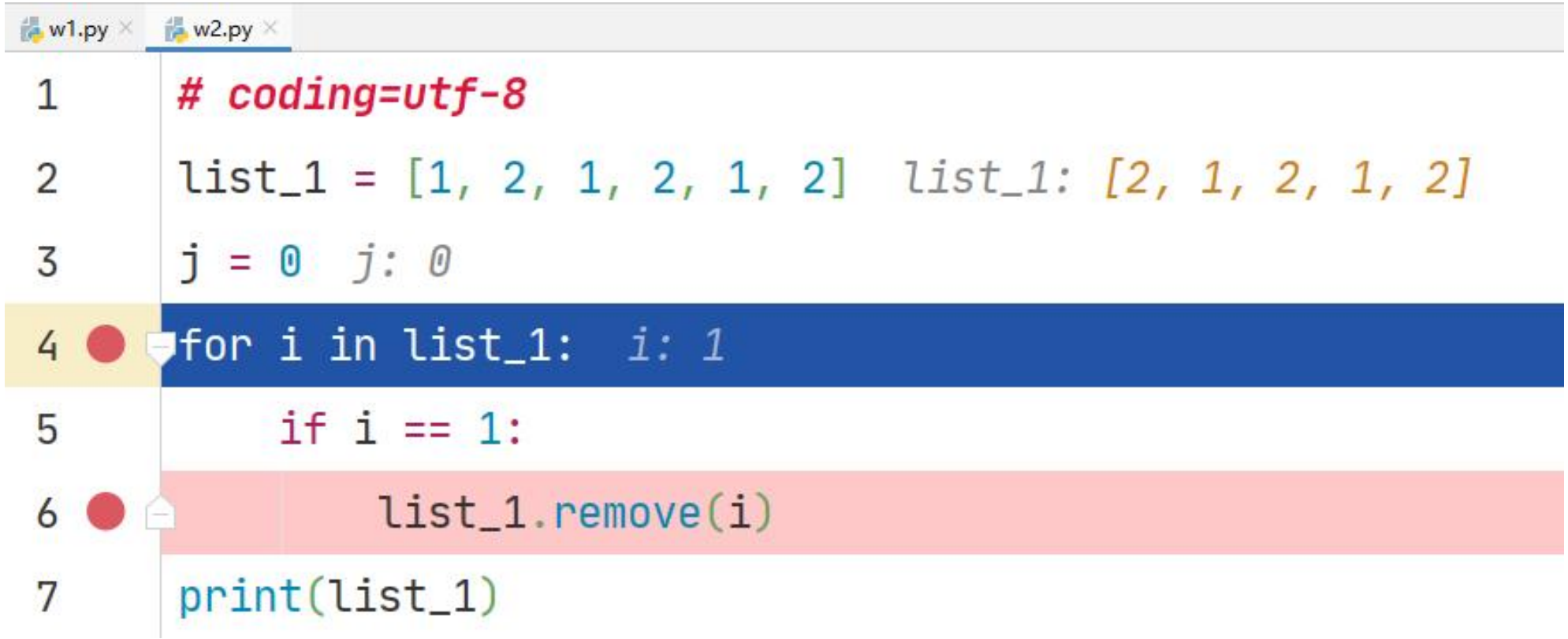
小知识：Python代码调试

- 调试方法4：IDE自带的debugger



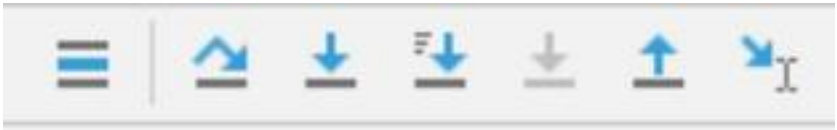
小知识：Python代码调试

- 调试方法4：IDE自带的debugger



小知识：Python代码调试

- 调试方法4：IDE自带的debugger
  - 以PyCharm为例



    - show execution point (F10) 显示当前所有断点
    - step over(F8) 单步调试
      - 若函数A内存在子函数a时，不会进入子函数a内执行单步调试，而是把子函数a当作一个整体，一步执行
    - step into(F7) 单步调试
      - 若函数A内存在子函数a时，会进入子函数a内执行单步调试
    - step into my code(Alt + Shift +F7) 执行下一行但忽略libraries（导入库的语句）
    - force step into(Alt + Shift +F7) 执行下一行忽略lib和构造对象等
    - step out（Shift+F8）当目前执行在子函数a中时，选择该调试操作可以直接跳出子函数a，而不用继续执行子函数a中的剩余代码。并返回上一层函数
    - run to cursor(Alt +F9) 直接跳到下一个断点

Python序列结构

- 列表元素的访问
  - 使用下标直接访问列表元素，如果指定下标不存在，则抛出异常

```
1  list_1 = list(range(6))
2  # list_1=[0, 1, 2, 3, 4, 5]
3  print(list_1[1])
4  #list_1[1]=1
5  list_1[2] = 10
6  # list_1=[0, 1, 10, 3, 4, 5]
7  print(list_1[10])
```

```
Traceback (most recent call last):
  File "C:/Users/xlkon/PycharmProjects/training/d1.py", line 7, in <module>
    print(list_1[10])
IndexError: list index out of range
```

Python序列结构

- 列表元素的访问
  - 使用列表对象的index()方法获取指定元素首次出现的下标，若列表对象中不存在指定元素，则抛出异常

```
1  list_1 = list(range(6))
2  # list_1=[0, 1, 2, 3, 4, 5]
3  print(list_1.index(2))
4  # 2
5  list_1[1] = 2
6  print(list_1.index(1))
```

```
Traceback (most recent call last):
  File "C:/Users/xlkon/PycharmProjects/training/d1.py", line 6, in <module>
    print(list_1.index(1))
ValueError: 1 is not in list
```

Python序列结构

- 列表元素的统计
  - count()方法统计指定元素在列表对象中出现的次数

- 列表元素的成员资格判断
  - 使用in关键字来判断一个值是否存在于列表中，返回结果为True或False

```
1   list_1 = [0, 1, 2, 3, 3, 2, 4, 5]
2   print(list_1.count(2))
3   # 2
4   print(list_1.count(1))
5   # 1
6   print(list_1.count(6))
7   # 0
8   print(2 in list_1)
9   # True
10  print(6 in list_1)
11  # False
12  print([2] in list_1)
13  # False
```

Python序列结构

- 列表元素排序
  - 使用列表对象的sort()方法进行原地排序

  list.sort(key=None, reverse=False)
    - key -- 主要是用来进行比较的元素，只有一个参数，具体的函数的参数就是取自于可迭代对象中，指定可迭代对象中的一个元素来进行排序。
    - reverse -- 排序规则，reverse = True 降序， reverse = False 升序（默认）

```
1  import random
2  list_1 = [1, 3, 5, 7, 9, 11]
3  random.shuffle(list_1) # list_1=[9, 1, 5, 7, 11, 3]
4  list_1.sort() # list_1=[1, 3, 5, 7, 9, 11]
5  list_1.sort(reverse=True) # list_1=[11, 9, 7, 5, 3, 1]
```

Python序列结构

- 列表元素排序
  - 使用列表对象的sort()方法进行原地排序
  - sort()的参数key用来表示排序的依据
  - lambda 创建一个匿名函数。冒号前是传入参数，后是一个处理传入参数的单行表达式

```
1  import random
2  list_1 = [1, 3, 5, 7, 9, 11, 13, 15]
3  random.shuffle(list_1) # list_1=[9, 13, 15, 3, 5, 7, 1, 11]
4  list_1.sort(key=lambda x: len(str(x)))
5  # 按照转换为字符串的长度排序 [9, 3, 5, 7, 1, 13, 15, 11]
6  list_2 = ['big', 'apple', 'book']
7  list_2.sort(key=lambda x: (x[0], x[1]))
8  #按照第一个字母排序，相同则按照第二个字母排序
9  #list_2=['apple', 'big', 'book']
```

Python序列结构

- 列表元素排序
  - 使用内置函数sorted()对列表排序并返回新列表

```
6   list_2 = ['big', 'apple', 'book']
7   list_2.sort(key=lambda x: (x[0], x[1]))
8   #按照第一个字母排序，相同则按照第二个字母排序
9   #list_2=['apple', 'big', 'book']
10  list_3 = sorted(list_2, key=lambda x: (x[0], x[1]), reverse=True)
11  #sorted()返回一个新列表，['book', 'big', 'apple']
```

Python序列结构

- 列表元素排序
  - 使用列表对象的reverse()方法进行原地逆序
  - 使用内置函数reversed()对列表元素逆序排列并返回迭代对象

- 常用内置函数

  len(列表): 返回列表中的元素个数

  max(列表)、min(列表)：返回列表中的最大或最小元素

  sum(列表)：对列表的元素进行求和运算

  zip(...)接受任意多个可迭代对象作为参数,将对象中对应的元素打包成一个元组,然后返回一个可迭代的zip对象

  enumerate(列表)：枚举列表元素，返回枚举对象，其中每个元素为包含下标和值的元组

Python序列结构

- 切片操作
  - 切片使用2个冒号分割的3个数字来完成：list[start: stop: step]
  - 第一个数字start表示切片的开始位置（默认是0）
  - 第二个数字stop表示切片截止（不包含）位置（默认是列表长度）
  - 第三个数字step表示切片的步长（默认为1），当步长省略时可以顺便省略最后一个冒号

```
1    list_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2    print(list_1[0:5:1]) # [1, 2, 3, 4, 5], 取值左闭右开，步长是1
3    print(list_1[::]) #返回包含所有元素的新列表，[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
4    print(list_1[::-1]) #逆序所有元素，[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Python序列结构

- 切片适用于列表、字符串、range对象等类型
- 切片可以用于截取、修改、删除、增加列表元素

```
1  list_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2  print(list_1[::2]) # 列表中偶数下标的元素 [1, 3, 5, 7, 9]
3  print(list_1[1::2]) # 列表中奇数下标的元素 [2, 4, 6, 8, 10]
4  print(list_1[3::]) # 从下标3开始的所有元素 [4, 5, 6, 7, 8, 9, 10]
5  print(list_1[0:100:1])
6  # 列表中的前100个元素，自动截断，[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
7  print(list_1[100:]) # 列表中下标100以后的元素，自动截断[]
8  print(list_1[100]) # 越界异常
```

Python序列结构

- 切片操作
  - 切片返回的是列表元素的浅复制，即生成一个新的列表，并且把原列表中所有元素的引用都复制到新列表中
  - 如果原列表中只包含整数、实数、复数等基本类型或元组、字符串这样的不可变类型的数据，切片操作不会改变原列表
  - 如果原列表对象中包含列表之类的可变数据类型，由于浅复制时只是把子列表的引用复制到新列表中，这样修改任何一个都会影响另外一个

Python序列结构

- 切片操作
  - 切片属于浅复制

```
1  list_1 = [1, 3, 5]
2  list_2 = list_1
3  list_2[0] = 10
4  print(list_1) # list_1改变 [10, 3, 5]
5  print(id(list_1) == id(list_2)) #True
```

```
1  list_1 = [1, 3, 5]
2  list_2 = list_1[::]
3  list_2[0] = 10
4  print(list_1) # list_1不改变 [1, 3, 5]
5  print(id(list_1) == id(list_2)) #False
```

Python序列结构

- 切片操作
  - 原列表中只包含基本类型或不可变类型的数据

```
original_list = [1, 2, 3, 'a', 'b', 'c']  # 创建一个包含整数和字符串的列表
sliced_list = original_list[1:4]   # 对列表进行切片

# 输出原列表和切片列表
print("Original list:", original_list)  # Output: [1, 2, 3, 'a', 'b', 'c']
print("Sliced list:", sliced_list)      # Output: [2, 3, 'a']
sliced_list[0] = 'x'   # 修改切片列表中的元素

# 再次输出原列表和切片列表
print("Original list after modification:", original_list)
# Output: [1, 2, 3, 'a', 'b', 'c']  # 原列表没有变化
print("Modified sliced list:", sliced_list)        # Output: ['x', 3, 'a']
```

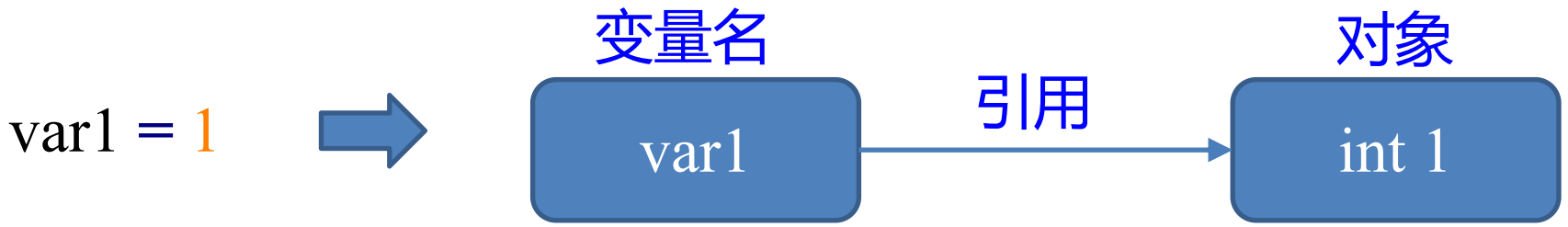
Python序列结构

- 切片操作
  - 原列表对象中包含列表之类的可变数据类型

```
1    list_1 = [1, 3, [5, 7]]
2    list_2 = list_1[::]
3    list_2[2].append(9)
4    print(list_1) # list_1改变 [1, 3, [5, 7, 9]]
5    print(id(list_1) == id(list_2)) #False
```

小知识：浅复制与深复制

- Python中对象的赋值其实就是对象的引用
  - 当创建一个对象，把它赋值给另一个变量的时候，Python并没有拷贝这个对象，只是拷贝了这个对象的引用

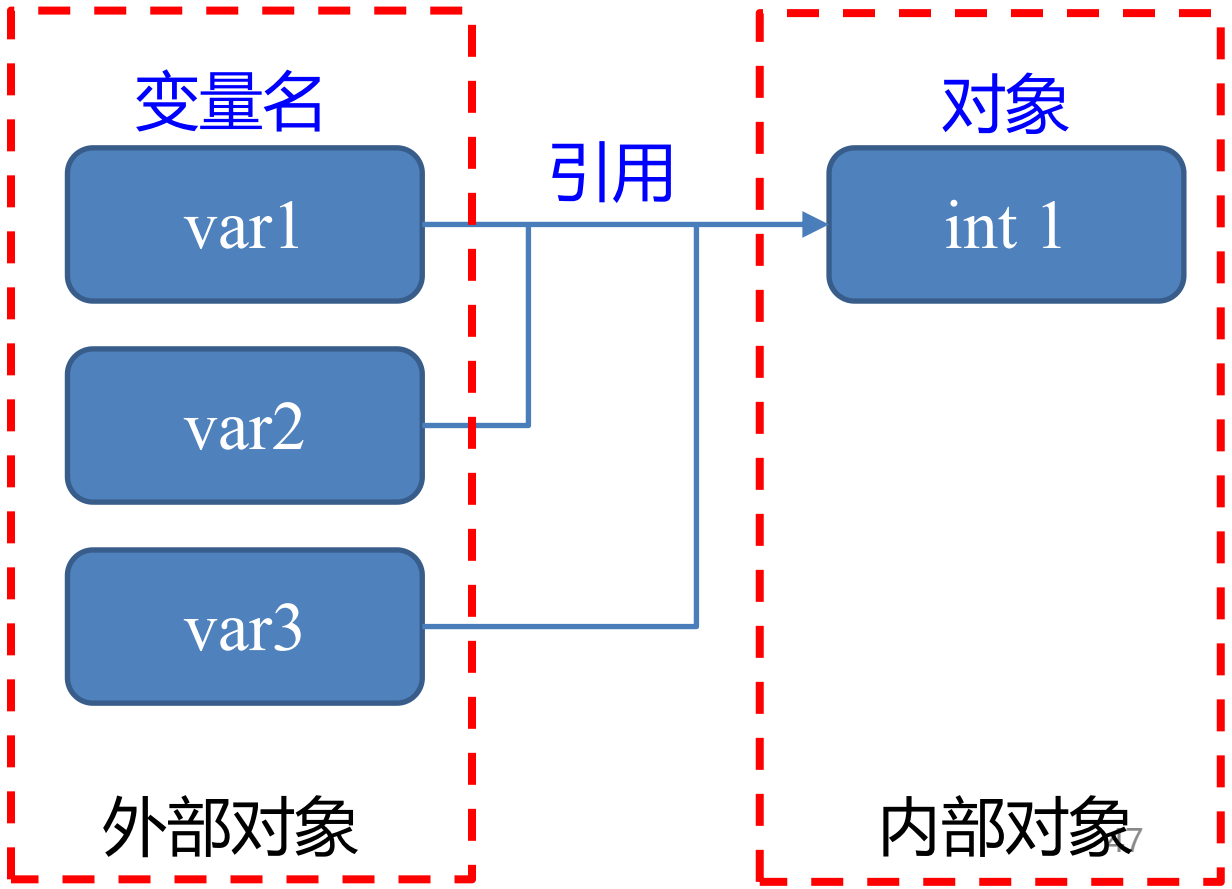


  - 变量通过引用（指针）指向具体对象的内存空间，取对象的值
  - 对象，类型已知，每个对象都包含一个头部信息（头部信息：类型标识符和引用计数器）
  - 变量名没有类型，类型属于被引用的对象
    - Python中万物皆是对象，变量名本身可看作是外围的对象

小知识：浅复制与深复制

- 浅复制
  - 复制了外部的对象本身，内部的元素都只是复制了一个引用
    - 把对象复制一遍，但是不复制该对象中引用的其他对象

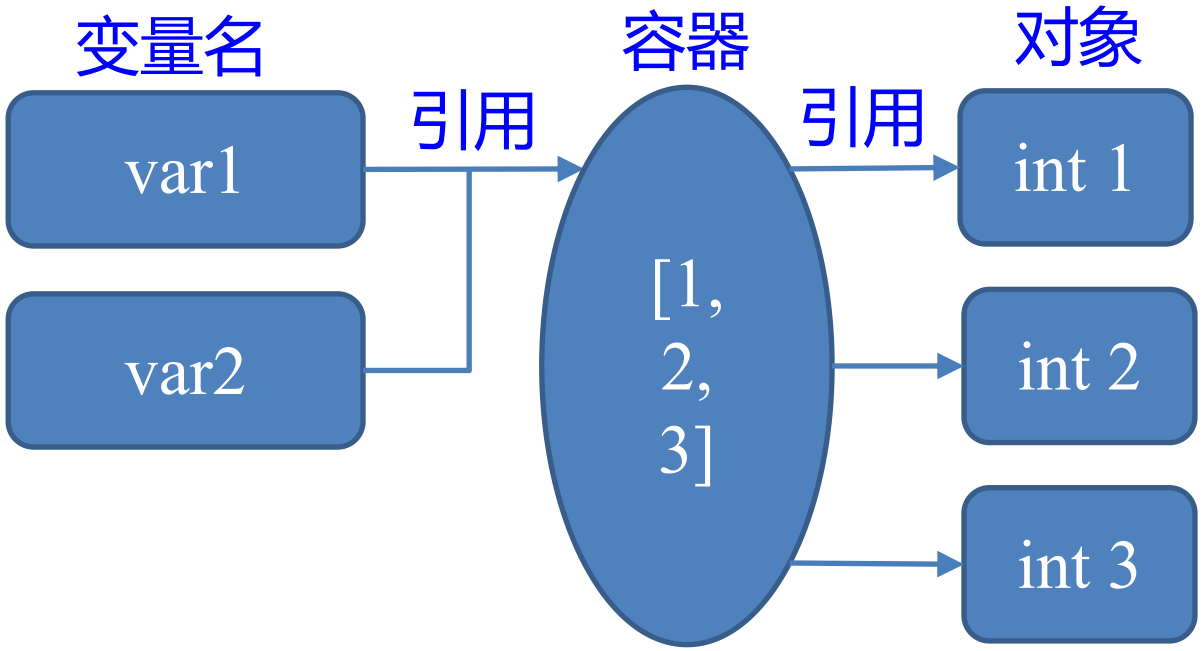
```
import copy
var1 = 1
var2 = var1                    #浅复制
var3 = copy.copy(var2)         #浅复制
#id(var1),id(var2)和id(var3)是一样的
```



小知识：浅复制与深复制

- 赋值
  - Python中赋值其实就是对象的引用（指针）

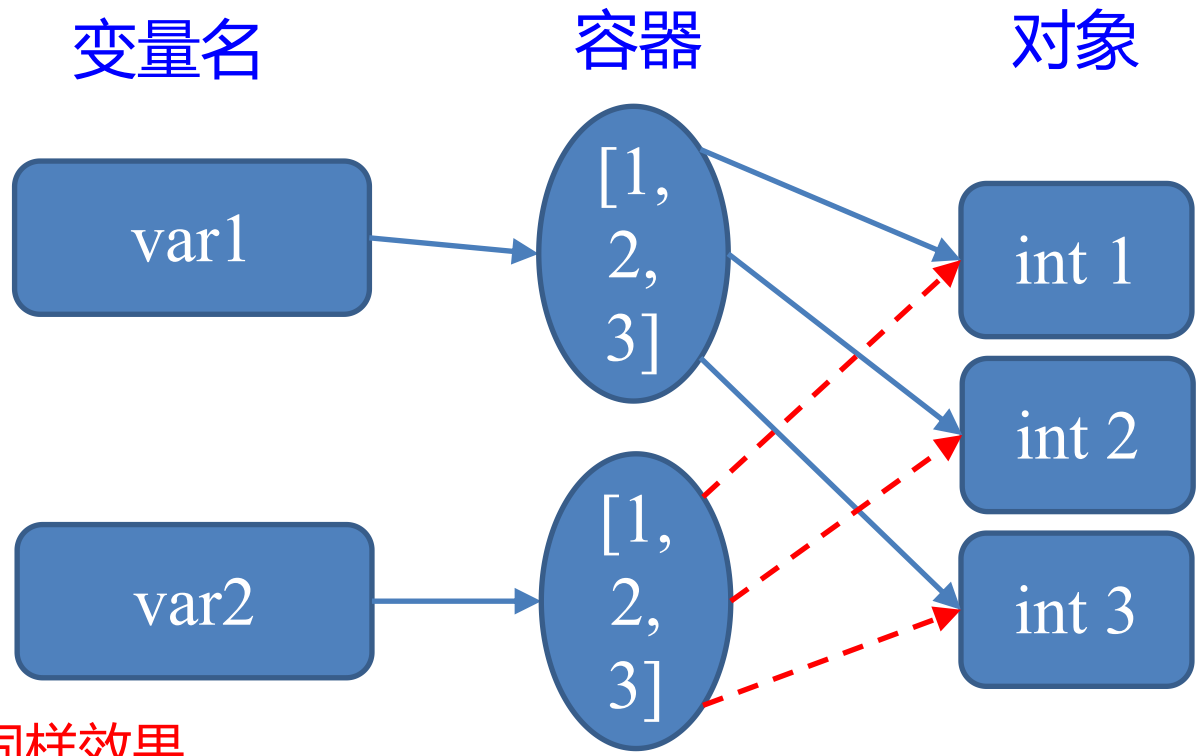
```
var1 = [1, 2, 3]
var2 = var1
#使用等号复制（赋值）
var2.append(4)
print(id(var1),id(var2))
#两者地址值相同
print(var1)
# var1= [1, 2, 3, 4]，跟着改变了
```



小知识：浅复制与深复制

- 浅复制
  - 复制了外部的对象本身，内部的元素都只是复制了一个引用
    - 浅复制（对于任何容器类型，包括列表、元组、字典等）通常意味着创建一个新的容器对象，但是该容器对象包含的是对原始容器元素的引用

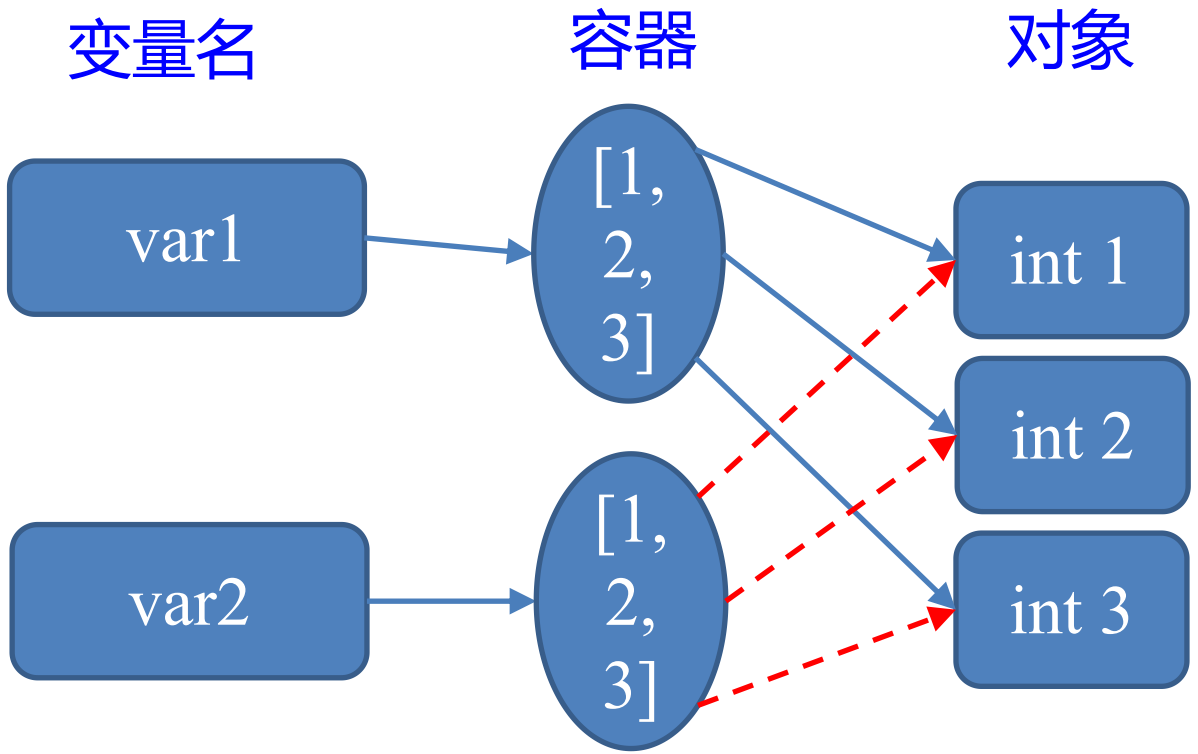
```
var1 = [1, 2, 3]
var2 = copy.copy(var1) #浅复制
var3 = var1[:]
#浅复制，等价于var1[::]
print(id(var1),id(var2),id(var3))
#三者互不相同
var2.append(4)
var3.append(5)
print(var1) # [1, 2, 3]
print(var2) # [1, 2, 3, 4]
print(var3) # [1, 2, 3, 5]
# list(var1), var1.copy(), var1*1也是同样效果
```



小知识：浅复制与深复制

- 浅复制
  - 复制了外部的对象本身，内部的元素都只是复制了一个引用
    - 浅复制（对于任何容器类型，包括列表、元组、字典等）通常意味着创建一个新的容器对象，但是该容器对象包含的是对原始容器元素的引用

```
var1 = [1, 2, [3]]
var2 = copy.copy(var1) #浅复制
var3 = var1[:]
#浅复制，等价于var1[::]
print(id(var1),id(var2),id(var3))
#三者互不相同
var2[2].append(4)
var3[2].append(5)
print(var1) # [1, 2, [3, 4, 5]]
print(var2) # [1, 2, [3, 4, 5]]
print(var3) # [1, 2, [3, 4, 5]]
```

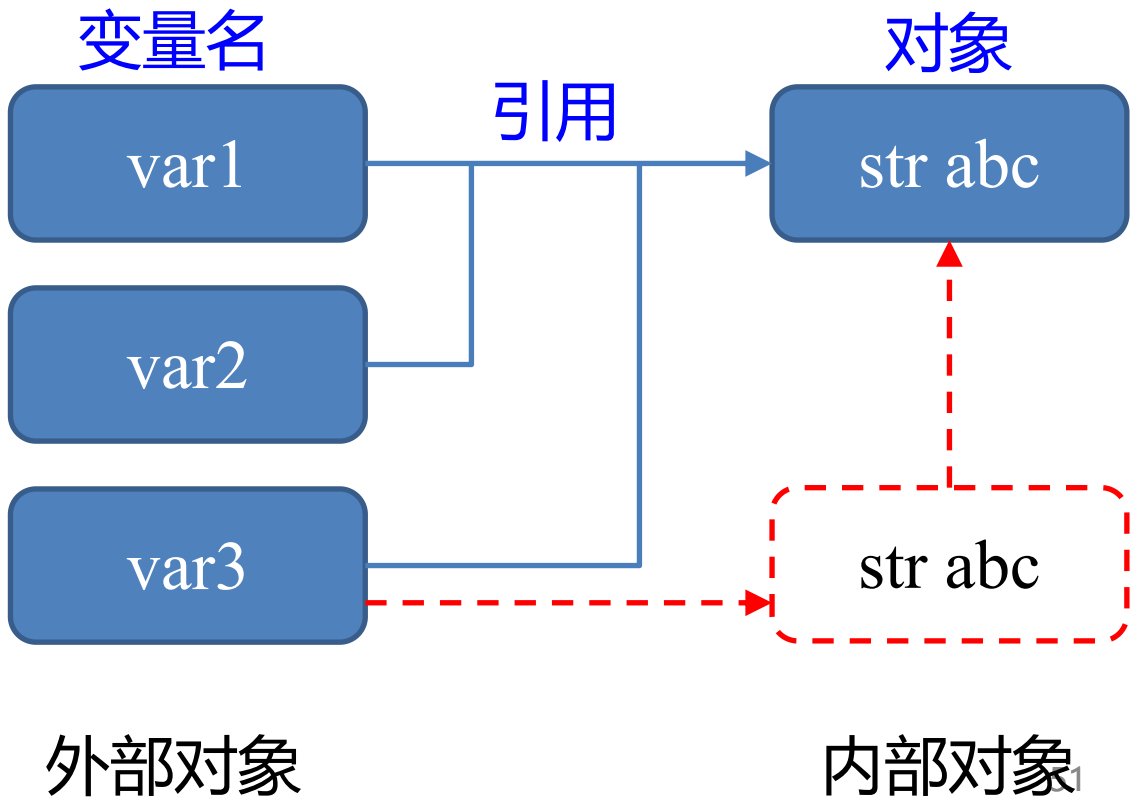


小知识：浅复制与深复制

- 深复制
  - 复制了外围和内部元素，而不是引用
    - 把对象复制一遍，也复制该对象中引用的其他对象

```
import copy

var1 = [1, 2, [3]]
var2 = copy.deepcopy(var1) #浅复制
#浅复制，等价于var1[::]
print(id(var1),id(var2),id(var3))
#三者互不相同
var2[2].append(4)
print(var1) # [1, 2, [3]]
print(var2) # [1, 2, [3, 4]]
```

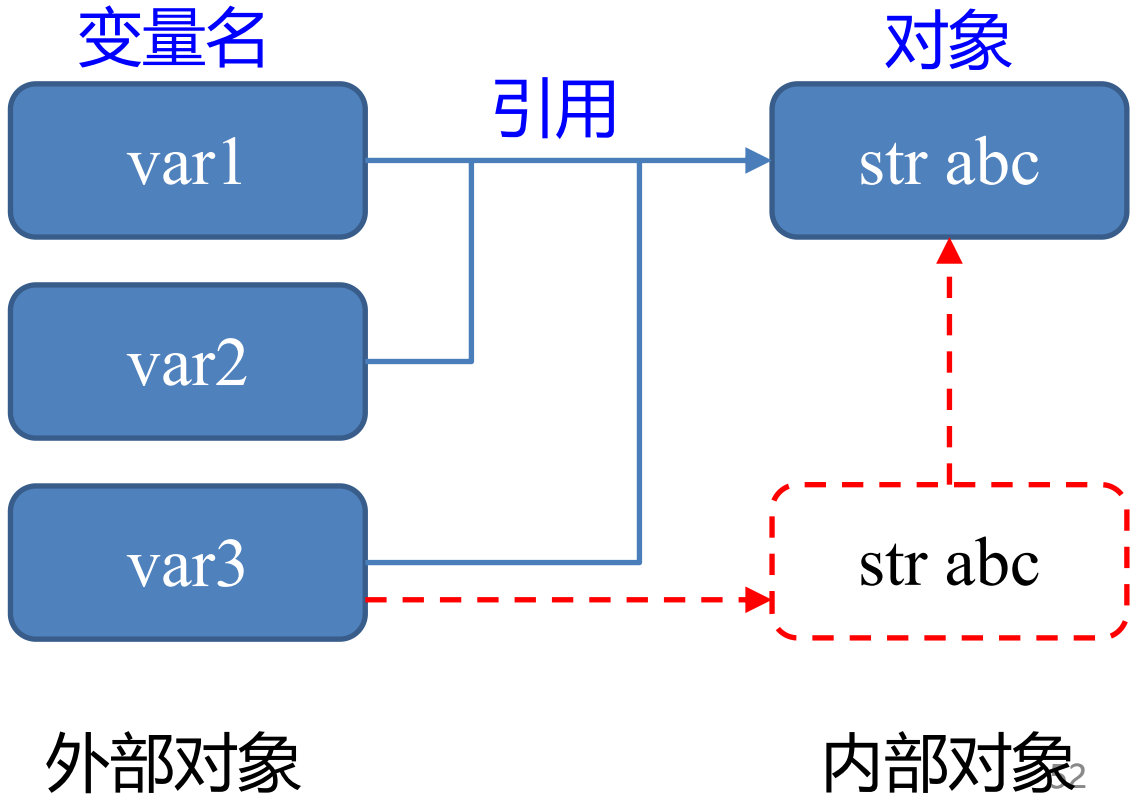


小知识：浅复制与深复制

- 深复制
  - 复制了外围和内部元素，而不是引用
    - 把对象复制一遍，也复制该对象中引用的其他对象

```
import copy
var1 = 'abc'
var2 = copy.copy(var1) #浅复制
var3 = copy.deepcopy(var1) #深复制
#id(var1),id(var2)和id(var3)是一样的
#Python使用基于值的内存管理机制
```

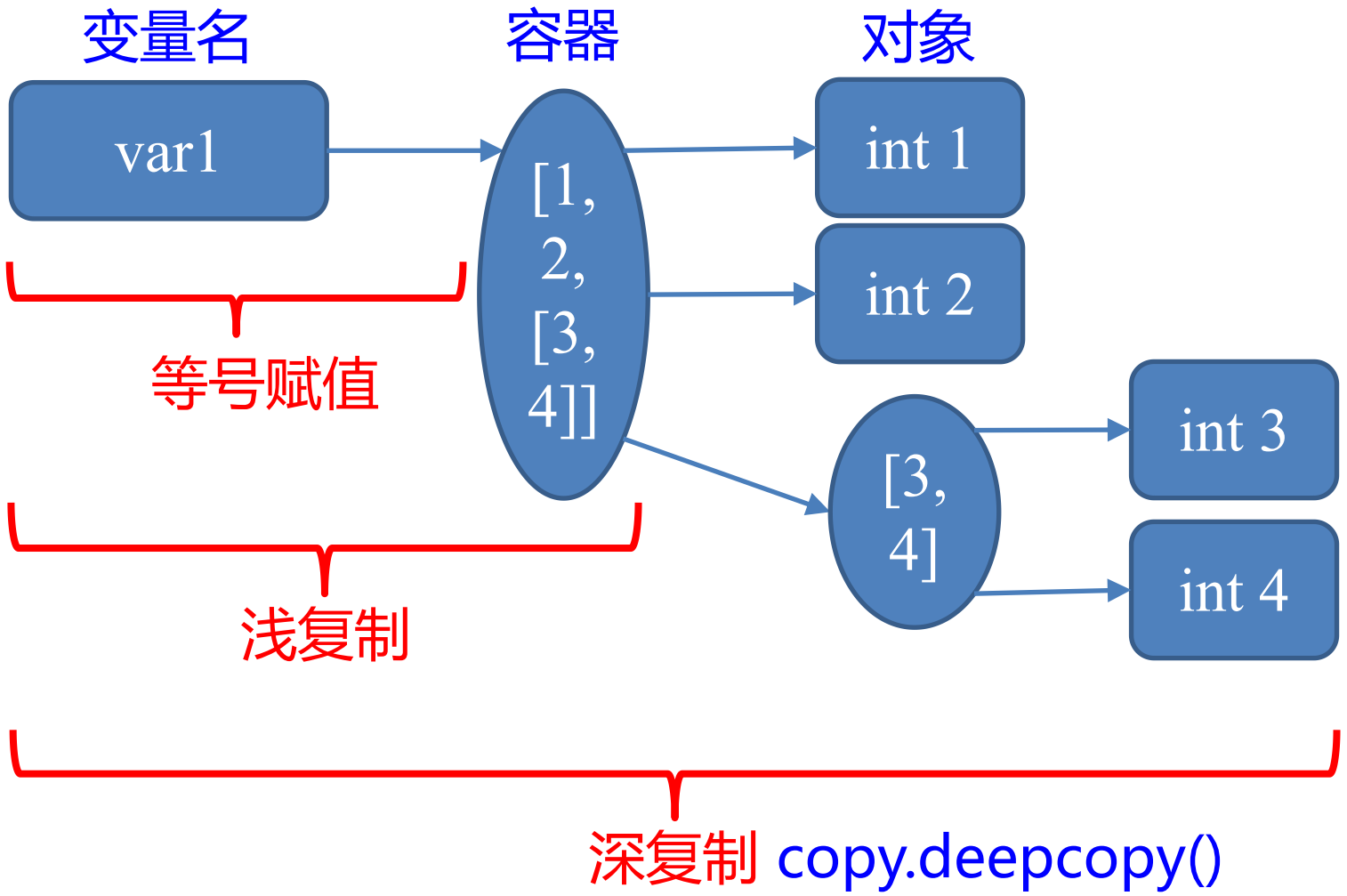
  - 不可变类型的对象
    - 不管是深or浅，地址值都不变



小知识：浅复制与深复制

- 多层引用

- **等号赋值**
  - **值相等，地址相等**
- **copy浅复制**
  - **值相等，地址不相等**
- deepcopy深复制
  - 值相等，地址不相等



小知识：Python内存回收

- 在Python中，每个对象都有指向该对象的引用总数---引用计数
- 查看对象的引用计数：sys.getrefcount()

```
import sys
var1 = ['x','y','z']
print(sys.getrefcount(var1)) # 2
```

- 当使用某个引用作为参数，传递给getrefcount()时，参数实际上创建了一个临时的引用。因此，getrefcount()所得到的结果，会比期望的多1。

小知识：Python内存回收

- 当Python的某个对象的引用计数降为0时，说明没有任何引用指向该对象，该对象就成为要被回收的垃圾

- 当Python运行时，会记录其中分配对象(object allocation)和取消分配对象(object deallocation)的次数。当两者的差值高于某个阈值时，垃圾回收才会启动。

```
import gc
print(gc.get_threshold()) # (700, 10, 10)
```

  - 700即是垃圾回收启动的阈值
  - 每10次0代垃圾回收，会配合1次1代的垃圾回收；而每10次1代的垃圾回收，才会有1次的2代垃圾回收
  - 经历过1次垃圾回收，代数+1，新建时为0

作业题

- 数组切片操作
  - 分别选取一个数组的奇数列和偶数列
  - 对数组元素逆序排列
  - 对数组的每个元素重复n次，输出新的数组，比如对[1,2,3]重复2次，新数组为[1,1,2,2,3,3] （不能用for实现）

- 简单数组操作
  - 对数组元素排序
  - 对数组元素求和
  - 对数组元素随机打乱顺序
  - 从数组中随机选择一个数

作业题

- 二维数组操作
  - 对一个二维数组按照行的中心线翻转，输出新的数组
  - 对一个二维数组按照列的中心线翻转，输出新的数组
  - 假设二维数组的行数和列数相等，输出其对角线上元素组成的数组
- 二维数组和Numpy转化
  - 假设二维数组的元素均为数字，将该二维数组转成Numpy向量
  - 将Numpy向量转换成二维数组
- 对列表中的可变数据类型和不可变数据类型，分别采用赋值、浅复制和深赋值，修改新的对象，比较新对象和原对象的数值及地址