

中山大学计算机学院人工智能本科生实验报告

2022学年春季学期

课程名称：Artificial Intelligence

教学班级	人工智能（陈川）	专业（方向）	计算机科学与技术人工智能与大数据
学号	20337025	姓名	崔璨明

一、实验题目

罗马尼亚旅行问题：请基于上周你编写的最短路径程序，扩展实现一个搜索罗马尼亚城市间最短路径的导航程序，要求如下：

1. 出发城市和到达城市由用户在查询时输入
2. 对于城市名称，用户可以输入全称，也可以只输入首字母，且均不区分大小写
3. 向用户输出最短路径时，要输出途经的城市，以及路径总路程
4. 输出内容在直接反馈给用户的同时，还需追加写入一个文本文件中，作为记录日志
5. 为提升代码灵活性，你应在代码中合理引入函数和类（各定义至少一个）
6. 此外，将你定义的一些函数和类，存储在独立的模块文件中

思考题：

1. 如果用列表作为字典的键，会发生什么现象？用元组呢？
2. 在本课件第2章和第4章提到的数据类型中，哪些是可变数据类型，哪些是不可变数据类型？试结合代码分析。

二、实验内容

1、算法原理

查询出发城市和到达城市之间的距离问题，实际上是查找图中两个点的最短路径问题，且每个城市之间可以双向通行、路程不为负数、不完全相同，因此是求无向带权图两点的最短路径问题。故采用Dijkstra算法来编写程序以解决问题，而Dijkstra算法是一种贪心算法，其算法原理如下：首先我们用邻接矩阵的方式来储存罗马尼亚的旅行图 $G(V,E)$ ，得到起点和终点后，算法运行原理如下：

1. 把图中顶点集合 V 分成两组，第一组为已求出最短路径的顶点集合（用 S 表示，初始时 S 中只有起点，以后每求得一条最短路径，就将加入到集合 S 中，直到全部顶点都加入到 S 中，算

法就结束了)，第二组为其余未确定最短路径的顶点集合（用U表示），按最短路径长度的递增次序依次把第二组的顶点加入S中。在加入的过程中，总保持从源点v到S中各顶点的最短路径长度不大于从源点v到U中任何顶点的最短路径长度。此外，每个顶点对应一个距离，S中的顶点的距离就是从v到此顶点的最短路径长度，U中的顶点的距离，是从v到此顶点只包括S中的顶点为中间顶点的当前最短路径长度。

2. 选出U中距离S中的顶点距离最短的顶点k，将k从U中删除并加入S中。
3. 更新S中顶点到U中顶点的距离，之所以能够更新，是因为加入了顶点k，可以使用判断 $(s, v) > (s, k) + (k, v)$ 来更新顶点距离
4. 重复步骤2和3，直到起点到所有顶点的最短距离都被找出来，然后输出到终点的距离即可。

2、算法伪代码

```
//结构vertex用于存储每个节点的信息和实现Dijkstra算法需要用到的信息
struct vertex{
    list adj    //用邻接表存图
    bool known  //记录每个节点是否已访问
    Distype dist //到每个节点距离
    Vertex path  //输出路径
}

//打印路径的函数
my_print(vertex v){
    if v.path :
        my_print(v.path)
        print("-->")

    print(v)
}
```

算法名称: `dijkstra(vertex s, start, end)`

input: 一个无向带权图s，起点start，终点end

output: 最短距离shortest和打印起点到终点的路径

```
dijkstra(vertex s, start, end):
    for each Vertex v:
        v.dist = MAX_NUM
        v.known = false
```

```

start.dist=0

while there is an unknown distance vertex:
    vertex v = smallest unkown distance vertex  //最近的节点
    v.known = true
    for each vertex w adjacent to v:
        if !w.known:
            Disttype cvw = costof edge from v to w
            if v.dist + cvw < w.dist:
                w.dist = v.dist + cvw  //更新距离
                w.path=v

my_print(end)
print(end.dist)
return

```

该伪代码是从顶点出发存储各种信息，并给每个顶点安排一个邻接表，但在该实验题的情况下，可能会增加算法的空间效率，因此我在优化程序时对其存储结构进行了优化，具体内容将在下文**创新点&优化**中介绍

3、关键代码展示

对伪代码的存储结构进行优化后，定义一个`roman_map`类，里面存有三个字典，分别建立起城市名字到编号的映射、编号到城市名字的映射、城市名和其缩写的映射、城市数量、道路数量和用

邻接矩阵存储的图的基本信息。在初始化类时，还对邻接矩阵的值进行初始化：

```
class roman_map():
    def __init__(self):#初始化
        self.book={}      #城市名字到编号的映射
        self.book2={}     #编号到城市名字的映射
        self.book3={}     #城市名与缩写的映射
        self.index=0
        self.m=20          #城市的数量
        self.n=23          #城市间道路的数量
        self.grp=[]        #用邻接矩阵存图
        for s1 in range(0,self.m):
            self.grp.append([])
            for s2 in range(0,self.m):
                if(s1==s2):
                    self.grp[s1].append(0)
                else:
                    self.grp[s1].append(float("inf"))#将边初始化为最大值
```

在`roman_map`类中定义一个成员函数`read_message`，用于读取文本文件中存储的图的信息：

```
def read_message(self,file_name):#读取文件信息以建立图
    file_object=open(file_name)
    i=0

    for line in file_object.readlines():
        if i!=0:
            v1,v2,w=line.split()
            w=int(w)
            v1=v1.lower()
            v2=v2.lower()

            self.book3[v1[0]]=v1
            self.book3[v2[0]]=v2

            if v1 in self.book.keys(): #输入时采用字典将城市名字转变为数字表示顶点编号
                v1=self.book[v1]

            else:
                self.book[v1]=self.index
                self.book2[self.index]=v1
                v1=self.index
                self.index+=1

            if v2 in self.book.keys():
                v2=self.book[v2]
            else:
                self.book[v2]=self.index
                self.book2[self.index]=v2
                v2=self.index
                self.index+=1
```

关键的算法Dijkstra算法实现部分如下，传入参数为`roman_map`类，起点和终点，具体代码内容和注释如下：

#dijkstra算法执行过程

```
def dj(a,sta,end):
    num=a.m
    dis=[Maxnum]*num
    dis[sta]=0          #dis[]存储距离
    qianzui=[-1]*num    #为输出路径，记录前缀
    vis=[0]*num
    u=0
    for i in range(num):
        min=Maxnum
        for j in range(num):
            if not vis[j] and dis[j]<min:
                min=dis[j]
                u=j
        #u为可到达的顶点中距离最小的
        vis[u]=True
        for k in range(num):
            if a.grp[u][k]<Maxnum:
                if dis[u]+a.grp[u][k]<dis[k]:
                    dis[k]=dis[u]+a.grp[u][k]
                    qianzui[k]=u          #记录前驱以输出路径
    #输出最短路程
    print(dis[end])
    #输出路径信息
    l=end
    lujin=a.book2[sta].title()+"-->"
    ans=[]
    city=0
    while l !=sta:
        ans.append(a.book2[l])
        city+=1
        l=qianzui[l]
    for i in range(city-1,-1,-1):
        lujin+=ans[i].title()
        if i!=0:
            lujin+='-->'
    print(lujin)
```

主函数执行部分如下：

```
if __name__ == '__main__':
    rm=ch.roman_map()
    rm.read_message("Romania.txt") #调用函数读入文件
    while 1:

        mess=input("enter two city(enter 'q' to exit):")
        if mess=='q': #输入q时退出程序
            break
        s,e=mess.split() # 对输入信息分割得到起点和终点

        s=s.lower()
        e=e.lower() #统一用小写形式存储

        #若输入为首字母，则将首字母转化为城市全称
        if len(s)==1:
            s=rm.book3[s]
        if len(e)==1:
            e=rm.book3[e]

        #搜索路径并输出
        fh.dj(rm,rm.book[s],rm.book[e])
```

关键的代码展示如上所述，为了使程序更加美观，我在优化程序的时候调用了python中的 **tkinter** 库，建立了可视化的界面，具体内容将在下文**创新点&优化**中介绍。

4、创新点&优化

创新点：

1. 在完成罗马尼亚旅行实验的过程中，题目要求用户可通过城市名字、城市名字的缩写来查询城市之间的最短路径，且无论大小写都可以，因此，为了提高算法的时间效率，我选择了运用三个字典，分别建立起城市名字到编号的映射、编号到城市名字的映射、城市名和其缩写的映射的方法来实现快速查找，虽然这种做法增加了程序所占用的空间，但避免了在字典中进行遍历的方法来实现双向映射的方法，大大提高了程序的运行效率。
2. 为使程序调用的界面更加美观和增加其工具性、我调用了python中的 **tkinter** 库，建立了可视化的界面，对原始程序进行了加工，因为界面简单，所以增加的代码量并不多，但实现的效果相比起黑幕的终端运行程序改善了许多，增加的代码段和实现效果如下：

```
window=tk.Tk()
window.title('罗马尼亚旅行问题')
window.geometry('700x500')

show=tk.Label(window,text='Enter two cities:')
show.grid(row=1,column=1,padx=10, pady=10)
show_first=tk.Label(window,text='From:').grid(row=4, column=4, padx=10, pady=5)
show_second=tk.Label(window,text='To:').grid(row=5, column=4, padx=10, pady=5)

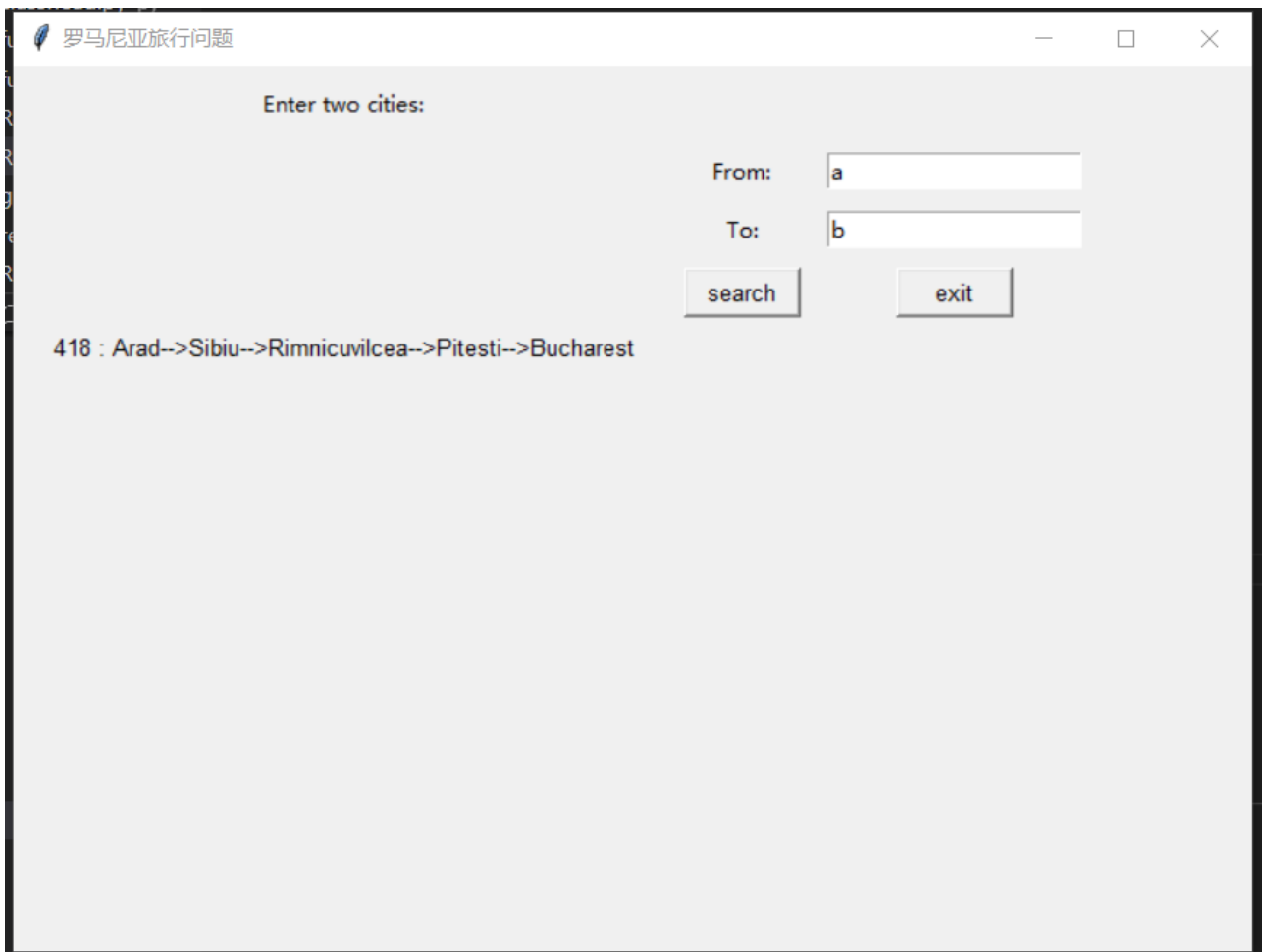
    #show_first.pack(padx=10, pady=20)
    #show_second.pack(padx=10, pady=40)
ip1=tk.Entry(window,show=None)
ip1.grid(row=4, column=5, padx=10, pady=5)
ip2=tk.Entry(window,show=None)
ip2.grid(row=5, column=5, padx=10, pady=5)
    #ip1.pack(padx=20, pady=20)
    #ip2.pack(padx=20, pady=40)

find_button=tk.Button(window,text='search',font=('Arial',10),width=7,height=1,command=run)
find_button.grid(row=6,column=4,padx=5, pady=5)

exit_button=tk.Button(window,text='exit',font=('Arial',10),width=7,height=1,command=efun)
exit_button.grid(row=6,column=5,padx=5, pady=5)

outmess=""

showmess=tk.StringVar()
showmess.set(outmess)
show=tk.Label(window,textvariable=showmess,font=('Arial',10))
show.grid(row=8,column=1,padx=20, pady=1)
```

优化程序

1. 我在一开始伪代码是从顶点出发存储各种信息，并给每个顶点安排一个邻接表，但根据伪代码来编写算法时，我发现该实验题的情况下，原伪代码的存储结构可能会增加算法的空间效率，因为我一开始的构想是给节点设计一个结构体，里面存有节点的邻接表和各种信息，以方便算法的进行，但在编写程序时，我想到用一个`graph`类来对节点进行统一管理或许能够大大减少内存的开销，因此我在优化程序时对其存储结构进行了优化，将`struct vertex`改成了统一用`class roman_map`来进行管理、读取文件信息，而算法的设计思路和执行过程保持不变。对我一开始编写的算法伪代码进行优化后，程序的代码量大大减少，而且编写程序时也更方便了。
2. 在一开始的Dijkstra算法中，在每次寻找未访问集合U中到已访问集合S的最短路径节点时，我用的是遍历dis数组的方法，这种方法的复杂度较大，为 $O(n)$ ，因此我优化了算法的该部分，利用了python中的优先队列`PriorityQueue`，将节点信息存入优先队列中，队列中节点按距离从小到大进行排序，因此每次只需调用`get()`函数获取队首即可，提高了算法的运行效率，具体改进代码如下：

改进前：

```

for i in range(num):
    min=Maxnum
    for j in range(num):
        if not vis[j] and dis[j]<min:
            min=dis[j]
            u=j
    #u为可到达的顶点中距离最小的
    vis[u]=True

```

改进后：

```

class Node:
    def __init__(self,a1,a2):
        self.id=a1
        self.distance=a2

    def __lt__(self,other):
        return self.distance < other.distance

```

```

    q.put(shuz[i])
    for i in range(num):
        #u为可到达的顶点中距离最小的
        u=q.get().id

        vis[u]=True
        for k in range(num):
            if a.grp[u][k]<Maxnum:
                if shuz[u].distance+a.grp[u][k]<shuz[k].distance:
                    shuz[k].distance=shuz[u].distance+a.grp[u][k]
                    qianzui[k]=u    #记录前驱以输出路径
#输出最短路程

```

三、实验结果及分析

实验结果展示示例

在该部分对编写的程序进行样例测试，首先是实验题目中给出的标准测试案例：

2. 分析罗马尼亚旅行问题的实验结果，至少展示以下城市间的路径：

- i. Arad → Bucharest
- ii. Fagaras → Dobreta
- iii. Mehadia → Sibiu

1、输入Arad Bucharest，程序运行结果如下：

The screenshot shows a window titled "罗马尼亚旅行问题" (Romanian Travel Problem). It contains a form with the label "Enter two cities:". There are two input fields: "From:" with the value "Arad" and "To:" with the value "Bucharest". Below these fields are two buttons: "search" and "exit". The output of the search is displayed at the bottom: "418 : Arad-->Sibiu-->Rimnicuvillea-->Pitesti-->Bucharest".

可以看见，Arad到Bucharest之间的最短路径长度为418，路径为Arad→Sibiu-->Rimnicuvillea→Pitesti→Bucharest

2、输入Fagaras Dobreta，程序运行结果如下：

The screenshot shows the same window as before, but with "From:" set to "Fagaras" and "To:" set to "Dobreta". The "search" button has been clicked, and the output is "445 : Fagaras-->Sibiu-->Rimnicuvillea-->Craiova-->Dobreta".

可以看见，Fagaras到Dobreta之间的最短路径长度为445，路径为Fagaras→Sibiu-->Rimnicuvillea→Craiova→Dobreta

3、输入Mehadia Sibiu，程序运行结果如下：

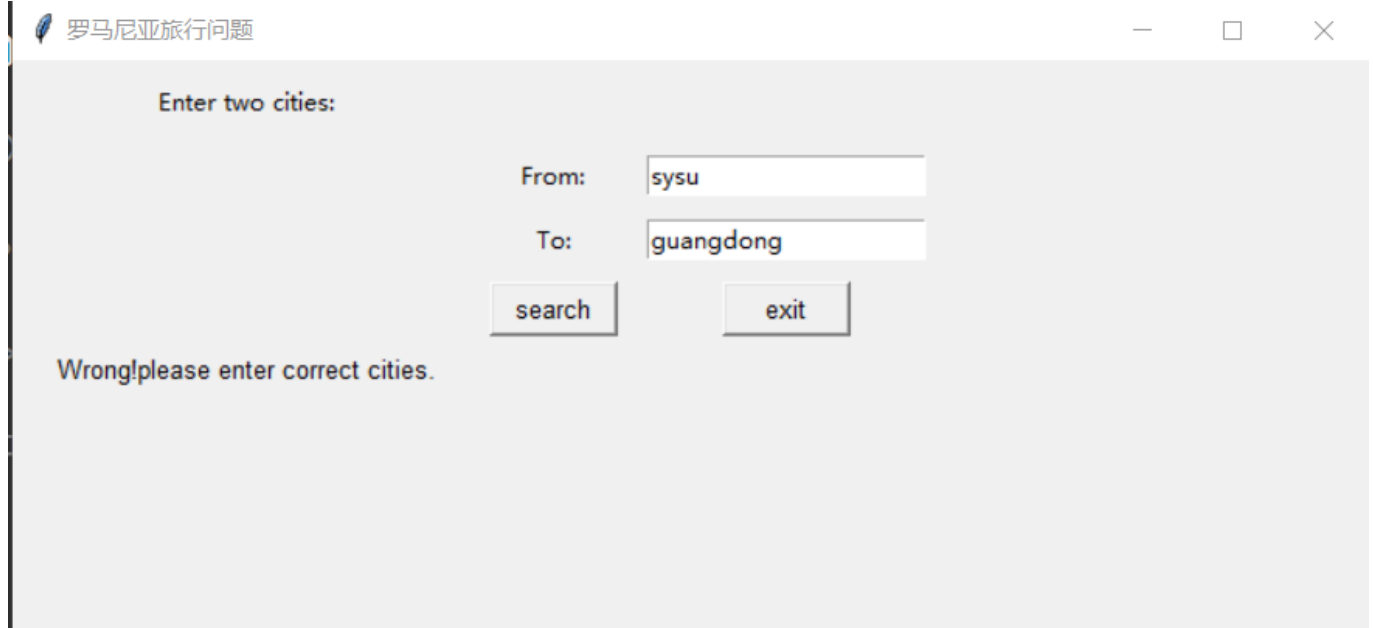
The screenshot shows the same window, but with "From:" set to "Mehadia" and "To:" set to "Sibiu". The "search" button has been clicked, and the output is "421 : Mehadia-->Dobreta-->Craiova-->Rimnicuvillea-->Sibiu".

可以看见，Mehadia到Sibiu之间的最短路径长度为421，路径为Mehadia→Dobreta-->Craiova→Rimnicuvillea→Sibiu

按照实验题目要求，查询的记录将会被写入文本文件record.txt中，record.txt中的查询记录如下：

```
Arad-->Sibiu-->Rimnicuvillea-->Pitesti-->Bucharest:418
Arad-->Sibiu-->Rimnicuvillea-->Pitesti-->Bucharest:418
Bucharest-->Pitesti-->Rimnicuvillea-->Sibiu-->Arad:418
Arad-->Sibiu-->Rimnicuvillea-->Pitesti-->Bucharest-->Urziceni-->Hirsova:601
Arad-->Sibiu-->Rimnicuvillea-->Pitesti-->Bucharest:418
Fagaras-->Sibiu-->Rimnicuvillea-->Craiova-->Dobreta:445
Mehadia-->Dobreta-->Craiova-->Rimnicuvillea-->Sibiu:421
Arad-->Sibiu-->Rimnicuvillea-->Pitesti-->Bucharest:418
Arad-->Sibiu-->Rimnicuvillea-->Pitesti-->Bucharest:418
Fagaras-->Sibiu-->Rimnicuvillea-->Craiova-->Dobreta:445
Mehadia-->Dobreta-->Craiova-->Rimnicuvillea-->Sibiu:421
```

4、错误的输入处理：当查询的城市名字不存在时，程序将会发出提示，并不会运行Dijkstra函数：



5、程序运行占用的空间和时间

总程序包括两个头文件和一个运行文件，分别为classhead.py、funhead2.py和Roman2.py，总大小为7KB，在程序中引入datetime库，并加入以下代码以计算查询一次的时间：

```
starttime = datetime.datetime.now()
```

```
endtime = datetime.datetime.now()  
print((endtime - starttime))
```

得到查询一次的时间为：

```
Arad-->Sibiu-->Rimnicuvillea-->Pitesti-->Bucharest  
0:00:00.001996
```

时间足够小，符合程序高效的要求。

四、思考题

1、如果用列表作为字典的键，会发生什么现象？用元组呢？

答：python中列表不能作为字典的键，因为python中字典原理是哈希表，因此的键的对象必须是可哈希的对象。像列表和字典这样的可变类型，由于它们不是可哈希的，所以不能作为键。而用列表作为字典的键时，会报错：

```
Traceback (most recent call last):  
  File "e:\VSCODE\py\tempCodeRunnerFile.py", line 3, in <module>  
    s[a]=7  
TypeError: unhashable type: 'list'
```

而python中可以使用元组作为字典中的键值，因为元组也是不可变的，是可哈希的：

```
1  s={}  
2  a=(2,3,4,6)  
3  s[a]=7  
4  print(s[a])
```

```
PS E:\VSCODE\py> python -u "e:\VSCODE\py\tempCodeRunnerFile.py"  
7  
PS E:\VSCODE\py> █
```

2、在本课件第2章和第4章提到的数据类型中，哪些是可变数据类型，哪些是不可变数据类型？试结合代码分析。

答：int类型是不可变数据类型，测试代码如下：

```
a=2
print(id(a))
b=2
print(id(b))
a=3
print(id(a))
```

可以看到，当值发生改变时，地址也发生了变化：

```
PS E:\VSCODE\py> python -u "e:\VSCODE\py\tempCodeRunnerFile.py"
2482699200848
2482699200848
2482699200880
```

同样，用以下代码测试，发现float也为不可变数据类型：

```
a=2.3
print(id(a))
b=2.3
print(id(b))
a=3.2
print(id(a))
```

```
2178983033968
2178983033968
2178983033904
PS E:\VSCODE\py> █
```

同理，字符串也为不可变数据类型：

```
a="asasa"
print(id(a))
b="asasa"
print(id(b))
a="hhhhh"
print(id(a))
```

```
PS E:\VSCODE\py> python -u "e:\VSCODE\py\tempCodeRunnerFile.py"
1980745072240
1980745072240
1980774624880
```

用同样的测试方法可知，不可变数据类型还有bool类型，元组类型

而列表属于可变类型：

```
a=[2,3,4]
print(id(a))
b=[2,3,4]
print(id(b))
a[1]=6
print(id(a))
```

```
PS E:\VSCODE\py> python -u "e:\VSCODE\py\tempCodeRunnerFile.py"
2350113674624
2350113934080
2350113674624
```

可以发现，在id(内存地址)不变的情况下，value（值）发生了改变，为可变数据类型。同样，字典也为可变数据类型：

```
a={1:'a',2:'b'}

print(id(a))

b={1:'a',2:'b'}

print(id(b))

a[1]='e'

print(id(a))
```

```
PS E:\VSCODE\py> python -u "e:\VSCODE\py\tempCodeRunnerFile.py"
3069137148224
3069137148416
3069137148224
PS E:\VSCODE\py> █
```

总结：在第2章和第4章提到的数据类型中，int、float、bool、str、tuple是**不可变数据类型**，list列表、字典是**可变数据类型**。

五、实验总结和感想

通过这次实验，我掌握了python的基本语法、常用数据类型和函数等知识，并能通过编写python程序来解决具体问题，实现特定的算法，除此之外，在完成实验的过程中我也遇到了许多困难，但通过学习来将这些困难解决后我感到受益匪浅，收获了许多。在实验过程中，我还养成了优化算法、提示效率等习惯，以后将会继续保持下去。

六、参考资料

- 《数据结构与算法分析——C++语言描述（第四版）》 作者：【美】Mark Allen Weiss
- [python学习——查看程序运行时间_知北行的博客-CSDN博客_python查看运行时间](#)