Untitled 3

Lesson 4: 复合数据类型

为了表示数据间的关系且方便我们批量存储数据，python引入了内置的容器类型

列表: list

- 列表是实际程序中最常用也是最灵活的数据结构，它强调的是数据之间的序关系，支持任意位置的插入、删除、修改
- 同个列表里可以存储不同的数据类型，但我们通常不建议这么做
- 在命名一个列表类型的变量时，我们通常不使用xxx_list的方式，而是直接使用xxxs

常用方法

- 插入

```
students = ['Salas', 'Jack', 'Mary']

students.append('Jerry') # 从尾部插入是最常用的插入方式

students.insert(1, 'Bob') # 插入到指定下标的前面
```

- 删除

```
students.pop() # 从尾部删除也是较为常见的方法

students.pop(1) # 也可以用pop(i)来从删除指定位置的元素
```

切片

在字符串部分我们也聊过切片， 列表切片的使用方法与字符串完全相同

但是字符串是不能修改单个字母的(使用replace方法修改事实上是创建了新的字符串对象)，所以我们看不出切片与直接赋值的区别

然而在列表中，直接赋值可以认为仅仅是让不同的变量名**共享了这个列表的控制权(包括读写)**

使用切片则是复制了列表中每个元素的**地址**, 对切片中的**不可变对象**的修改不会影响原列表

```
list1 = [1, 2, 3]

list2 = list1

list1[0] = 5

print(list2)

list2[0] = 7

print(list1)

list3 = list1[:]

list3.pop()

print(list3)

print(list1)
```

列表中的元素也可以是列表，你可以用多个[]来访问嵌套列表中的元素。根据列表切片的复制原理，推测以下代码的运行结果

```
list1 = [[1, 2, 3], [4, 5, 6]]

list2 = list1[:]

list2[-1][-1] = 8

print(list2)

print(list1)
```

```
list2[0] = 'hello'

print(list2)

print(list1)
```

由此可见，列表的切片仅仅复制了列表中每个元素的地址，也就是说**切片中的子列表与原列表子列表仍然是相同的元素(包括值和地址)**，故而直接替换子列表不会对原列表产生影响，而改变子列表中的元素会对子列表产生影响。

如果你确实需要获得一个多重列表或者其他复杂对象的一个拷贝，又不希望对这个拷贝的任何修改影响到原对象，你可以使用**deepcopy**，这一方法会保证你获得一个与原对象完全一致且绝对断绝关系的一个拷贝

```
from copy import deepcopy # 从copy库中引用了deepcopy函数



list1 = [[1, 2, 3], [4, 5, 6]]

list2 = deepcopy(list1)

list2[-1][0] = 5

print(list2)

print(list1)
```

列表与for循环

列表常常和for循环配合使用，以下是两种常见的情况

```
for student in students:

        some code # 对每个学生进行处理
```

```
results = []

for _ in num_xxxs: # _表示占位符，表示用不着的变量

        some code # 处理一些复杂逻辑

        result = ... # 计算一个度量

        results.append(result)
```

列表推导式

列表推导式是一个非常实用的特性，且不说性能方面的收益，至少会让你的代码非常pythonic。

列表推导式允许我们用非常简洁的方式实现列表到列表之间的映射

```
''' bad '''



res = []

for i in range(1, 11):

        temp = i ** 3

        res.append(temp)
```

```
''' good '''
```

```
res = [i ** 3 for i in range(1, 11)]
```

```
''' bad '''


res = []
for student in students:

        res.append(f(student))
```

```
''' good '''


res = [f(student) for student in students]
```

再举一个比较实用的例子

```
''' 生成要爬取的url '''


urls = [f'https://www.cnblogs.com/#p{i}' for i in range(2, 51)]
```

```
urls.insert(0, 'https://www.cnblogs.com/')
```

元组: tuple

元组与列表相似， 也是一种有序的结构，同样可以存储相同或不同类型的元素(实际存储的是地址)，同样支持for循环，同样支持索引和切片。元组与列表的最大区别在于，元组是不可变的结构，一经生成，不允许对直接存储的元素进行任何修改。

```
t = (1, 2, 3)

print(t[0],t[1], t[2])

t[0] = 5 # 不允许的操作
```

但是，对不直接储存的元素，元组的不可变性对其没有约束力

```
t = ([1, 2, 3], [4, 5, 6])

t[0][0] = 8
```

补充：可变对象与不可变对象

- 可变: 在不改变自身地址的情况下可能使内部存储的内容发生变化，即原地修改
- 不可变: 在不改变自身地址的情况下不可能使内部存储的内容发生变化，仅能通过重新构造对象的方式来进行‘伪修改’

这里需要关注两个细节

1. 可不可变强调的能否**原地**修改
2. 可不可变仅针对直接存储的内容，如元组只能保证直接存储的子元素地址不能被修改，如果子元素能做到原地修改，那么元组无法约束其修改。所以如果想要保证一个复杂的对象是绝对不可修改的，必须保证每一层次都满足不可变性。

特殊情况: 单元素元组

```
a = ([1, 2])

print(a) # 此时的a仍是列表而不是元组，因为()被理解为运算优先级



a = ([1, 2],) # 用逗号标注这是一个元组

print(a)
```

解析列表/元组

有时候，我们要从列表/元组中把内部元素还原出来，我们可以使用索引访问，也可以直接用以下方式

```
t = [1, 2]

a, b = t

print(a, b)
```

```
l = (1, 2, 3, 4)

c, *d, e = l # *d的含义是d将打包接受到的参数并以**列表**形式存储()

print(c, d, e)
```

集合: set

有时候，我们并不关心某个元素具体的位置，也不关心后续如何再次找到该元素、修改该元素，我们只关心一个元素是否在某个集合中，只关心在或不在一个属性。我们可以使用集合来表示这种关系。

构造集合

最常见的方式是用列表来构造集合

```
empty = set() # 构造一个空集

s1 = set([2, 3, 1, 5]) # 用列表构造集合

s2 = set([2, 3, 3, 3]) # 集合不会存储重复元素，所以也可以利用这一特性来去重
```

```
''' 列表去重 '''

l = [1, 2, 2]

l = list(set(l)) # 先转为集合去重，再转回列表
```

集合的基本操作

- 添加元素

```
s = set()

s.add('hello')

s.add('name')
```

- 判断元素是否在集合中

```
if 'name' in s:

        print('exist')

if 'mark' in s:

        print('exist')

else:

        print('none')
```

- 删除元素: 删除前务必进行判断!

```
s.remove('hello') # 删除已有元素

s.remove('mark') # 删除不存在元素将报错！所以删除前务必进行判断！
```

- 交、并、差、补

```
s1 = set([1, 2])

s2 = set([2, 3])

s_intersection = s1 & s2 # 取交集

s_union = s1 | s2 # 取并集

s_diffirence = s1 - s2 # 取减集

s_complement = s1 ^ s2 # 取补集
```

字典: dict

字典是服务于查询需求的一种结构。字典不以下标位置(毕竟这东西随时可能随着其他元素的插入删除而改变)来标注一个元素所在的位置，而是事先设置一个关键字(key)来用于查询元素(value),也就是key-value结构。因此，key不允许重复，而value没有限制。因此，也可以把set看成value为key本身的dict。

这种结构的好处在于，查询时间几乎不随数据规模的扩大而上升，能保持一个高效且稳定的查询速度。

基本操作

- 初始化

key不要求一定是字符串(尽管通常是)，只要是不可变对象即可

```
''' 基本格式: {key1:value1, key2:value2...} '''

d = {'John': 99, 'Cindy': 88}
```

- 查询

与列表的索引取值类似，只是[ ]内从下标变为了key

```
''' dict[key] '''

score1 = d['John']
d['John'] += 1
score2 = d['John']
print(score1, score2)
```

查询有失败的风险，会抛出一个KeyError并退出代码，这在实际编程中是很致命的，所以我们常用更安全的get方法来替代直接查询

```
```

```
''' dict.get(key, error_value=None) '''



score3 = d['jack'] # keyError

score3 = d.get('jack') # 找不到该key对应的值，返回None

score4 = d.get('jack', -1) # 也可以手动设定查询失败的返回值，方便外部统一处理

score5 = d.get('Cindy') # 查询成功时与直接查询的结果没有区别
```

for循环与dict

直接对字典进行遍历时，默认对字典的key进行遍历

如果需要对字典的value进行遍历

```
for value in d.values():

        print(value)
```

如果需要同时遍历字典的key与value

```
for key, value in d.items():

        print(key, value)
```

字典的适用场景

- 一方面，字典可以像列表一样存储多个不同的复杂元素，取这些元素的某个特征作为key来建立字典。如用学号作为key(满足唯一性)，把存储学生所有信息的对象作为value
- 另一方面，也可以用字典来表示单个元素，其中以不同的属性名作为key, 属性值作为对应的value

```
student = {'id':2201150506, 'name': Lau, 'score': 80}
```

```
```