# Python Cheat Sheet-2. 数据结构 （Data Structure）-list, tuple, dict, set

?> PCS-2（    ）：2.1 列表（List）、元组（Tuple）和字典（Dictionary），及集合（set）；2.2 列表（List）；2.3 元组（Tuple）；2.4 字典（Dictionary）；2.5 集合（Set）

知识点

描述

代码段

运算结果

备注

**2.1** 列表（List）、元组（Tuple）和字典（Dictionary），及集合（set）

对于一组序列（sequence）数据（有序的），包含有一个或者多个值时，如果要赋予一个变量，则需要使用列表进行存储。同时，可以对数据进行组织管理，即为可变的（mutable），这包括使用索引（index）、分片（slicing）及提供的内置（built-in）函数或方法等途径，实现数据提取、插入、检索和列表间的运算等。

元组类似于列表，只是元组不可修改数据，但可以提取项值，及用内置函数查看数据属性，例如数据长度、最大最小值等。

如果有一组空气污染物浓度的数据，希望可以通过检索污染物名称（string类型）提取数值，而并不要求浓度数据为序列型，则需要使用字典类型存储数据，并可赋予一个变量。对字典实现数据的组织管理，通过对键值操作、内置的函数和方法实现。

集合为一组无序不重复元素集，可以进行交集、差集和并集等操作，在某些算法中可以简化运算复杂程度，例如在“蚁群算法（Ant Colony Optimization，ACO）”中，或旅行商问题，建立禁忌城市集合TabuCitySet，通过差集，每次迭代逐次减去已经走过的城市，那么后续计算只需考虑差集后的集合，至集合为空完成计算。

列表、元组和字典存储的值可以是字符串、数值，也可以是列表、元组或字典自身，或者其它数据类型和结构。对于包含列表的列表为嵌套列表（nested list）。集合通常由列表或字符串转换而来，但是不可包含嵌套列表，否则会提示错误TypeError: unhashable type: 'list'。

究竟选择哪类数据结构用于组织数据分析，需要根据具体分析的内容、途径，优先考虑便于操作的类型或者组合。

当实际工作研究时，对于城市空间数据或用[sklearn（scikit-learn）机器学习](https://scikit-learn.org/stable/)，及[PyTorch](https://pytorch.org/)深度学习时，扩展库[NumPy](https://numpy.org/)和[pandas](https://pandas.pydata.org/)（及[GeoPandas](https://geopandas.org/en/stable/)）提供的数组（array）和表（DataFrame，Series）数据结构无法避免①。

**2.2** 列表（List）

语法为[v0,v1,v2,...,vn]。默认索引关系为从0开始，递增1计算，也可以逆序（Negative Indexing），如:。

* 建立列表

一种是按照语法直接书写并赋予变量，或用list()实现。用内置函数的方法时，如果是字符串会自动切分为独个字母，如何是与range()配合，会产生一个给定始末和步幅值的序列。

letters\_1=['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm']  
print(letters\_1)  
  
letters\_2=list('defghijklm')  
print(letters\_2)  
  
sequence\_range=range(5,20,3) # 内置函数，语法range(start, stop, step)，给定始末和步幅值，构建序列  
print(sequence\_range)  
sequence\_lst=list(range(5,20,3))  
print(sequence\_lst)

['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm']  
['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm']  
range(5, 20, 3)  
[5, 8, 11, 14, 17]

enumerate(iterable, start=0)方法，第一个参数为iterable可迭代对象（列表或元组），第二个参数为计数起始值，默认为0。返回的枚举对象为一个元组列表，每个元组第一个对象为计数值，第二个对象为对应计数的可迭代对象数值。enumerate()函数通常用于循环语句中，循环可迭代对象同时，返回索引值对象，用于计数或者作为其它对应数据提取值时的索引对象。返回的枚举对象<enumerate object at 0x000001C6EDEC1500>可以用list()方法提取具体值。

idx\_value=enumerate(letters\_1,start=0)  
print(idx\_value)  
print(list(idx\_value))

<enumerate object at 0x000001C6EDEE3B00>  
[(0, 'd'), (1, 'e'), (2, 'f'), (3, 'g'), (4, 'h'), (5, 'i'), (6, 'j'), (7, 'k'), (8, 'l'), (9, 'm')]

* 提取值-用索引或者分片（slicing）的方式

给定索引可以直接提取对应索引的值。而slicing的方式需要给始末索引值及步幅值，语法为list[a:b:c]，a为开始值，b为结束值，C为步幅值。三个参数可以配置负值逆序。

下述的案例中，首先用了一个实际研究项目的数据来说明用列表存储多组数据的方式。可以将各组数据单独存储于单个列表中；亦可以用嵌套列表的形式将其存储于一个列表中。不过对于多组数据的存储通常使用dict，array（NumPy）或DataFrame（pandas）的形式处理更加便捷。

在解释数据结构具体操作方式时，使用简单的数据形式，较之具体研究项目的数据更加简单，亦于专注于操作本身。但是同样给出一个具体的案例，来说明在具体数据分析时数据是如何存储的。

多组数据单独存储各自列表的方式。

# 这组数据来自城市环境传感器（AoT, array of things），提取了部分数据，包括字段有：地址（string）、ID编号(string)、经纬度(float)共4类值，以单独列表形式存储。  
coordi\_lon=[-87.628, -87.616, -87.631, -87.59, -87.711, -87.628, -87.586, -87.713, -87.676, -87.624]  
coordi\_lat=[41.878, 41.858, 41.926, 41.81, 41.866, 41.883, 41.781, 41.751, 41.852, 41.736]  
node\_id=['ba46', 'ba3b', 'f02f', 'ba8f', 'ba16', '7e5d', 'ba8b', 'ba13', 'ba18', 'bc10']  
address=['State St & Jackson Blvd Chicago IL', '18th St & Lake Shore Dr Chicago IL', 'Lake Shore Drive & Fullerton Ave Chicago IL', 'Cornell & 47th St Chicago IL', 'Homan Ave & Roosevelt Rd Chicago IL', 'State St & Washington St Chicago IL', 'Stony Island Ave & 63rd St Chicago IL', '7801 S Lawndale Ave Chicago IL', 'Damen Ave & Cermak Chicago IL', 'State St & 87th Chicago IL']

'{0},{1}'.format(v0,v1)该种字符串格式化的方式，可以给定索引值，这样就可以不受位置影响，比较方便的增加{}待格式内容。

如果打印字符串语句比较长，编辑起来不是很方便的话，可以用\将其分成多段处理。打印字符串时，如果需要换行打印显示，则加入\n换行符处理。

# 提取索引值为2对应的数值  
idx\_AoT=2  
idx\_lon=coordi\_lon[idx\_AoT]  
idx\_lat=coordi\_lat[idx\_AoT]  
idx\_nodeID=node\_id[idx\_AoT]  
idx\_address=address[idx\_AoT]  
  
print('values with the index 2: \  
 \nid={0};\nlon={2};\nlat={3};\naddress={1}'.format(idx\_nodeID,idx\_address,idx\_lon,idx\_lat))  
  
# 提取分片数据，slice(2,5)为即为[2,3,4]  
slice\_AoT=slice(2,5) # 默认步幅为1  
print("\_"\*50)  
print(idxes\_AoT)  
slice\_lon=coordi\_lon[slice\_AoT]  
slice\_lat=coordi\_lat[slice\_AoT]  
slice\_nodeID=node\_id[slice\_AoT]  
slice\_address=address[slice\_AoT]  
  
print('values with the index {4}: \  
 \nid={0};\nlon={2};\nlat={3};\naddress={1}'.format(slice\_nodeID,slice\_address,slice\_lon,slice\_lat,list(range(2,5))))

values with the index 2:   
id=f02f;  
lon=-87.631;  
lat=41.926;  
address=Lake Shore Drive & Fullerton Ave Chicago IL  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
slice(2, 5, None)  
values with the index [2, 3, 4]:   
id=['f02f', 'ba8f', 'ba16'];  
lon=[-87.631, -87.59, -87.711];  
lat=[41.926, 41.81, 41.866];  
address=['Lake Shore Drive & Fullerton Ave Chicago IL', 'Cornell & 47th St Chicago IL', 'Homan Ave & Roosevelt Rd Chicago IL']

以嵌套列表的形式存储多组数据。很多时候为了方便获知变量的数据结构，而不用打印print()或者用type()的方式查看数据，则可以在起变量名时，直接增加一个可以表明数据结构的单词或缩写，例如list->lst，dict->dict，DataFrame->df，GeoDataFrame->gdf，array->array等。

AoT\_info\_lst=[['001e0610ba46', -87.627678, 41.87837699999999, 'State St & Jackson Blvd Chicago IL'],   
 ['001e0610ba3b', -87.616055, 41.85813599999999, '18th St & Lake Shore Dr Chicago IL'],   
 ['001e0610f02f', -87.6307578, 41.926261399999994, 'Lake Shore Drive & Fullerton Ave Chicago IL'],   
 ['001e0610ba8f', -87.590228, 41.81034199999999, 'Cornell & 47th St Chicago IL'],   
 ['001e0610ba16', -87.710543, 41.866349, 'Homan Ave & Roosevelt Rd Chicago IL'],   
 ['001e06107e5d', -87.6277685, 41.88320529999999, 'State St & Washington St Chicago IL'],   
 ['001e0610ba8b', -87.58645600000001, 41.7806, 'Stony Island Ave & 63rd St Chicago IL'],   
 ['001e0610ba13', -87.71299, 41.75123799999999, '7801 S Lawndale Ave Chicago IL'],   
 ['001e0610ba18', -87.675825, 41.85217899999999, 'Damen Ave & Cermak Chicago IL'],   
 ['001e0610bc10', -87.62417900000001, 41.73631399999999, 'State St & 87th Chicago IL']]  
print(AoT\_info)

[['001e0610ba46', -87.627678, 41.87837699999999, 'State St & Jackson Blvd Chicago IL'], ['001e0610ba3b', -87.616055, 41.85813599999999, '18th St & Lake Shore Dr Chicago IL'], ['001e0610f02f', -87.6307578, 41.926261399999994, 'Lake Shore Drive & Fullerton Ave Chicago IL'], ['001e0610ba8f', -87.590228, 41.81034199999999, 'Cornell & 47th St Chicago IL'], ['001e0610ba16', -87.710543, 41.866349, 'Homan Ave & Roosevelt Rd Chicago IL'], ['001e06107e5d', -87.6277685, 41.88320529999999, 'State St & Washington St Chicago IL'], ['001e0610ba8b', -87.58645600000001, 41.7806, 'Stony Island Ave & 63rd St Chicago IL'], ['001e0610ba13', -87.71299, 41.75123799999999, '7801 S Lawndale Ave Chicago IL'], ['001e0610ba18', -87.675825, 41.85217899999999, 'Damen Ave & Cermak Chicago IL'], ['001e0610bc10', -87.62417900000001, 41.73631399999999, 'State St & 87th Chicago IL']]

含有括号、大括号的语句，其中的内容可以按逗号分段对其书写，使得代码易读。

print('values with the index 2: \  
 \nid={0};\nlon={2};\nlat={3};\naddress={1}'.format(AoT\_info\_lst[2][0],  
 AoT\_info\_lst[2][1],  
 AoT\_info\_lst[2][2],  
 AoT\_info\_lst[2][3]))

values with the index 2:   
id=001e0610f02f;  
lon=41.926261399999994;  
lat=Lake Shore Drive & Fullerton Ave Chicago IL;  
address=-87.6307578

letters\_1直接敲入的字母，可以用list(map(chr,range(100,110)))语句提取。其中，chr(i, /)，是给定索引值，返回[Unicode](https://home.unicode.org/)② （统一码、万国码、单一码）字符串。map(func, \*iterables)函数，输入的第一个参数func为一个函数对象，第二个参数\*iterables为可迭代对象（列表、元组等），以迭代器方式逐个返回给定迭代对象的函数计算值。因为返回的是迭代器，本例返回值为<map object at 0x000001C6EDDB96D0>，可以通过list(iterate object)方式读取。这个函数可以简单理解为，一次性给定函数多个参数值，返回所有函数计算值。在案例中又给出了两个小案例方便理解。

所有函数，关键字等都可以通过help()函数查看说明文档。

print(map(chr,range(100,110))) # 返回的为迭代对象  
  
lst=list(map(chr,range(100,110)))   
print(lst)  
print("\_"\*50)  
print(chr(100)) # 打印索引值为100的unicode对象  
  
print("\_"\*50)  
lst4len=['python','C++','C','hello guys!']  
string\_len=list(map(len,lst4len))  
print(string\_len)  
  
lst4max=[[1,2],[5,77],[33,65]]  
max\_v=list(map(max,lst4max))  
print(max\_v)

<map object at 0x000001C6EDEDFDF0>  
['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm']  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
d  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
[6, 3, 1, 11]  
[2, 77, 65]

help(map)

Help on class map in module builtins:  
  
class map(object)  
 | map(func, \*iterables) --> map object  
 |   
 | Make an iterator that computes the function using arguments from  
 | each of the iterables. Stops when the shortest iterable is exhausted.  
 |   
 | Methods defined here:  
 |   
 | \_\_getattribute\_\_(self, name, /)  
 | Return getattr(self, name).  
 |   
 | \_\_iter\_\_(self, /)  
 | Implement iter(self).  
 |   
 | \_\_next\_\_(self, /)  
 | Implement next(self).  
 |   
 | \_\_reduce\_\_(...)  
 | Return state information for pickling.  
 |   
 | ----------------------------------------------------------------------  
 | Static methods defined here:  
 |   
 | \_\_new\_\_(\*args, \*\*kwargs) from builtins.type  
 | Create and return a new object. See help(type) for accurate signature.

分片练习

print(lst[3:6:1])  
print(lst[3:6])  
print(lst[3:6:2])  
print(lst[-3:-1])  
print(lst[-3:])  
print(lst[-1])  
print(lst[:3])  
print(lst[:])

['g', 'h', 'i']  
['g', 'h', 'i']  
['g', 'i']  
['k', 'l']  
['k', 'l', 'm']  
m  
['d', 'e', 'f']  
['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm']

元素赋值+分片赋值+删除元素+列表相加+列表的乘法+成员运算符

lst[5]=99 # 元素赋值  
print(lst)  
lst\_none=lst+[None]\*6 # 列表相加，及列表的乘法  
print(lst\_none)  
lst\_none[13]=2015  
print(lst\_none)  
lst\_none[-6:-3]=list(range(100,104,2)) # 分片赋值  
print(lst\_none)  
lst\_none[1:1]=[0,0,0,12]  
print(lst\_none)  
del lst\_none[-2:] # 删除元素  
print(lst\_none)  
  
print([1000, 1200, 1500] in lst\_none) # 成员运算符  
print([1000, 1200, 1500] not in lst\_none) # 成员运算符

['\*', ')', 99, 'h', 'g', 99, [1000, 1200, 1500], 'e', 'd']  
['\*', ')', 99, 'h', 'g', 99, [1000, 1200, 1500], 'e', 'd', None, None, None, None, None, None]  
['\*', ')', 99, 'h', 'g', 99, [1000, 1200, 1500], 'e', 'd', None, None, None, None, 2015, None]  
['\*', ')', 99, 'h', 'g', 99, [1000, 1200, 1500], 'e', 'd', 100, 102, None, 2015, None]  
['\*', 0, 0, 0, 12, ')', 99, 'h', 'g', 99, [1000, 1200, 1500], 'e', 'd', 100, 102, None, 2015, None]  
['\*', 0, 0, 0, 12, ')', 99, 'h', 'g', 99, [1000, 1200, 1500], 'e', 'd', 100, 102, None]  
True  
False

* 内置函数

可以用于列表的内置函数，除了构建列表的list()函数外，可以用len()函数计算列表长度，max()函数返回列表最大值，min()函数返回列表最小值等。表述为函数时，通常是以func(auguments)语法操作，函数可以理解为将要执行的动作，输入参数为被执行的对象。

在练习时，如果手工随机输入一组数据，即使数值比较简单，但是还是会消磨掉一些对技能增长无用的时间，因此练习代码需要随机数值就可以的话，就用代码自动生成一组数据。下述调入random库，使用该库提供的sample(population, k)方法，随机生成一组值，其中第一个参数population为序列（列表、元组等）或集合（set）的一组数；第二个参数为随机提取的数量。返回值为在给定的一组数中随机抽取k个，组成一个新的列表。每运行一次抽取的随机数都会发生改变，如果希望每次运行结果保持一致，则需要用seed(a=None, version=2)固定一个随机种子，关于参数的解释可以用help(random.seed)方法查看。

letters\_lst=list(map(chr,range(60,70)))   
print(lst)  
  
import random   
random.seed(10)  
numericalVals\_lst=random.sample(range(1,200),10)  
print(numericalVals\_lst)  
  
print(help(random.sample))

['<', '=', '>', '?', '@', 'A', 'B', 'C', 'D', 'E']  
[147, 9, 110, 124, 148, 4, 53, 119, 126, 72]  
Help on method sample in module random:  
  
sample(population, k) method of random.Random instance  
 Chooses k unique random elements from a population sequence or set.  
   
 Returns a new list containing elements from the population while  
 leaving the original population unchanged. The resulting list is  
 in selection order so that all sub-slices will also be valid random  
 samples. This allows raffle winners (the sample) to be partitioned  
 into grand prize and second place winners (the subslices).  
   
 Members of the population need not be hashable or unique. If the  
 population contains repeats, then each occurrence is a possible  
 selection in the sample.  
   
 To choose a sample in a range of integers, use range as an argument.  
 This is especially fast and space efficient for sampling from a  
 large population: sample(range(10000000), 60)  
  
None

print(len(letters\_lst),len(numericalVals\_lst))  
print(max(letters\_lst),max(numericalVals\_lst))  
print(min(letters\_lst),min(numericalVals\_lst))

10 10  
E 148  
< 4

* 内置方法

列表的方法使用的语法为lst.method()，包括：

append()，在列表末尾追加新的对象;

extend()，在列表的末尾一次性追加另一个序列中的多个值;

insert()，将对象插入到列表中;

pop()，根据指定的索引值移除列表中的项值，并返回该项值，在默认无参数的条件下移除最后一个;

remove()，移除项值，输入参数为指定的项值而不是索引值;

count()，统计某个元素在列表中出现的次数；

index()，从列表中找到某一个值第一个匹配项的索引位置;

reverse()，翻转列表;

clear()，清空列表;

sort()，是按一定的顺序重新排序;

copy()，复制列表;

查看帮助文件时，使用help(list.method)语法。

lst=list(map(chr,range(100,105)))  
print(lst)  
lst.append(99)  
print(lst)  
lst.append(list(range(50,80,5)))  
print(lst)  
lst\_b=['\*',')','\*']  
print(lst)  
lst.extend(lst\_b)  
print(lst)  
lst.count('\*')  
print(lst)  
lst.index('e')  
print(lst)  
lst.insert(2,[1000,1200,1500])  
print(lst)  
lst.pop(7)  
print(lst)  
lst.remove('\*')  
print(lst)  
lst.reverse()  
print(lst)  
lst.sort() # 将提示错误，TypeError: '<' not supported between instances of 'int' and 'str'

['d', 'e', 'f', 'g', 'h']  
['d', 'e', 'f', 'g', 'h', 99]  
['d', 'e', 'f', 'g', 'h', 99, [50, 55, 60, 65, 70, 75]]  
['d', 'e', 'f', 'g', 'h', 99, [50, 55, 60, 65, 70, 75]]  
['d', 'e', 'f', 'g', 'h', 99, [50, 55, 60, 65, 70, 75], '\*', ')', '\*']  
['d', 'e', 'f', 'g', 'h', 99, [50, 55, 60, 65, 70, 75], '\*', ')', '\*']  
['d', 'e', 'f', 'g', 'h', 99, [50, 55, 60, 65, 70, 75], '\*', ')', '\*']  
['d', 'e', [1000, 1200, 1500], 'f', 'g', 'h', 99, [50, 55, 60, 65, 70, 75], '\*', ')', '\*']  
['d', 'e', [1000, 1200, 1500], 'f', 'g', 'h', 99, '\*', ')', '\*']  
['d', 'e', [1000, 1200, 1500], 'f', 'g', 'h', 99, ')', '\*']  
['\*', ')', 99, 'h', 'g', 'f', [1000, 1200, 1500], 'e', 'd']  
  
  
  
---------------------------------------------------------------------------  
  
TypeError Traceback (most recent call last)  
  
Input In [141], in <cell line: 23>()  
 21 lst.reverse()  
 22 print(lst)  
---> 23 lst.sort()  
  
  
TypeError: '<' not supported between instances of 'int' and 'str'

print(lst)  
  
import random  
r\_lst=random.sample(range(10),6)  
print(r\_lst)  
r\_lst.sort()  
print(r\_lst)  
lst\_copy=r\_lst.copy()  
print(lst\_copy)  
lst\_copy.clear()  
print(lst\_copy)

['\*', ')', 99, 'h', 'g', 'f', [1000, 1200, 1500], 'e', 'd']  
[7, 2, 4, 5, 8, 1]  
[1, 2, 4, 5, 7, 8]  
[1, 2, 4, 5, 7, 8]  
[]

help(list.extend)

Help on method\_descriptor:  
  
extend(self, iterable, /)  
 Extend list by appending elements from the iterable.

**2.3** 元组（Tuple）

元组的语法为(v0,v1,v2,...,vn)，不同于列表的中括号，为小括号表示，中间逗号隔开。元组不能够修改数据是其主要特性。元组的建立方法一种是2,5,6,在成员对象末尾直接加一个逗号；或则使用tuple(iterable=(), /)函数，参数为可迭代对象；亦可直接敲入元组语法(2,5,6)。元组含有两个方法，一个是count()，用于统计给定项值的数量；另一个是index()，用于返回给定项值的第一个出现项值的索引值。内嵌函数除构建函数外与列表同。

对于代码，所有语法中所涉及符号，均为英文格式。

tup=2,5,6,  
print(tup)  
print(type(tup))  
print(type((2,5,6)))  
  
print(3\*(20\*3))  
print(3\*(20\*3,))  
  
tup\_0=tuple([5,8,9])  
print(tup\_0)  
tup\_1=tuple((5,8,9))  
print(tup\_1)  
  
tup\_0N1=tup\_0+tup\_1  
print(tup\_0N1)  
  
print(tup\_0N1.count(5))  
print(tup\_0N1.index(8))  
  
tup\_0N1[2]=9999 # 尝试修改项值时，将提示错误，TypeError: 'tuple' object does not support item assignment

(2, 5, 6)  
<class 'tuple'>  
<class 'tuple'>  
180  
(60, 60, 60)  
(5, 8, 9)  
(5, 8, 9)  
(5, 8, 9, 5, 8, 9)  
2  
1  
  
  
  
---------------------------------------------------------------------------  
  
TypeError Traceback (most recent call last)  
  
Input In [166], in <cell line: 20>()  
 17 print(tup\_0N1.count(5))  
 18 print(tup\_0N1.index(8))  
---> 20 tup\_0N1[2]=9999  
  
  
TypeError: 'tuple' object does not support item assignment

**2.4** 字典（Dictionary）

字典语法为{k0:v0,k1:v1,k2:v2,...,kn:vn}，其中k为键，v为值，键值对之间:冒号分割，大括号括起。键的类型可以为数值型或者字符串类型。字典是无序的，键值对的位置可能会发生变动。

字典键的存在，相当于数据库中用于检索的字段名，方便数据组织、查询、管理，避免像列表在存储数据时，需要根据索引值定位项值，而索引值是不具有自定义含义的默认整数。解释列表应用数据的途径时，使用了实际研究项目的数据，将其转换为三种以字典形式存储数据的方式。第一种，AoT\_info\_dict\_1存储方式，以’id’,‘lon’,‘lat’,‘address’为键，可以快速的提取对应的列表形式的值；第二种，AoT\_info\_dict\_2存储方式，以node\_id列表的’ID’值为键，以对应’ID’值的经纬度和地址组成的元组为值，方便通过’ID’值查询对应信息；第三种，AoT\_info\_dict\_3存储方式，是嵌套字典，第一层仍旧以’ID’值为键，嵌套的内层字典则以’lon’,’lat’和’address’为键，这样处理的好处是，可以根据具有含义的键方便搜索具体细分的值，而不用根据列表或者元组无意义的索引值进行定位。

通过4个列表构建上述三类字典存储形式时，采用了不同的方式，这包括dict()函数构建字典，同时配合使用了zip(\*iterables)函数。zip()函数是将多个序列对象，按照索引值一一对应组合，如下述帮助文件提供的案例。对于嵌套字典的构建方式是列表推导式（comprehensions）的方法，这将在基本语句部分深入解释。

对于复杂的代码语句，如果不能直观的看出具体的逻辑，可以将其拆分为部分，逐部分打印查看数据，及数据的变化，从而理解代码所要解决的逻辑。例如案例中拆分AoT\_info\_dict\_2构建代码。

print(list(zip('abcdefg', range(3), range(4)))) # zip()帮助文件提供的案例

[('a', 0, 0), ('b', 1, 1), ('c', 2, 2)]

coordi\_lon=[-87.628, -87.616, -87.631, -87.59, -87.711, -87.628, -87.586, -87.713, -87.676, -87.624]  
coordi\_lat=[41.878, 41.858, 41.926, 41.81, 41.866, 41.883, 41.781, 41.751, 41.852, 41.736]  
node\_id=['ba46', 'ba3b', 'f02f', 'ba8f', 'ba16', '7e5d', 'ba8b', 'ba13', 'ba18', 'bc10']  
address=['State St & Jackson Blvd Chicago IL', '18th St & Lake Shore Dr Chicago IL', 'Lake Shore Drive & Fullerton Ave Chicago IL', 'Cornell & 47th St Chicago IL', 'Homan Ave & Roosevelt Rd Chicago IL', 'State St & Washington St Chicago IL', 'Stony Island Ave & 63rd St Chicago IL', '7801 S Lawndale Ave Chicago IL', 'Damen Ave & Cermak Chicago IL', 'State St & 87th Chicago IL']  
  
AoT\_info\_dict\_1=dict(zip(['id','lon','lat','address'],[node\_id,coordi\_lon,coordi\_lat,address]))  
print(AoT\_info\_dict\_1)  
  
AoT\_info\_dict\_2=dict(zip(node\_id,zip(coordi\_lon,coordi\_lat,address)))  
print("\_"\*50)  
print(AoT\_info\_dict\_2)  
  
AoT\_info\_dict\_3={id:{'lon':lon,'lat':lat,'address':addr} for id,lon,lat,addr in zip(node\_id,coordi\_lon,coordi\_lat,address)}  
print("\_"\*50)  
print(AoT\_info\_dict\_3)

{'id': ['ba46', 'ba3b', 'f02f', 'ba8f', 'ba16', '7e5d', 'ba8b', 'ba13', 'ba18', 'bc10'], 'lon': [-87.628, -87.616, -87.631, -87.59, -87.711, -87.628, -87.586, -87.713, -87.676, -87.624], 'lat': [41.878, 41.858, 41.926, 41.81, 41.866, 41.883, 41.781, 41.751, 41.852, 41.736], 'address': ['State St & Jackson Blvd Chicago IL', '18th St & Lake Shore Dr Chicago IL', 'Lake Shore Drive & Fullerton Ave Chicago IL', 'Cornell & 47th St Chicago IL', 'Homan Ave & Roosevelt Rd Chicago IL', 'State St & Washington St Chicago IL', 'Stony Island Ave & 63rd St Chicago IL', '7801 S Lawndale Ave Chicago IL', 'Damen Ave & Cermak Chicago IL', 'State St & 87th Chicago IL']}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
{'ba46': (-87.628, 41.878, 'State St & Jackson Blvd Chicago IL'), 'ba3b': (-87.616, 41.858, '18th St & Lake Shore Dr Chicago IL'), 'f02f': (-87.631, 41.926, 'Lake Shore Drive & Fullerton Ave Chicago IL'), 'ba8f': (-87.59, 41.81, 'Cornell & 47th St Chicago IL'), 'ba16': (-87.711, 41.866, 'Homan Ave & Roosevelt Rd Chicago IL'), '7e5d': (-87.628, 41.883, 'State St & Washington St Chicago IL'), 'ba8b': (-87.586, 41.781, 'Stony Island Ave & 63rd St Chicago IL'), 'ba13': (-87.713, 41.751, '7801 S Lawndale Ave Chicago IL'), 'ba18': (-87.676, 41.852, 'Damen Ave & Cermak Chicago IL'), 'bc10': (-87.624, 41.736, 'State St & 87th Chicago IL')}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
{'ba46': {'lon': -87.628, 'lat': 41.878, 'address': 'State St & Jackson Blvd Chicago IL'}, 'ba3b': {'lon': -87.616, 'lat': 41.858, 'address': '18th St & Lake Shore Dr Chicago IL'}, 'f02f': {'lon': -87.631, 'lat': 41.926, 'address': 'Lake Shore Drive & Fullerton Ave Chicago IL'}, 'ba8f': {'lon': -87.59, 'lat': 41.81, 'address': 'Cornell & 47th St Chicago IL'}, 'ba16': {'lon': -87.711, 'lat': 41.866, 'address': 'Homan Ave & Roosevelt Rd Chicago IL'}, '7e5d': {'lon': -87.628, 'lat': 41.883, 'address': 'State St & Washington St Chicago IL'}, 'ba8b': {'lon': -87.586, 'lat': 41.781, 'address': 'Stony Island Ave & 63rd St Chicago IL'}, 'ba13': {'lon': -87.713, 'lat': 41.751, 'address': '7801 S Lawndale Ave Chicago IL'}, 'ba18': {'lon': -87.676, 'lat': 41.852, 'address': 'Damen Ave & Cermak Chicago IL'}, 'bc10': {'lon': -87.624, 'lat': 41.736, 'address': 'State St & 87th Chicago IL'}}

print('node\_id={}'.format(AoT\_info\_dict\_1['id']))  
  
ba8f\_info\_from2=AoT\_info\_dict\_2['ba8f']  
print("\_"\*50)  
print(ba8f\_info\_from2)  
print('ba8f\_lat={},lon={}'.format(ba8f\_info\_from2[1],ba8f\_info\_from2[0]))  
  
ba8f\_info\_from3=AoT\_info\_dict\_3['ba8f']  
print("\_"\*50)  
print(ba8f\_info\_from3)  
print('ba8f\_lat={},lon={}'.format(ba8f\_info\_from3['lat'],ba8f\_info\_from3['lon']))

node\_id=['ba46', 'ba3b', 'f02f', 'ba8f', 'ba16', '7e5d', 'ba8b', 'ba13', 'ba18', 'bc10']  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
(-87.59, 41.81, 'Cornell & 47th St Chicago IL')  
ba8f\_lat=41.81,lon=-87.59  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
{'lon': -87.59, 'lat': 41.81, 'address': 'Cornell & 47th St Chicago IL'}  
ba8f\_lat=41.81,lon=-87.59

拆分代码，易于查看内在数据组织方式。这里需要注意，对于enumerate()，zip()，map()等函数返回值为迭代器（iterator）。迭代器的特点是只能前进，而不能后退，当遍历所有值后，引发StopIteration异常，变量值为空。具体解释查看迭代器部分。这里为了避免list(iterator)读取迭代器值后为空，使用了itertools库提供的tee()方法，将迭代器复制为独立的两份，一份用于打印查看数据；另一份用于后续的输入参数。

AoT\_info\_dict\_2=dict(zip(node\_id,zip(coordi\_lon,coordi\_lat,address)))  
print(AoT\_info\_dict\_2)  
  
  
coordi\_address=zip(coordi\_lon,coordi\_lat,address)  
from itertools import tee  
coordi\_address\_first,coordi\_address\_second=tee(coordi\_address)  
print("\_"\*50)  
print(list(coordi\_address\_first))  
  
id\_coordi\_address=zip(node\_id,coordi\_address\_second)  
print("\_"\*50)  
id\_coordi\_address\_first,id\_coordi\_address\_second=tee(id\_coordi\_address)  
print(list(id\_coordi\_address\_first))  
  
AoT\_info\_dict\_2\_split=dict(id\_coordi\_address\_second)  
print("\_"\*50)  
print(AoT\_info\_dict\_2\_split)

{'ba46': (-87.628, 41.878, 'State St & Jackson Blvd Chicago IL'), 'ba3b': (-87.616, 41.858, '18th St & Lake Shore Dr Chicago IL'), 'f02f': (-87.631, 41.926, 'Lake Shore Drive & Fullerton Ave Chicago IL'), 'ba8f': (-87.59, 41.81, 'Cornell & 47th St Chicago IL'), 'ba16': (-87.711, 41.866, 'Homan Ave & Roosevelt Rd Chicago IL'), '7e5d': (-87.628, 41.883, 'State St & Washington St Chicago IL'), 'ba8b': (-87.586, 41.781, 'Stony Island Ave & 63rd St Chicago IL'), 'ba13': (-87.713, 41.751, '7801 S Lawndale Ave Chicago IL'), 'ba18': (-87.676, 41.852, 'Damen Ave & Cermak Chicago IL'), 'bc10': (-87.624, 41.736, 'State St & 87th Chicago IL')}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
[(-87.628, 41.878, 'State St & Jackson Blvd Chicago IL'), (-87.616, 41.858, '18th St & Lake Shore Dr Chicago IL'), (-87.631, 41.926, 'Lake Shore Drive & Fullerton Ave Chicago IL'), (-87.59, 41.81, 'Cornell & 47th St Chicago IL'), (-87.711, 41.866, 'Homan Ave & Roosevelt Rd Chicago IL'), (-87.628, 41.883, 'State St & Washington St Chicago IL'), (-87.586, 41.781, 'Stony Island Ave & 63rd St Chicago IL'), (-87.713, 41.751, '7801 S Lawndale Ave Chicago IL'), (-87.676, 41.852, 'Damen Ave & Cermak Chicago IL'), (-87.624, 41.736, 'State St & 87th Chicago IL')]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
[('ba46', (-87.628, 41.878, 'State St & Jackson Blvd Chicago IL')), ('ba3b', (-87.616, 41.858, '18th St & Lake Shore Dr Chicago IL')), ('f02f', (-87.631, 41.926, 'Lake Shore Drive & Fullerton Ave Chicago IL')), ('ba8f', (-87.59, 41.81, 'Cornell & 47th St Chicago IL')), ('ba16', (-87.711, 41.866, 'Homan Ave & Roosevelt Rd Chicago IL')), ('7e5d', (-87.628, 41.883, 'State St & Washington St Chicago IL')), ('ba8b', (-87.586, 41.781, 'Stony Island Ave & 63rd St Chicago IL')), ('ba13', (-87.713, 41.751, '7801 S Lawndale Ave Chicago IL')), ('ba18', (-87.676, 41.852, 'Damen Ave & Cermak Chicago IL')), ('bc10', (-87.624, 41.736, 'State St & 87th Chicago IL'))]  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
{'ba46': (-87.628, 41.878, 'State St & Jackson Blvd Chicago IL'), 'ba3b': (-87.616, 41.858, '18th St & Lake Shore Dr Chicago IL'), 'f02f': (-87.631, 41.926, 'Lake Shore Drive & Fullerton Ave Chicago IL'), 'ba8f': (-87.59, 41.81, 'Cornell & 47th St Chicago IL'), 'ba16': (-87.711, 41.866, 'Homan Ave & Roosevelt Rd Chicago IL'), '7e5d': (-87.628, 41.883, 'State St & Washington St Chicago IL'), 'ba8b': (-87.586, 41.781, 'Stony Island Ave & 63rd St Chicago IL'), 'ba13': (-87.713, 41.751, '7801 S Lawndale Ave Chicago IL'), 'ba18': (-87.676, 41.852, 'Damen Ave & Cermak Chicago IL'), 'bc10': (-87.624, 41.736, 'State St & 87th Chicago IL')}

* 字典的基本操作

构建字典可以直接使用字典语法构建，dict()方式键值对构建外，还可以构建空的字典{}后更新字典，或者dict()方式下以赋值的方式给出键值对的形式。

random.sample()是从一个序列中随机抽取数值。random.random()方法是随机生成位于区间[0, 1)内的一个随机数；random.uniform(a, b)方法则是随机生成给定区间内的一个随机数，为浮点型，是否包含区间边界b，依赖于四舍五入的结果。

items\_lst=[(0, [0, 1, 2, 3, 4]), ('key', [[100, 157, 150]]), (2, 'python')]  
items\_dict=dict(items\_lst) # 按照键值对的方式建立字典  
print(items\_dict)  
print(items\_dict[0]) # 根据键提取值  
print(items\_dict['key'])  
  
items\_dict[2]='C' # 指定键，替换新值  
print(items\_dict)  
  
import random  
random.seed(10)  
  
items\_dict[3]=(random.random(),random.uniform(200,300))  
print(items\_dict)  
  
del items\_dict[0] # 删除键值对，该类方法尽量避免使用  
print(items\_dict)  
  
print('key' in items\_dict) # 成员运算符  
print('key' not in items\_dict) # 成员运算符  
  
print("\_"\*50)  
CO\_dict\_1={} # 建立空字典后，更新键值对  
CO\_dict\_1['name']='CO'  
CO\_dict\_1['concentration']=2.1  
CO\_dict\_1['node\_id']='ba46'  
print(CO\_dict\_1)  
  
CO\_dict\_2=dict(name='CO', concentration=2.1, node\_id='ba46') # 以变量赋值的方式构建字典  
print(CO\_dict\_2)  
  
CO\_dict\_2['concentration']+=1 # 用赋值运算符 自加的方式更新对应键的值  
print(CO\_dict\_2)

{0: [0, 1, 2, 3, 4], 'key': [[100, 157, 150]], 2: 'python'}  
[0, 1, 2, 3, 4]  
[[100, 157, 150]]  
{0: [0, 1, 2, 3, 4], 'key': [[100, 157, 150]], 2: 'C'}  
{0: [0, 1, 2, 3, 4], 'key': [[100, 157, 150]], 2: 'C', 3: (0.5714025946899135, 242.88890546751145)}  
{'key': [[100, 157, 150]], 2: 'C', 3: (0.5714025946899135, 242.88890546751145)}  
True  
False  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
{'name': 'CO', 'concentration': 2.1, 'node\_id': 'ba46'}  
{'name': 'CO', 'concentration': 2.1, 'node\_id': 'ba46'}  
{'name': 'CO', 'concentration': 3.1, 'node\_id': 'ba46'}

* 内置函数

len()函数，返回键值对的数量；

sorted()函数，返回排序后的键列表；

any()函数，为如果字典中存在任何对象则返回True，如果为空字典{}则返回False;

all()函数，为如果字典中的键非零或为空字典，则返回True，如果存在键为0，则返回False。

n=6  
dict\_A=dict(zip(random.sample(range(0,10,1),n),random.sample(range(100,110,1),n)))  
print(dict\_A)  
  
print(len(dict\_A))  
print(sorted(dict\_A))  
print(any(dict\_A))  
print(any({}))  
  
print("\_"\*50)  
print(all(dict\_A))  
print(all({}))  
dict\_B={5:0,1:5} # 键非0，而值有0时  
print(all(dict\_B))  
dict\_C={0:7,1:5} # 键有0，而值非0  
print(all(dict\_C))

{9: 102, 0: 100, 3: 107, 7: 109, 6: 108, 2: 101}  
6  
[0, 2, 3, 6, 7, 9]  
True  
False  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
False  
True  
True  
False

* 内置方法

D.clear()，清除字典中所有的键/值项，但是这个方法属于原地操作，并不返回值；

D.copy()， 可以复制字典，但是属于浅复制，当复制的字典已有键/值发生改变时，被复制的字典也会随之发生改变。如果增加新的键值对，则不会发生变化；

D.get()， 可以根据指定的键返回值，但是如果指定的键在被访问的字典中没有，则不返回任何值；

D.items()， 将所有的字典项即键/值对以列表方式返回；

D.keys()， 返回字典的键对象列表；

D.values()， 返回字典的值对象列表；

D.pop()， 根据指定的键返回值，并同时移除字典中对应的键值对；

D.popitem()， 随机弹出键值对，并同时移除字典中对应的键值对；

D.setdefault()， 更新的键值对，其键不在原有字典键列表中，则增加该键值对，否则不增加；

D.update()，用一个字典更新另一字典，键值相同的项将被替换；

记忆字典方法时，最好是按照类似项分组记忆，例如，D.keys()+D.values()+D.items()，D.setdefaulte()+D.update()，D.pop()+D.popitems()等。

关于浅复制（shallow copy）和深复制（deep copy）及赋值变量

在数据分析师，以赋值的方式将一个变量的值传递给一个新的变量，通常用在类定义时值的初始化，一般情况下通常一组值对应一个变量，避免变量混淆。如果要变化原始数据，通常直接操作原始数据后赋予给一个新的变量，原始数据不会受到影响，因此不会用D.copy()，copy.copy()或copy.deepcopy()复制的方法，先复制后操作，这是完全不必要的。但是在有些情况下，例如DataFrame数据结构，即使对原始数据操作赋予给新变量后，原始数据也会发生变化，而不希望改变原始数据时，则需要进行复制。在复制时，可以用字典的内置方法D.copy()实现，也可以用copy中的方法库实现。复制包括浅复制和深复制，从下述代码示例中可以观察到，浅复制会在已有键值对变动下影响已复制对象，反之亦然。深复制则可以理解为复制对象和被复制对象是两个完全不相关的变量，即使值相同。

lst\_A=list(range(6,20,3))  
lst\_B=list(range(100,150,15))  
d={0:lst\_A,1:lst\_B}  
print(d)  
d\_copy=d  
print(d\_copy)  
return\_v=d.clear()  
print(d,d\_copy,return\_v) # 直接赋值时，原变量发生改变，被赋值的变量也会发生变化  
  
d[5]=list(range(1,9,2))  
print("\_"\*50)  
print(d)  
d\_copy=d.copy()  
print(d\_copy)  
d[8]=[5,7]  
print(d,'---',d\_copy) # 用copy()复制的方法，增加新键值对时，赋值的变量不会增加新键值对  
d[5].append(9999)  
print(d,'---',d\_copy) # 用copy()复制的方法，已有键值对发生改变时，赋值的变量已有的键值对发生变化  
d\_copy[5].pop()  
print(d,'---',d\_copy) # 复制对象已有键值对发生变化时，被复制对象也发生变化。  
d\_copy['new\_key']='new\_value'  
print(d,'---',d\_copy) # 复制对象增加新键值对后，被复制对象不会发生变化。  
  
  
print('\_'\*50)  
import copy  
d\_deepCopy=copy.deepcopy(d)  
print(d,'---', d\_deepCopy)  
d[5][0]\*=100  
print(d,'---',d\_deepCopy) # 使用deepcopy()方法，被复制对象不会受到原始字典变动的影响

{0: [6, 9, 12, 15, 18], 1: [100, 115, 130, 145]}  
{0: [6, 9, 12, 15, 18], 1: [100, 115, 130, 145]}  
{} {} None  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
{5: [1, 3, 5, 7]}  
{5: [1, 3, 5, 7]}  
{5: [1, 3, 5, 7], 8: [5, 7]} --- {5: [1, 3, 5, 7]}  
{5: [1, 3, 5, 7, 9999], 8: [5, 7]} --- {5: [1, 3, 5, 7, 9999]}  
{5: [1, 3, 5, 7], 8: [5, 7]} --- {5: [1, 3, 5, 7]}  
{5: [1, 3, 5, 7], 8: [5, 7]} --- {5: [1, 3, 5, 7], 'new\_key': 'new\_value'}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
{5: [1, 3, 5, 7], 8: [5, 7]} --- {5: [1, 3, 5, 7], 8: [5, 7]}  
{5: [100, 3, 5, 7], 8: [5, 7]} --- {5: [1, 3, 5, 7], 8: [5, 7]}

import random  
random.seed(10)  
D=dict(zip(random.sample(range(0,10,1),7),random.sample(range(100,110,1),7)))  
print(D)  
  
print(D.get(9))  
print(D.items())  
print(D.keys())  
print(D.values())  
print(D.pop(0))  
print(D)  
print(D.popitem())  
print(D)

{9: 107, 0: 109, 6: 104, 3: 105, 4: 101, 8: 100, 1: 103}  
107  
dict\_items([(9, 107), (0, 109), (6, 104), (3, 105), (4, 101), (8, 100), (1, 103)])  
dict\_keys([9, 0, 6, 3, 4, 8, 1])  
dict\_values([107, 109, 104, 105, 101, 100, 103])  
109  
{9: 107, 6: 104, 3: 105, 4: 101, 8: 100, 1: 103}  
(1, 103)  
{9: 107, 6: 104, 3: 105, 4: 101, 8: 100}

更新字典的方法——D.setdefault()和D.update()

定义函数时，有时会预先以字典的形式配置一些参数值传入参数，作为默认值，但是会根据具体情况调整某些配置参数，通常会使用D.update（）方法更新字典，即以键为参数名，以键对应的值为参数值。如果需要增加新的参数，则也可以使用D.setdefault()方法，可以避免对原有参数无意中的修改。

random.seed(10)  
print("\_"\*50)  
D=dict(zip(random.sample(range(0,10,1),7),random.sample(range(100,110,1),7)))  
print(D)  
return\_v=D.setdefault(22,[55,66])  
print(return\_v)  
print(D)  
D.setdefault(22,'update value')  
print(D)  
  
print("\_"\*50)  
D.update({22:'update value'})  
print(D)

{9: 107, 0: 109, 6: 104, 3: 105, 4: 101, 8: 100, 1: 103} [55, 66] {9: 107, 0: 109, 6: 104, 3: 105, 4: 101, 8: 100, 1: 103, 22: [55, 66]} {9: 107, 0: 109, 6: 104, 3: 105, 4: 101, 8: 100, 1: 103, 22: [55, 66]} \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ {9: 107, 0: 109, 6: 104, 3: 105, 4: 101, 8: 100, 1: 103, 22: ‘update value’}

这是一组实际研究的数据，在检索百度地图的兴趣点（point of interest，POI）数据时，需要指定query、page\_size、scope等参数值。对于POI数据划分有很多类，为poi\_classificationName列表值，在每次检索数据时，query对象即为分类对象，只可以指定一个类，因此为了避免重复工作，用字典更新的方法更新query参数。

这里用到了后续才会阐述的for循环语句

query\_dic={  
 'query':'旅游景点',  
 'page\_size':'20',   
 'scope':2,  
 }  
  
poi\_classificationName=["美食","酒店","购物","生活服务","丽人","旅游景点","休闲娱乐"]  
for i in poi\_classificationName:  
 query\_dic.update({'query':i})  
 print(query\_dic)

{'query': '美食', 'page\_size': '20', 'scope': 2}  
{'query': '酒店', 'page\_size': '20', 'scope': 2}  
{'query': '购物', 'page\_size': '20', 'scope': 2}  
{'query': '生活服务', 'page\_size': '20', 'scope': 2}  
{'query': '丽人', 'page\_size': '20', 'scope': 2}  
{'query': '旅游景点', 'page\_size': '20', 'scope': 2}  
{'query': '休闲娱乐', 'page\_size': '20', 'scope': 2}

**2.5** 集合（Set）

集合最大的特点是集合中不含重复的元素，如果通过序列数据构建集合时存在重复元素，则构建的集合会只保留一个。集合不具有像列表索引值的属性，不能用索引值提取项值。集合元素的唯一性，及集合间的运算，可以非常方便的处理一些元素具有唯一性变化的数据，例如，下面用了旅行商问题的数据，构建一个城市集合cities\_set，这个数据用列表也可以，示例是说明直接构建集合的方式。tabu\_cities\_set集合包括所有城市，current\_cities\_set建立了一个空集合，当循环cities\_set元素时，current\_cities\_set会通过add()方法增加每次循环的城市名，而tabu\_cities\_set集合则会移除每次循环的城市名，注意，这里-自减运算中，减去的是current\_cities\_set集合，也可以减去set([city])，即每次循环的一个城市，也可以用discard()方法移除。

cities\_set={'Beijing','Chicago','Xian','San Francisco','San Diego'}  
tabu\_cities\_set=set(['Beijing','Chicago','Xian','San Francisco','San Diego'])  
current\_cities\_set=set()  
print(cities\_set)  
print(tabu\_cities\_set)  
print(current\_cities\_lst)  
  
for i,city in enumerate(cities\_set):  
 current\_cities\_set.update([city])  
 tabu\_cities\_set-=set(current\_cities\_set)  
 print("\_"\*50,i)  
 print('current\_cities:{};\ntabu\_citeis:{}'.format(current\_cities\_set,tabu\_cities\_set))

{'Beijing', 'Chicago', 'San Francisco', 'San Diego', 'Xian'}  
{'Beijing', 'Chicago', 'San Francisco', 'San Diego', 'Xian'}  
['Beijing', 'Chicago', 'San Francisco', 'San Diego', 'Xian']  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 0  
current\_cities:{'Beijing'};  
tabu\_citeis:{'Chicago', 'San Francisco', 'San Diego', 'Xian'}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 1  
current\_cities:{'Beijing', 'Chicago'};  
tabu\_citeis:{'San Francisco', 'San Diego', 'Xian'}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2  
current\_cities:{'Beijing', 'Chicago', 'San Francisco'};  
tabu\_citeis:{'San Diego', 'Xian'}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 3  
current\_cities:{'Beijing', 'Chicago', 'San Diego', 'San Francisco'};  
tabu\_citeis:{'Xian'}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 4  
current\_cities:{'Beijing', 'Chicago', 'San Francisco', 'San Diego', 'Xian'};  
tabu\_citeis:set()

* 集合的方法

S.add()，向集合中增加元素；

S.clear()，移除所有元素，成空集合；

S.copy()，复制集合；

S.pop()，返回第一个元素，同时原集合移除该元素。如果为空集合，则引发KeyError异常；

S.remove()，移除一个集合元素，如果输入参数值不在集合中，则引发KeyError异常;

S.discard()，移除一个集合元素，如果输入参数值不在集合中，则无变化，也并提示错误；可以比较S.remove()方法;

S.update()，更新集合，输入参数为序列或集合。如果存在相同元素则保持不变。

lst4set=[1,1,1,2,2,6,33,8,9,55,0,'string',(33,55,99)] # 不能包含嵌套列表，否则提示错误，TypeError: unhashable type: 'list'  
print(lst4set)  
set\_1=set(lst4set)  
print(set\_1)  
  
empty\_set=set() # empty\_dict={}是空字典建立的方法  
print(empty\_set)  
  
print('\_'\*50)  
set\_1.add('new value')  
print(set\_1)  
set\_1.update([1,2,'update'])  
print(set\_1) # 更新时，重复的元素时保持不变的，新元素被增加到集合中  
set\_1.update()  
set\_1.update([1,2,'lst\_v'],{1,32,56,78})   
print(set\_1) # 更新时，可以传入列表，也可以传入集合  
set\_1.remove((33, 55, 99))  
print(set\_1)  
set\_1.remove('new value')  
print(set\_1)  
set\_1.discard(1)  
print(set\_1)  
set\_1.discard(33333) # 如果使用set\_1.remove(33333)，则提示错误为KeyError: 33333。discard方法则忽略  
print(set\_1)  
pop\_v=set\_1.pop() # 返回第一个元素，并从集合中移除  
print(pop\_v)  
print(set\_1)  
set\_1.clear()  
print(set\_1)

[1, 1, 1, 2, 2, 6, 33, 8, 9, 55, 0, 'string', (33, 55, 99)]  
{0, 1, 2, 33, 6, 8, 9, 'string', 55, (33, 55, 99)}  
set()  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
{0, 1, 2, 33, 6, 'new value', 8, 9, 'string', 55, (33, 55, 99)}  
{0, 1, 2, 33, 'update', 6, 'new value', 8, 9, 'string', 55, (33, 55, 99)}  
{0, 1, 2, 33, 'lst\_v', 'update', 6, 'new value', 8, 9, 32, 78, 'string', 55, 56, (33, 55, 99)}  
{0, 1, 2, 33, 'lst\_v', 'update', 6, 'new value', 8, 9, 32, 78, 'string', 55, 56}  
{0, 1, 2, 33, 'lst\_v', 'update', 6, 8, 9, 32, 78, 'string', 55, 56}  
{0, 2, 33, 'lst\_v', 'update', 6, 8, 9, 32, 78, 'string', 55, 56}  
{0, 2, 33, 'lst\_v', 'update', 6, 8, 9, 32, 78, 'string', 55, 56}  
0  
{2, 33, 'lst\_v', 'update', 6, 8, 9, 32, 78, 'string', 55, 56}  
set()

S1.uion(S2)，两个集合的并集运算，同符号|;

S1.intersection(S2)，两个集合的交集运算，同符号&；

S1.difference(S2)，两个（多个）集合的差集运算，同符号-；

S1.symmetric\_difference(S2)，返回两个集合中不重复的元素为一个新集合，同符号^；

S1.intersection\_update(S2)，同S1.intersection(S2)，只是对S1本地更新；

S1.differnce\_update(S2)，同S1.difference(S2)，只是对S1本地更新；

S1.symmetric\_difference\_update(S2)，同S1.symmetric\_difference(S2)，只是对S1本地更新；

S.isdisjoint()，判断两个集合是否包括相同的元素，如果包含返回False，不包含返回True；

S.issubset()，判断一个数据集是否是另一个的子集，同符号<=；

S.issuperset()，判断一个数据集是否是另一个的超集（父集），同符号>=。

S\_1=set([1, 2, 3, 4, 5])  
S\_2=set([4, 5, 6, 7, 8])  
S\_1\_copy1,S\_1\_copy2,S\_1\_copy3=[S\_1.copy() for i in range(3)]  
S\_2\_copy1,S\_2\_copy2,S\_2\_copy3=[S\_2.copy() for i in range(3)]  
print(S\_1)  
print(S\_2)  
print("\_"\*50)  
  
# 并集  
print(S\_1 | S\_2)  
print(S\_1.union(S\_2))  
print('\_'\*50)  
  
# 交集  
print(S\_1 & S\_2)  
print(S\_1.intersection(S\_2))  
print('\_'\*50)  
  
# 差集  
print(S\_1 - S\_2)  
print(S\_1.difference(S\_2))  
print(S\_2-S\_1)  
print(S\_2.difference(S\_1))  
print('\_'\*50)  
  
# symmetric对称差集  
print(S\_1 ^ S\_2)  
print(S\_1.symmetric\_difference(S\_2))  
print('\_'\*50)  
  
print("#"\*50)  
S\_1\_copy1.intersection\_update(S\_2\_copy1)  
print(S\_1\_copy1)  
  
S\_1\_copy2.difference\_update(S\_2\_copy2)  
print(S\_1\_copy2)  
  
S\_1\_copy3.symmetric\_difference\_update(S\_2\_copy3)  
print(S\_1\_copy3)  
  
print("\_"\*50)  
print(S\_1.isdisjoint(S\_2))  
  
print(set([1,2]).issubset(S\_1))  
print(S\_1.issuperset(set([1,2])))

{1, 2, 3, 4, 5}  
{4, 5, 6, 7, 8}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
{1, 2, 3, 4, 5, 6, 7, 8}  
{1, 2, 3, 4, 5, 6, 7, 8}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
{4, 5}  
{4, 5}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
{1, 2, 3}  
{1, 2, 3}  
{8, 6, 7}  
{8, 6, 7}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
{1, 2, 3, 6, 7, 8}  
{1, 2, 3, 6, 7, 8}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
##################################################  
{4, 5}  
{1, 2, 3}  
{1, 2, 3, 6, 7, 8}  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
False  
True  
True

* frozenset()——不可变（immutable）集合

列表为可变（mutable）序列对象，元组为不可变（immutable）序列对象。对于集合而言可以用frozenset()构建不可变集合，支持FS.copy(),FS1.difference(FS2)，FS1.intersection(FS2)，FS1.isdisjoint(FS2)，FS1.issubset(FS2)，FS1.issuperset(FS2)，FS1.symmetric\_difference(FS2)，FS1.union(FS2)等方法，但是不支持remove()，discard()等对集合元素产生变动的方法。

FS\_1=frozenset([1, 2, 3, 4, 5])  
FS\_2=frozenset([4, 5, 6, 7, 8])  
  
print(FS\_1.isdisjoint(FS\_2))  
print(FS\_1.difference(FS\_2))  
print(FS\_1|FS\_2)

False  
frozenset({1, 2, 3})  
frozenset({1, 2, 3, 4, 5, 6, 7, 8})

注释（Notes）：

① sklearn（scikit-learn）机器学习，（<https://scikit-learn.org/stable>）；PyTorch，（<https://pytorch.org>）；NumPy，（<https://numpy.org>）；pandas，（<(https://pandas.pydata.org>）；GeoPandas，（<https://geopandas.org/en/stable>）。

② Unicode，The Unicode Standard，是一个信息技术标准，用于对世界上大多数书写系统所表达的文本进行统一编码、表示和处理。该系统由Unicode Consortium 维护，截至目前的版本（15.0）共定义了149,186个字符，涵盖了161种现代和历史文字，以及符号、表情符号（包括颜色）和非视觉控制和格式化代码（<https://home.unicode.org/>）。

PC2-ipynb download