Day 3：list 和 tuple 的基本操作、深浅拷贝和切片操作详细等 5 个方面总结

1. 列表

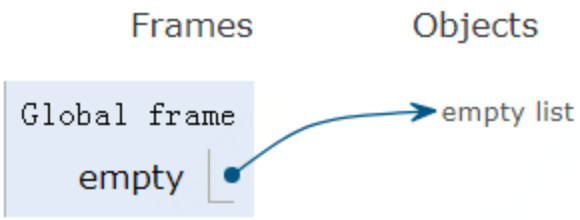
列表（list）作为 Python 中最常用的数据类型之一，是一个可增加、删除元素的可变（mutable）容器。

1.1 基本操作

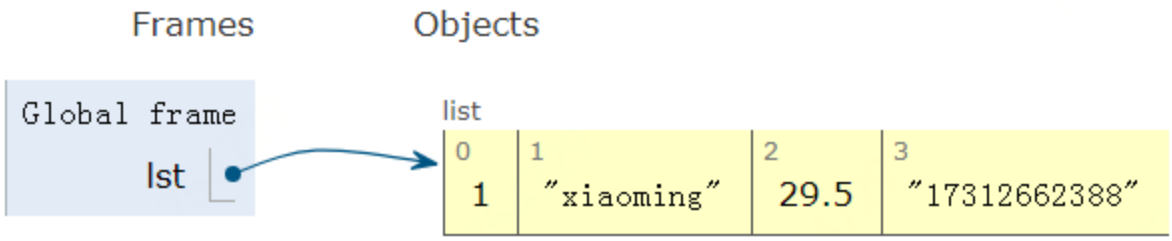
创建 list 的方法非常简单，只使用一对中括号 `[]`。如下创建三个 list：

```
empty = []
lst = [1, 'xiaoming', 29.5, '17312662388']
lst2 = ['001', '2019-11-11', ['三文鱼', '电烤箱']]
```

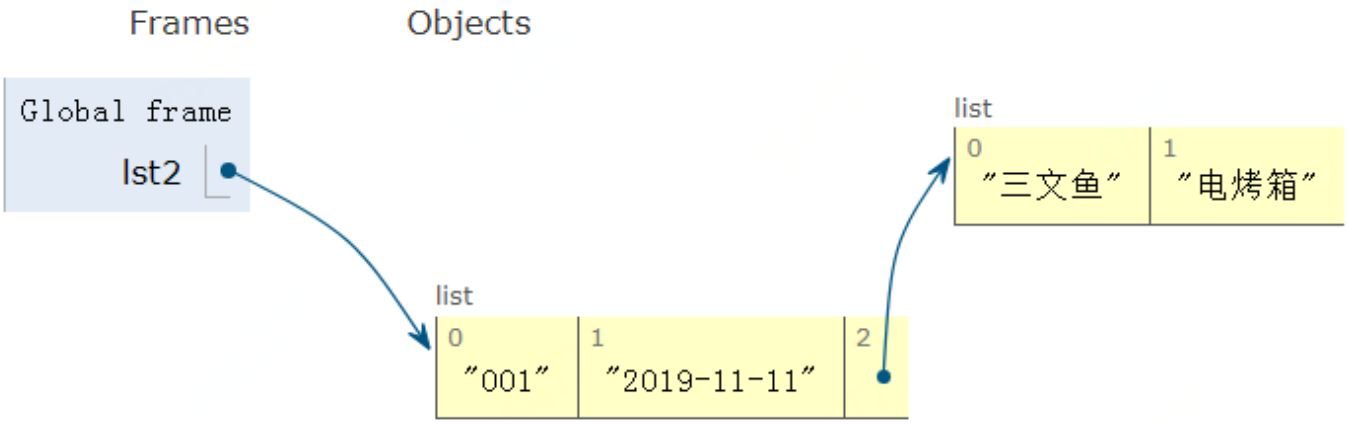
empty 在内存中的示意图：



lst 在内存中的示意图：



lst2 在内存中的示意图：



使用 Python 的内置函数 len 求 list 内元素个数：

```
1 len(empty)  # 0
2 len(lst)  # 4
3 len(lst2)  # 3
```

依次遍历 lst 内每个元素并求对应类型，使用 `for in` 对遍历，内置函数 type 得到类型：

```
for _ in lst:
    print(f'{_}的类型为{type(_)}')
```

打印结果如下，列表 lst 内元素类型有 3 种：

```
1的类型为<class 'int'>
xiaoming的类型为<class 'str'>
29.5的类型为<class 'float'>
17312662388的类型为<class 'str'>
```

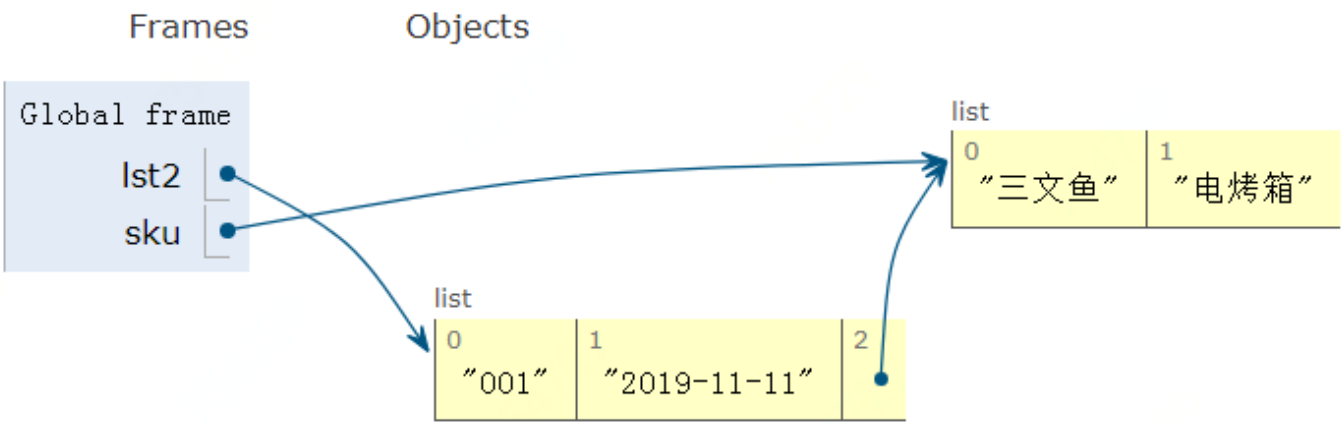
因此，Python 的列表不要求元素类型一致。

如何向 lst2 的第三个元素 `['三文鱼','电烤箱']` 内再增加一个元素 `'烤鸭'`。

首先，使用“整数索引”取出这个元素：

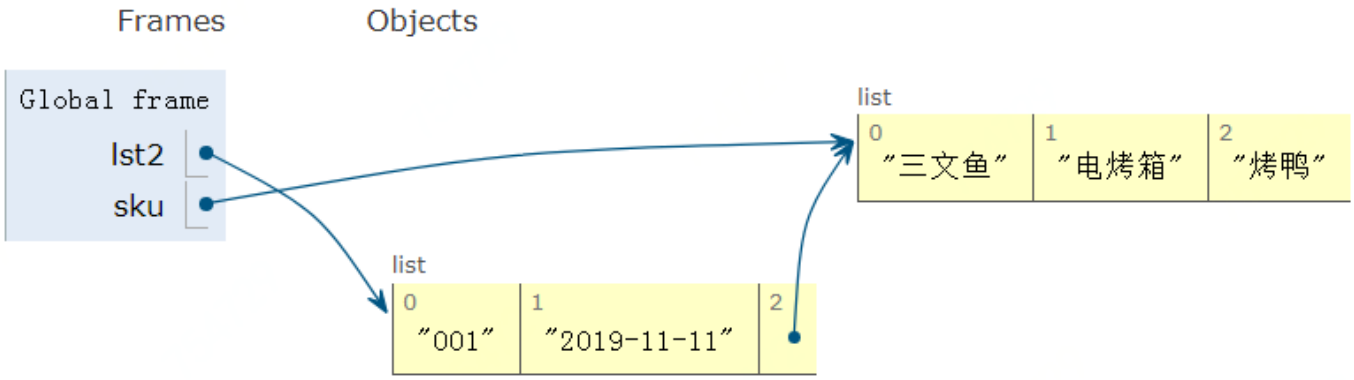
```
sku = lst2[2]  # sku 又是一个列表
```

sku 变量位于栈帧中，同时指向 lst2[2]：



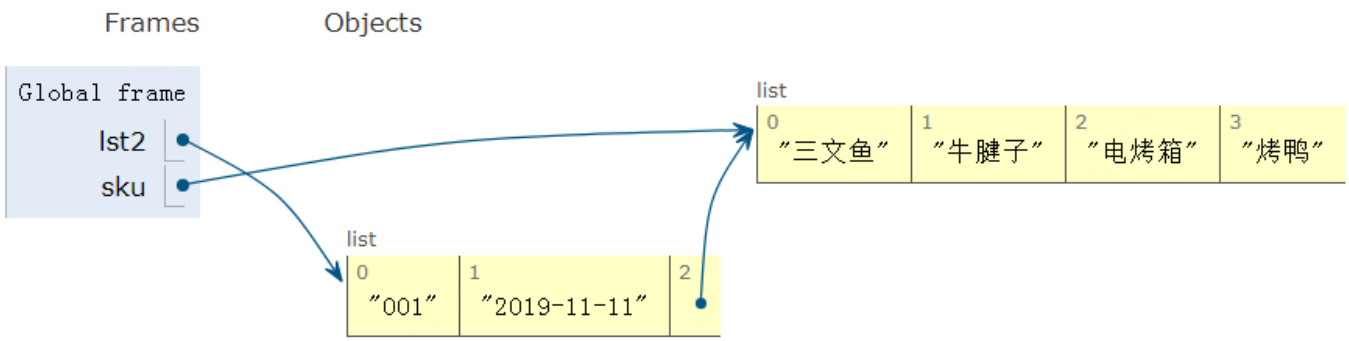
然后，使用列表的 append 方法增加元素，append 默认增加到 sku列表尾部：

```
sku.append('烤鸭')
print(sku)  # ['三文鱼', '电烤箱', '烤鸭']
```



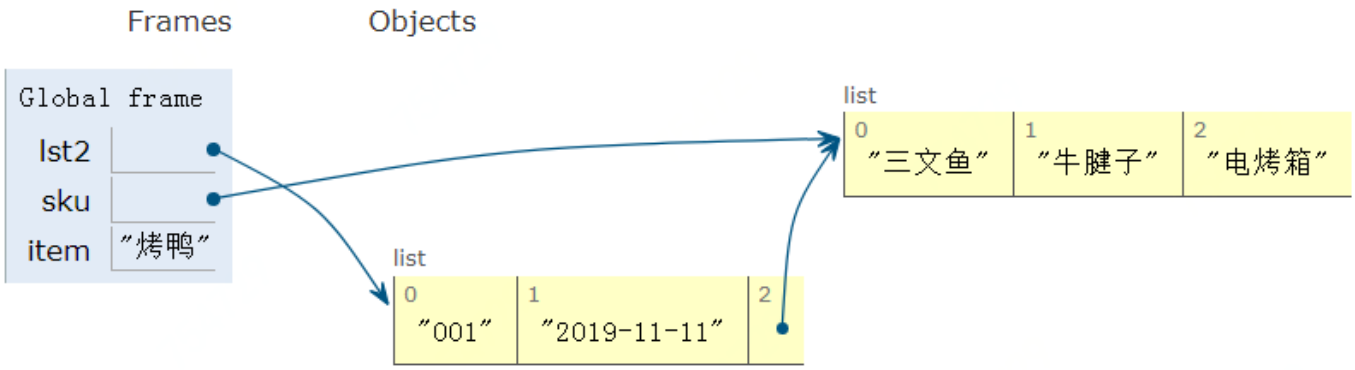
此时想在 sku 指定索引 1 处插入“牛腱子”，使用列表的 insert 方法：

```
sku.insert(1, '牛腱子')
print(sku)  # ['三文鱼', '牛腱子', '电烤箱', '烤鸭']
```



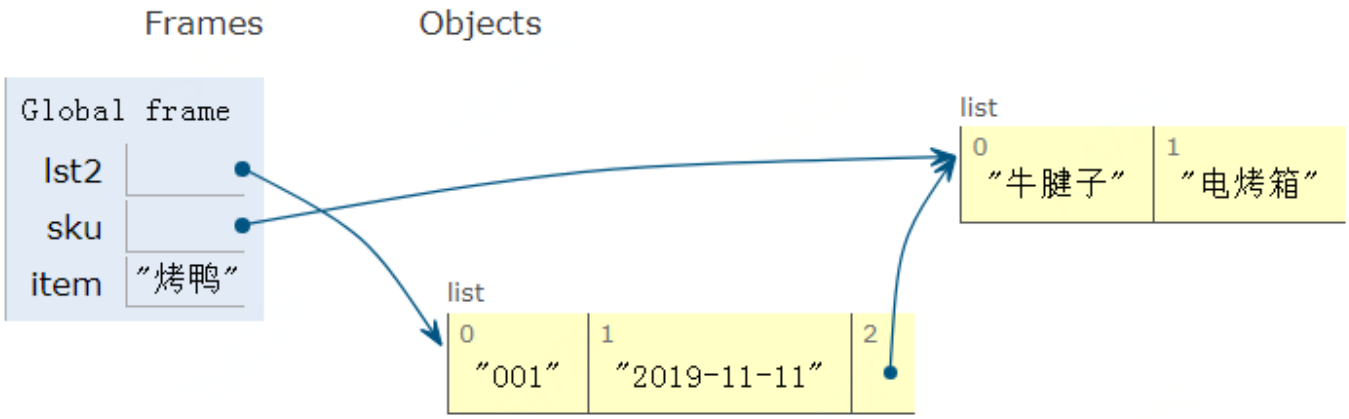
在购买烤鸭和牛腱子后，发现超出双十一的预算，不得不放弃购买烤鸭，使用 pop 方法可直接移除列表尾部元素：

```
item = sku.pop()  # 返回烤鸭
print(sku)  # ['三文鱼', '牛腱子', '电烤箱']
```



发现还是超出预算，干脆移除三文鱼，pop 因为只能移除表尾元素，幸好列表有 remove 方法：

```
sku.remove('三文鱼')  # 更好用：sku.remove(sku[0])
print(sku)  # ['牛腱子', '电烤箱']
```



1.2 深浅拷贝

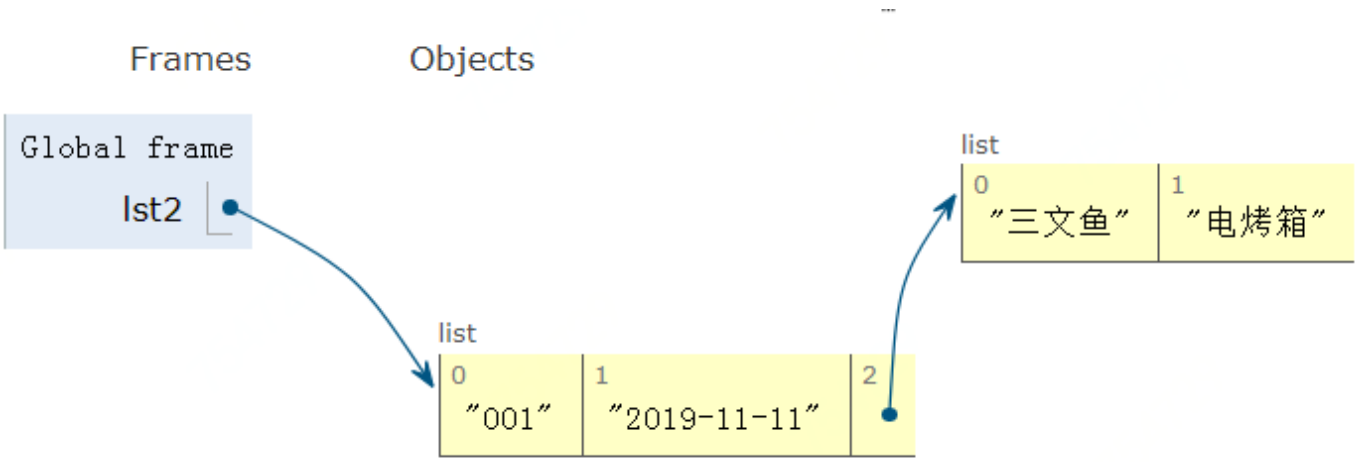
打印 lst2，发现第三个元素也对应改变，因为 sku 引用 lst2 的第三个元素，sku 指向的内存区域改变，所以 lst2 也会相应改变。

```
print(lst2)  # ['001', '2019-11-11', ['牛腱子', '电烤箱']]
```

如果不想改变 lst2 的第三个元素，就需要复制出 lst2 的这个元素，列表上有 copy 方法可实现复制：

```
1 lst2 = ['001', '2019-11-11', ['三文鱼', '电烤箱']]  # 这是lst2的初始值
```

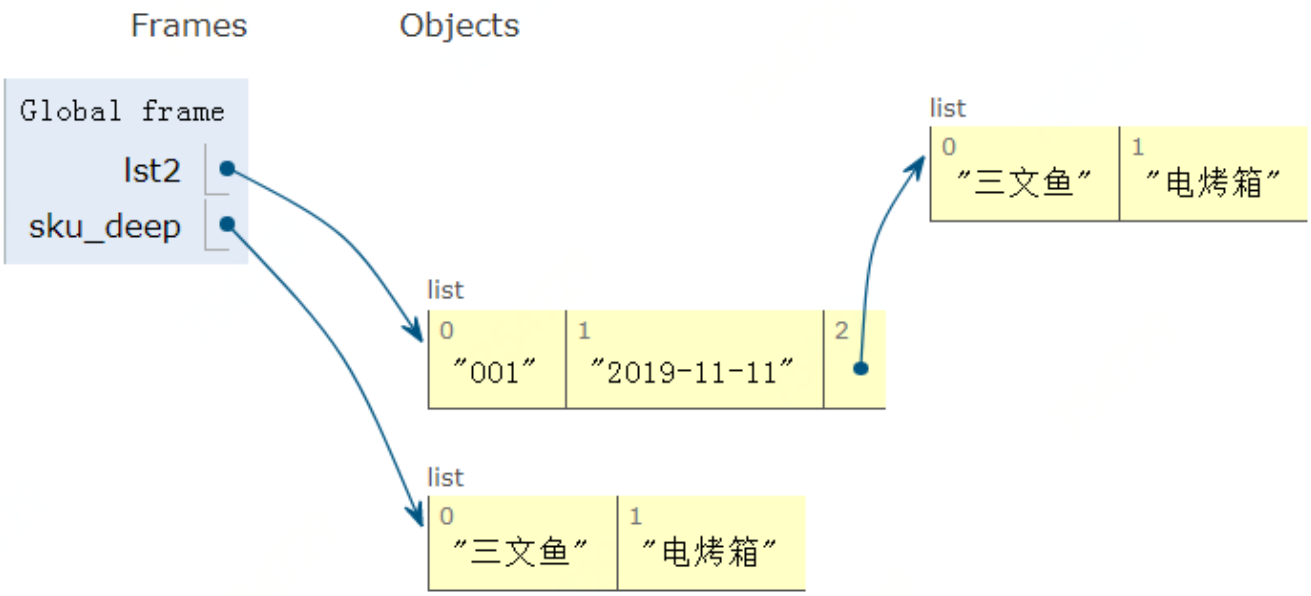
可视化此行代码，lst2 位于全局帧栈中，其中三个元素内存中的可视化图如下所示：




```
1 sku_deep = lst2[2].copy()
```

注意，copy 函数，仅仅实现对内嵌对象的一层拷贝，属于 shallow copy。**「浅拷贝」**

此时可视化图为如下，因为拷贝 `lst2[2]`，所以 sku_deep 位于栈帧中指向一块新的内存空间：



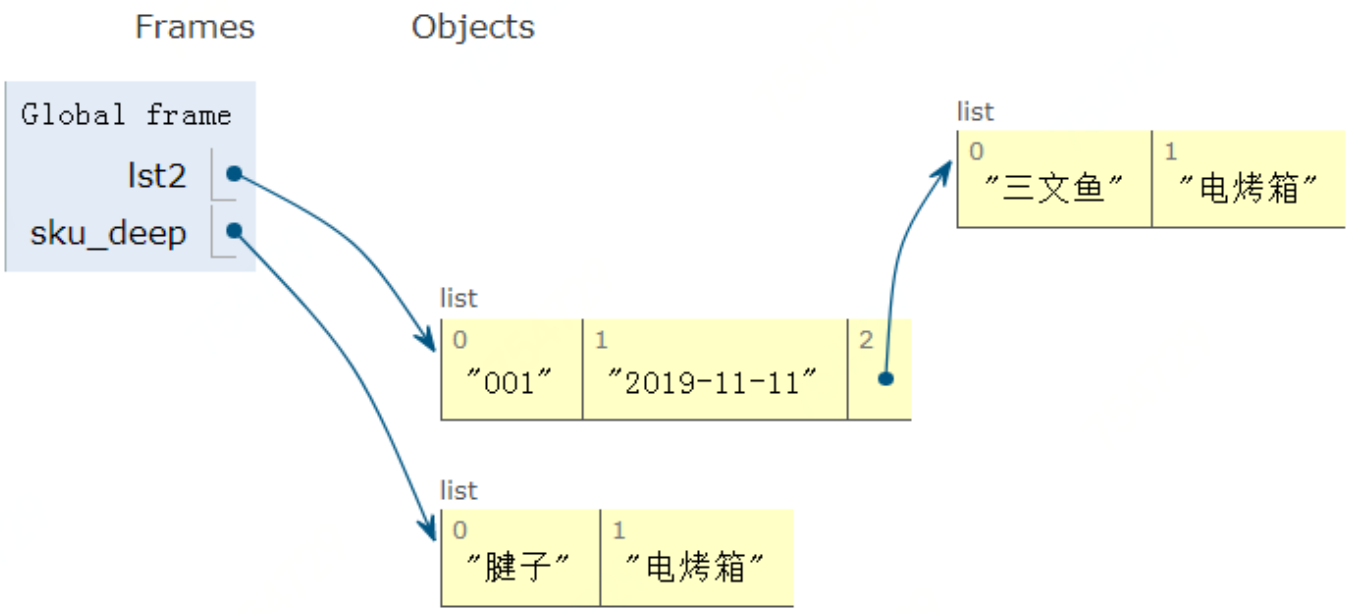
此时，再对 sku_deep 操作，便不会影响 lst2[2] 的值。

如下修改 sku_deep 的第一个元素（Python 的列表索引从 0 开始编号），lst2 未受到任何影响。

```
sku_deep[0] = '腱子'
print(lst2[2])  # ['三文鱼','电烤箱']
```

修改 sku_deep 时，不会影响 `lst2[2]`。

因为它们位于不同的内存空间中，如图所示，`lst2[2]` 中的第一个元素依然是“三文鱼”，而不是“腱子”。



至此，仅仅使用 shallow copy。那么，它与**深拷贝**，英文叫 deepcopy，又有什么不同?

请看下面例子，a 是内嵌一层 list 的列表，对其浅拷贝生成列表 ac，修改 ac 的第三个元素，也就是列表 [3,4,5] 中的第二个元素为 40：

```
a = [1, 2, [3, 4, 5]]
ac = a.copy()
ac[0] = 10  # [10, 2, [3, 4, 5]]
ac[2][1] = 40  # [10, 2, [3, 40, 5]]
```

修改后，分别测试两个值的相等性。

```
print(a[0] == ac[0])  # False
```

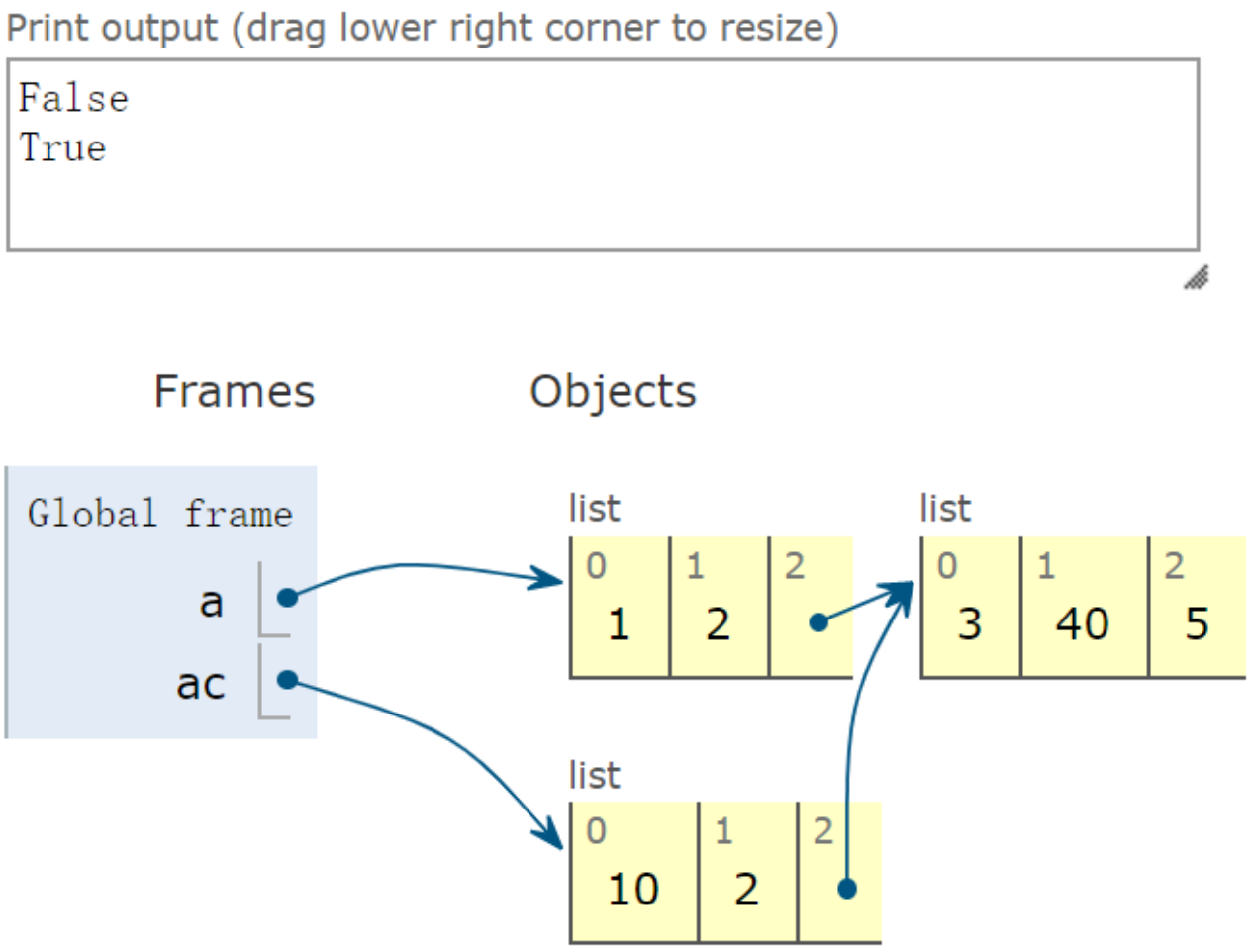
返回 False，证明实现拷贝。

而 `ac[2][1]` 是否与原数组 a 的对应位置元素相等：

```
print(a[2][1] == ac[2][1])  # True
```

返回 True，进一步证明是浅拷贝，不是深拷贝。

如下图所示：copy 只完成了一层 copy，即 `[1,2, id([3,4,5])]` 复制一份，而复制后，仍然指向 [3,4,5] 所在的内存空间：



要想实现深度拷贝，需要使用 copy 模块的 deepcopy 函数：

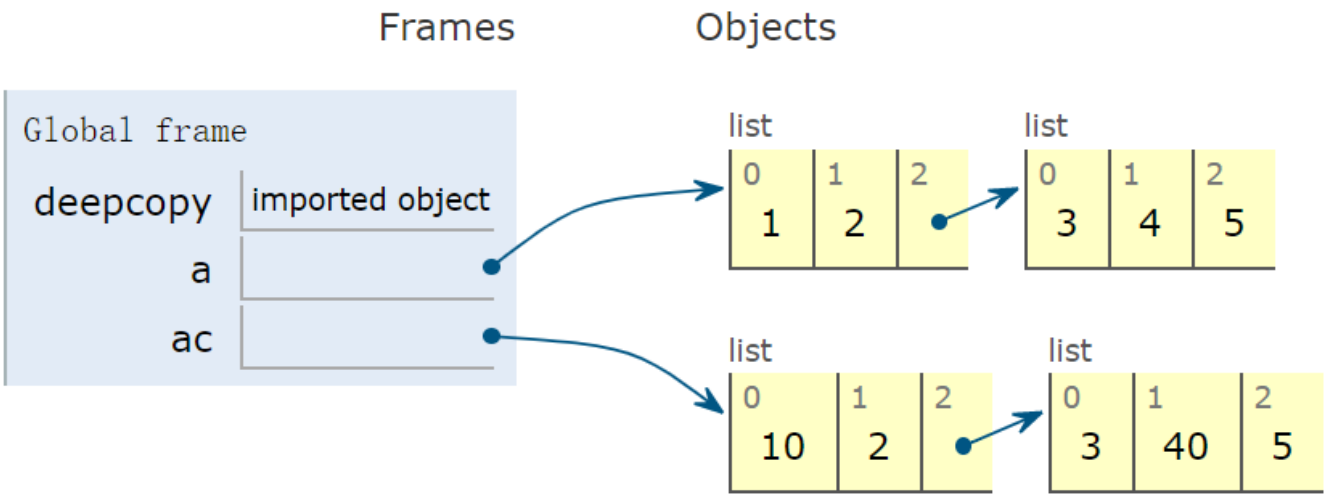
```
from copy import deepcopy

a = [1, 2, [3, 4, 5]]
ac = deepcopy(a)
ac[0] = 10
```

```
ac[2][1] = 40
print(a[0] == ac[0])  # False
print(a[2][1] == ac[2][1])  # False
```

打印结果，都为 False，结合下图，也能看出内嵌的 list 全部完成复制，都指向了不同的内存区域。

Print output (drag lower right corner to resize)

```
False
False
```



1.3 切片

Java 和 C++ 中，访问数组中的元素只能一次一个，但 Python 增加切片功能为访问列表带来极大便利。利用内置函数 `range(start,stop,step)` 生成序列数据，并转为 list 类型：

```
a = list(range(1, 20, 3))
print(a)  # [1, 4, 7, 10, 13, 16, 19]
```

使用 `a[:3]` 获取列表 a 的前三个元素，形象称这类操作为**“切片”**，切片本身也是一个列表 `[1,4,7]`：

- 使用 `a[-1]` 获取 a 的最后一个元素，返回 int 型，值为 19；
- 使用 `a[:-1]` 获取除最后一个元素的切片 [1, 4, 7, 10, 13, 16]；
- 使用 `a[1:5]` 生成索引为 [1, 5)（不包括索引 5）的切片 [4, 7, 10, 13]；
- 使用 `a[1:5:2]` 生成索引 [1, 5) 但步长为 2 的切片 [4, 10]；
- 使用 `a[::3]` 生成索引 [0, len(a)) 步长为 3 的切片 [1, 10, 19]；
- 使用 `a[::-3]` 生成逆向索引 [len(a), 0) 步长为 3 的切片 [19, 10, 1]。

> 逆向：从列表最后一个元素访问到第一个元素的方向。

特别地，使用列表的逆向切片操作，只需一行代码就能逆向列表：

```
def reverse(lst):
    return lst[::-1]
```

调用 `reverse` 函数：

```
ra = reverse(a)
print(ra)  # [19, 16, 13, 10, 7, 4, 1]
```

说完列表，还有一个与之很相似的数据类型——元组（tuple）。

2. 元组

元组既然是不可变（immutable）对象，自然也就没有增加、删除元素的方法。

2.1 基本操作

使用一对括号（`()`）就能创建一个元组对象，如：

```python
a = ()  # 空元组对象
b = (1, 'xiaoming', 29.5, '17312662388')
c = ('001', '2019-11-11', ['三文鱼', '电烤箱'])
```

它们都是元组，除了 list 是用 `[]` 创建外，其他都与 list 很相似，比如都支持切片操作。

> **特别注意：**一个整数加一对括号，比如 (10)，返回的是整数。必须加一个逗号 (10,) 才会返回元组对象。

```python
tup = (10)
print(type(tup))  # <class 'int'>
tup2 = (10,)
print(type(tup2))  # <class 'tuple'>
```

列表和元组都有一个很好用的统计方法 count，实现对某个元素的个数统计：

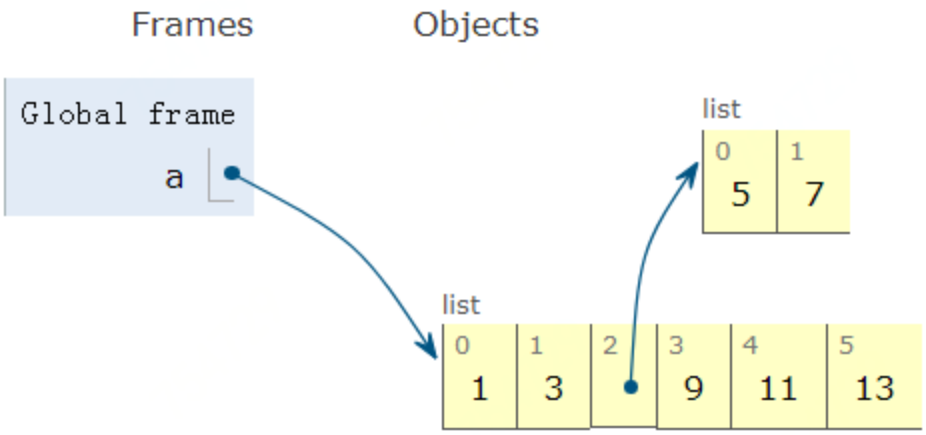
```python
from numpy import random

a = random.randint(1, 5, 10)  # 从 [1,5) 区间内随机选择 10 个数
at = tuple(a)  # 转 tuple：(1, 4, 2, 1, 3, 3, 2, 3, 4, 2)
at.count(3)  # 统计 3 出现次数，恰好也为 3 次
```

2.2 可变与不可变

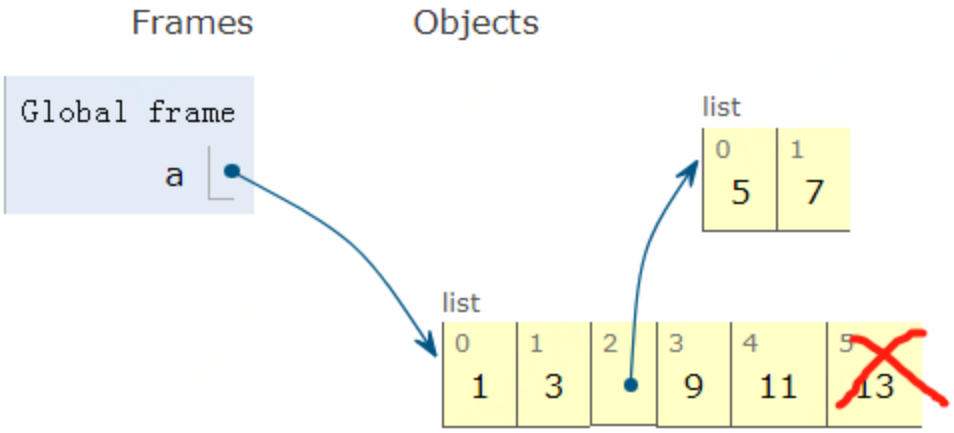
文章开头提到列表是一个**可变**容器，可变与不可变是一对很微妙的概念。

因为网上经常出现，所以再重点总结下。

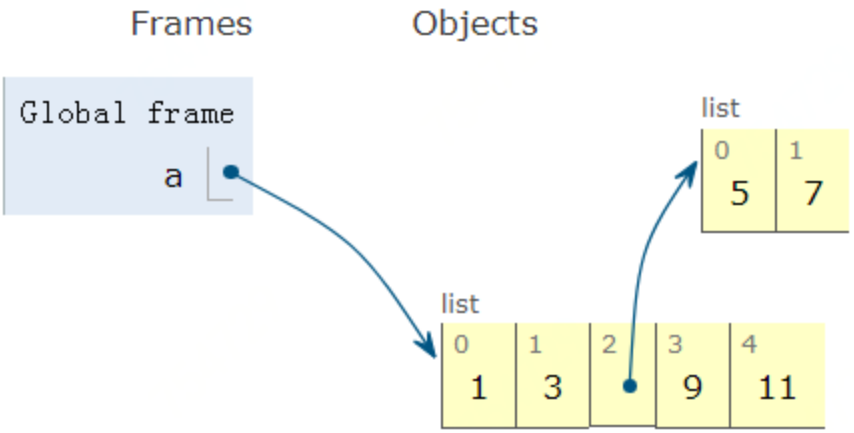
创建一个列表 `a = [1, 3, [5, 7], 9, 11, 13]`，存储示意图：



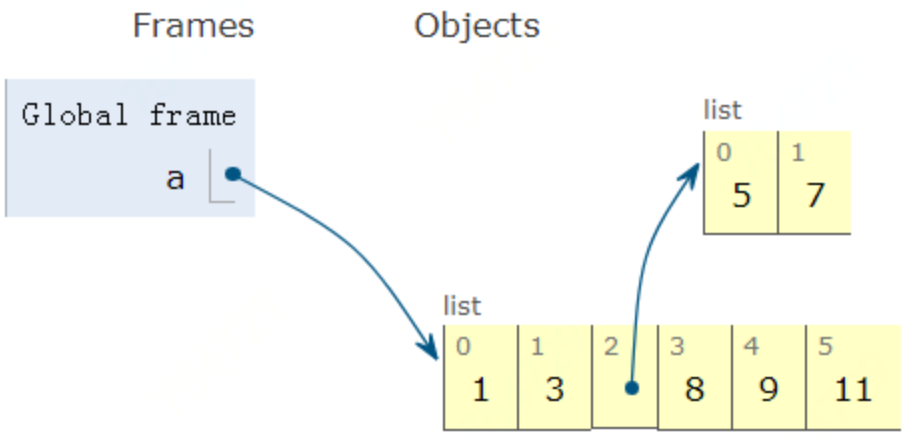
执行 `a.pop()` 后删除最后一个元素：



删除后：



再在索引 3 处增加一个元素 8，`a.insert(3, 8)`，插入后如下：

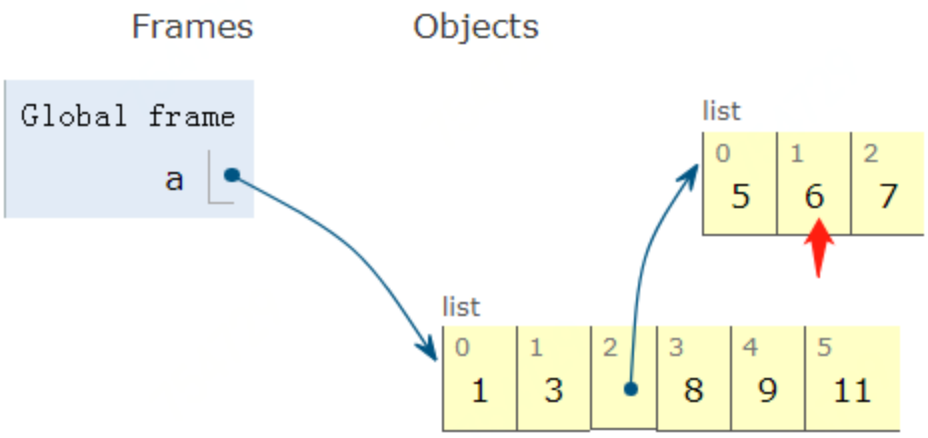


因此，对列表而言，因为它能增加或删除元素，所以它是可变的。

但是，如果**仅仅**在列表 a 中做这一步操作：

```
1 a[2].insert(1, 6)  # 在 a[2]（也是一个列表）中插入元素 6
```

插入后可视化图：



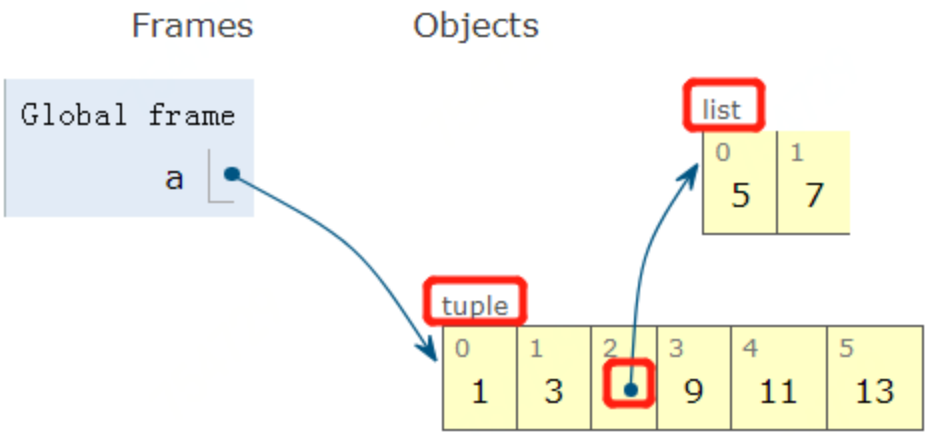
对于**可变**这个概念而言，这就不是真正调整 a 为**可变**的操作。

tuple 就是一个典型的不可变容器对象。对它而言，同样也可以修改嵌套对象的取值，但这并没有真正改变 tuple 内的元素。

如下所示，有一个元组 a：

```
1 a = (1, 3, [5, 7], 9, 11, 13)
```

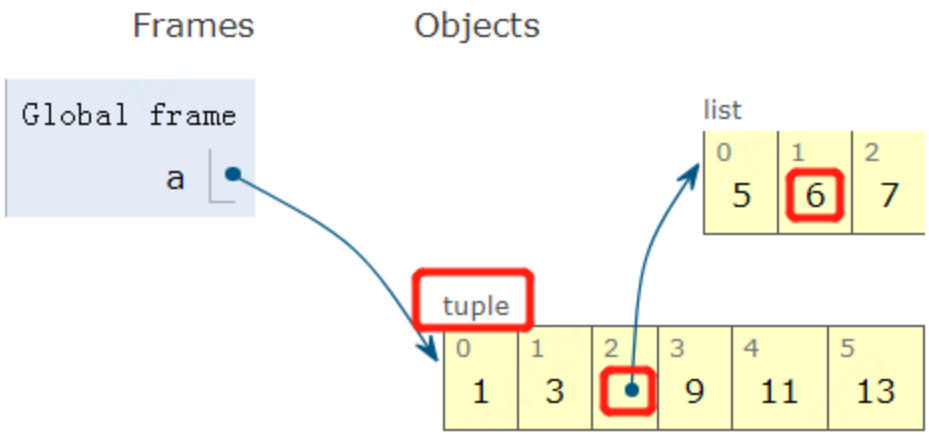
a 的存储示意图如下：



下面插入一个元素 6：

```
1 a[2].insert(1, 6)
```

可以看到，a 内元素没增没减，长度还是 6：



这就是不可变对象的本质，元组一旦创建后，长度就被唯一确定。

但是，对于 list 而言，列表长度会有增有减，所以它是可变的。

3. 小结

今天总结了：

- 列表的基本操作
- 重要深、浅拷贝问题
- 常见的切片操作
- 元组（tuple）的基本操作
- 可变对象，不可变对象