**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 |  | 专业（方向） | **ICS** |
| 学号 | **20337268** | 姓名 | **张文沁** |

# 实验题目

Python程序设计基础——最短路径搜索

# 实验内容

1. 算法原理

Dijkstra算法

Dijkstra算法通过贪心思想实现，即把顶点分为两个集合，一个（A）存放已经找到最短路径的点，另一个（U）存放没有找到的点，每次求得一个顶点的最短路径就更新两个集合，直到所有点都在已经找到的集合中。

先把起点到所有点中最短的（v）找到，放入A，以v为中间点，考虑起点通过v到达v的可达点的距离是否小于不通过中间点的距离，然后对路径进行更新，并以v为新的起点，刚刚找到的v的可达点为新的中间点，重复上述内容，直到所有点都在A中。

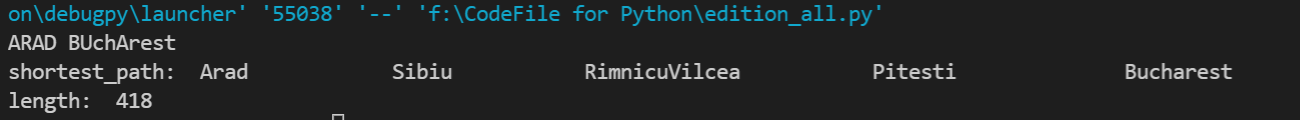
以下是算法的核心部分伪代码：

1. Dijkstra(G, dis[], s)：
2. #初始化;
3. **for**(n)：#循环n次
4. u = min(dis) && unvisited[]#使dis[u]最小的还未访问的顶点的标号;
5. visited[u] = 1 #记u已被访问;
6. **for**(all accessible[u,v] v) #从u出发能到达的所有顶点v
7. {
8. **if**(v **is** **not** visited && dis[s,u] + dis[u,v] < dis[s,v])： #以u为中介点使s到顶点v的最短距离d[v]更优
9. 优化d[v]
10. path.append(u)
11. 伪代码
12. **from** dis **import** dis
13. **from** itertools **import** count
15. **from** numpy **import** append

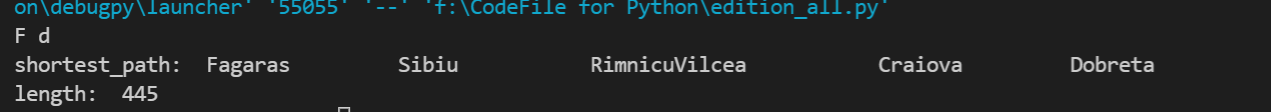
18. max  = 99999
19. m,n = input()
20. #读入节点数和边数
21. nodes = []
22. #定义点的数组
23. dic = {}
24. #定义点和数字的对应关系
25. matrix = [m][n] = 0
26. #初始化
27. node\_1,node\_2,value = input()
28. #左边、右边、权值
30. #建造邻接矩阵
31. **for** i **in** range(n):
32. #对n个点都进行对应，下述代码1换成2，2换成1并列选择执行
33. **if**(node\_1 **not** **in** nodes):
34. nodes.append(node\_1)
35. dic[node\_1] = count
36. count += 1
37. matrix[dic[node\_1]][dic[node\_2]] = value
39. a,b = input().split()
40. #读入待解决的点
41. state[n] = 0
42. #是否已经找到最短路径
43. length[m] = matrix[dic[a]][dic[[node]]
44. #初始化
45. pre\_able[m] = a
46. #前驱点，方便找到路径
47. **while** 0 **in** state:
48. **for** node1 **in** nodes:
49. #第一次先找到可达最短点
50. **if** state[dic[node1]] == 0 && length[dic[node1]] < dis:
51. dis = length[dic[node1]]
52. node = node1
53. state[dic[node]] = 1
54. **for** node2 **in** nodes:
55. #核心算法，进行更新
56. **if** state[dic[node2]] == 0 &&length[dic[node]] + matrix[dic[node2]][dic[node]] < length[dic[node2]]:
57. length[dic[node2]] = length[dic[node]] + matrix[dic[node2]][dic[node]]
58. pre\_able[dic[node2]] = node
59. shortest\_path = []
60. #存储路径的数组
61. node\_back = b
62. #存储下一个节点
63. **while** node\_back != a:
64. shortest\_path.append(node\_back)
65. node\_back = pre\_able[dic[node\_back]]
66. shortest\_path.append(a)
68. **print**(path,shortest\_path)
69. 关键代码展示（带注释）
    * 1. 核心代码：
70. **for** node **in** nodes:
71. length[dic[node]] = matrix[dic[a]][dic[node]]
72. #最短路径初始化为其他点到它的距离，直到所有点都遍历完，结束循环
73. **while** 0 **in** state:
74. #找到未遍历的点中，与已遍历的点距离最近的那一个
75. node = a
76. dis = max #初始比较距离
77. **for** node1 **in** nodes: #nodes是所有点
78. **if** state[dic[node1]] == 0 **and** int(length[dic[node1]]) < int(dis):#第一次是把可达点都遍历一遍
79. dis = length[dic[node1]]
80. node = node1
81. #找到可达点able进行更新
82. #将此点标记，并更新其它未遍历点的length与able
83. state[dic[node]] = 1
84. **for** node2 **in** nodes: #nodes是所有点
85. **if** state[dic[node2]] == 0 **and** int(length[dic[node]] )+ int(matrix[dic[node2]][dic[node]]) < int(length[dic[node2]]):#第二遍判断作为中间节点是否可以缩短路径
86. length[dic[node2]] = int(length[dic[node]]) + int(matrix[dic[node2]][dic[node]])
87. pre\_able[dic[node2]] = node
88. #更新able在最短路径上的点
89. #short\_path记录点a到点b的最短路径
90. node\_back = b
91. **while** node\_back != a:
92. shortest\_path.append(node\_back)
93. #压入其中，顺序方便输出
94. node\_back = pre\_able[dic[node\_back]]
95. #通过pre\_able可以快速找到前驱节点，简化了时间
96. shortest\_path.append(a)
    * 1. 日志实现：
97. filename = 'diary.txt'
98. with open(filename,'a') as file\_object:
99. file\_object.write(time.strftime("%Y-%m-%d %H:%M:%S : from\t",time.localtime()))
100. file\_object.write(a)
101. file\_object.write('\tto\t')
102. file\_object.write(b)
103. file\_object.write('\n')
104. #由于一次只能有一个参数，所以代码冗长
105. 创新点&优化（如果有）
     1. 第一版使用ASII码作为字典的键来查找节点，之后改为使用计数数字来查找键
     2. 第一版使用数组的线性查找来打印路径和查找路径，之后使用前驱节点和后继节点来简化查找节点的过程改线性查找时间为O（1）

# 实验结果及分析

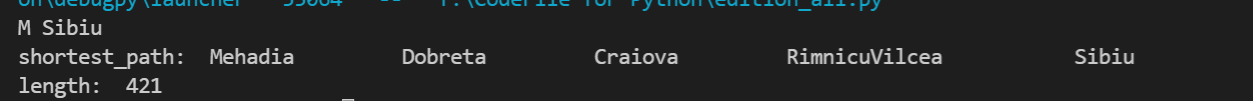
1. 实验结果展示示例（可图可表可文字，尽量可视化）
   1. 大小写混输：



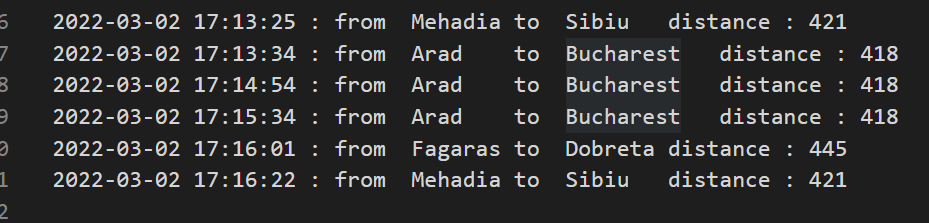
* 1. 首字母大小写



* 1. 首字母和全输入



* 1. 测试时的日志：



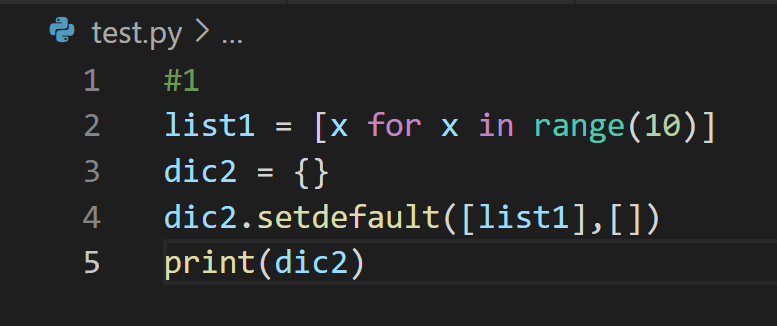
1. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）

**|-----------如有优化，请重复1，2，分析优化后的算法结果-----------------------|**

# 思考题

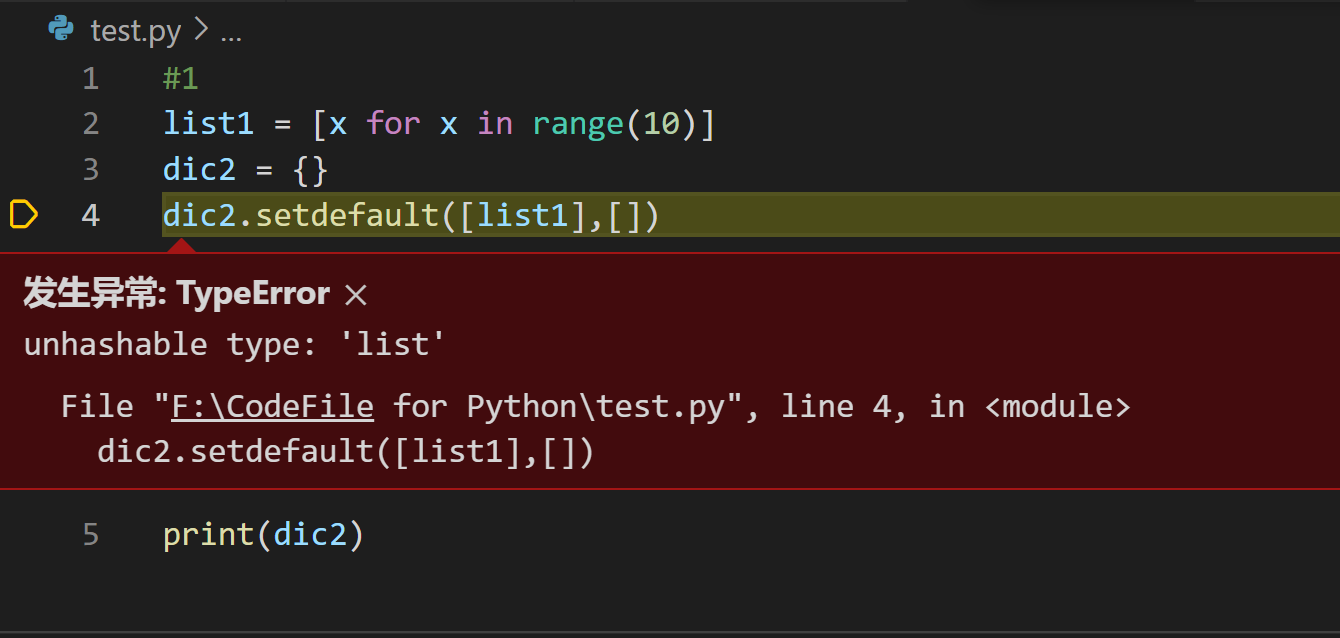
# **如果用列表作为字典的键，会发生什么现象？用元组呢？**

列表不可以作为字典的键，会报错。分析如下：



运行结果如下：

##1



所以 list 不可以作为字典的键的原因为“unhashable type”。

深入分析：[1]

字典中的值没有任何限制，可以是任意对象。但是字典的键是有类型限制的。

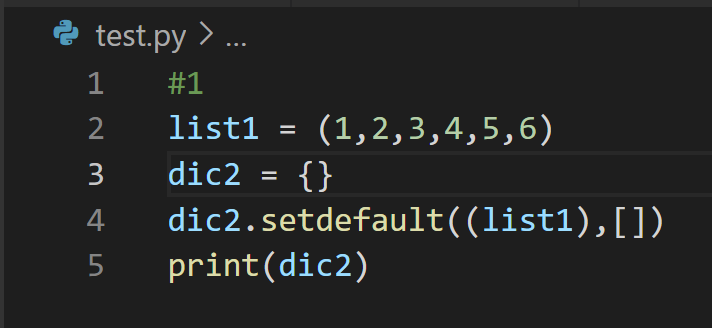
1. 不允许一个键对应多个值

一个键只能对应一个值，但是Python并不会因为一个键对应多个值而报错，而是取最近的一次赋值结果。

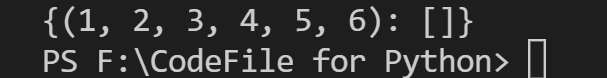
1. 键必须是可哈希的（即要求不可变）

解释器调用哈希函数，根据字典中的键的值来计算存储位置，如果键是可变的值，那么键一旦改变，哈希函数结果也会发生变化，得到的存储位置也会变化，导致字典不可用。

元组可以作为字典的键。分析如下：



运行结果：



分析：因为元组属于不可变数据类型，满足字典键的要求，于是可以执行。

1. **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。**

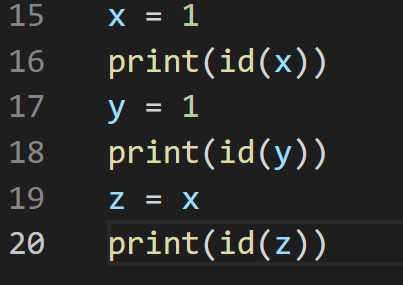
**•可变/不可变数据类型：变量值发生改变时，变量的内存地址不变/改变。**

**•提示：①你可能会用到id()函数。②Python的赋值运算符（=）是引用传递。**

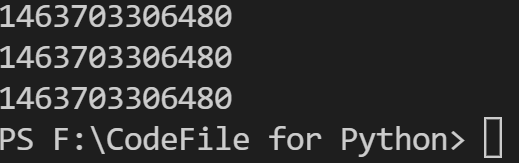
数字

整数 不可变

代码如下:

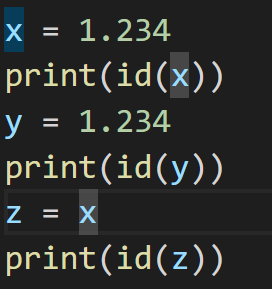


运行结果：

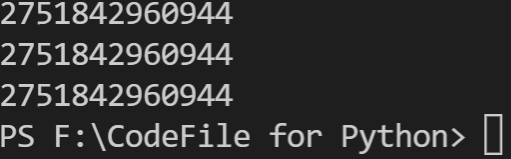


浮点数 不可变

代码如下

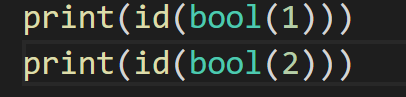


运行结果：

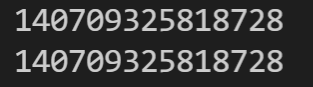


布尔值 不可变

代码如下：

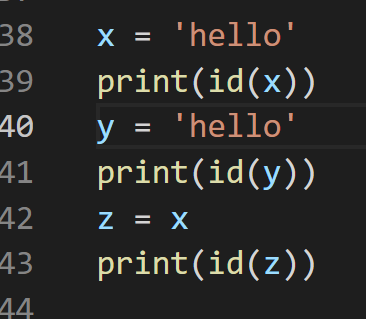


运行结果：

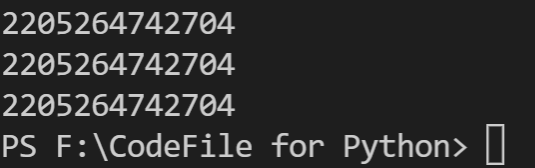


字符串 不可变

代码如下：

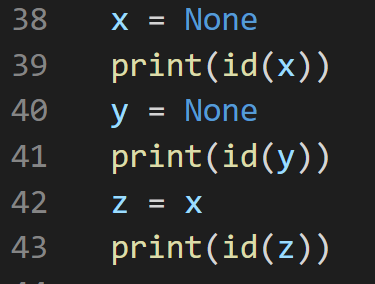


运行结果：

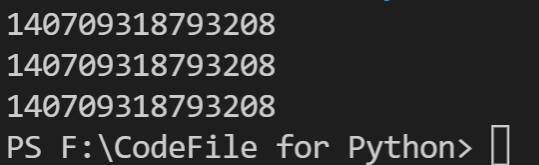


空值 不可变

代码如下：

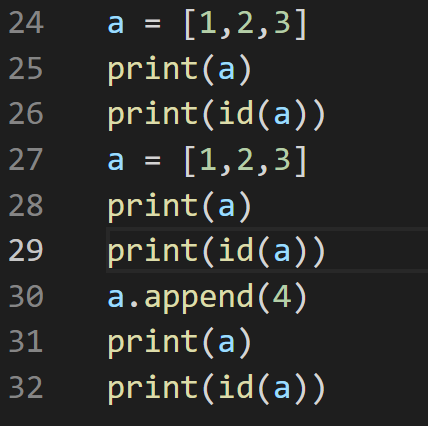


运行结果：

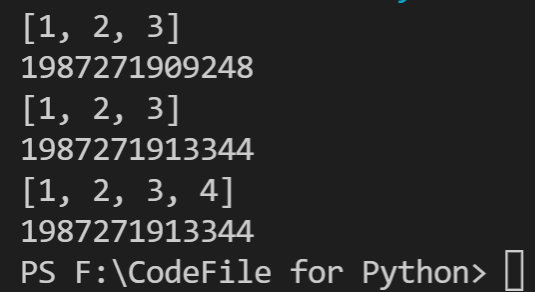


列表 可变

代码如下：

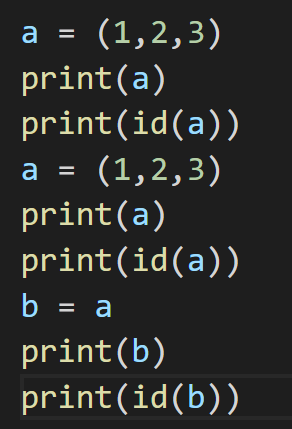


运行结果：

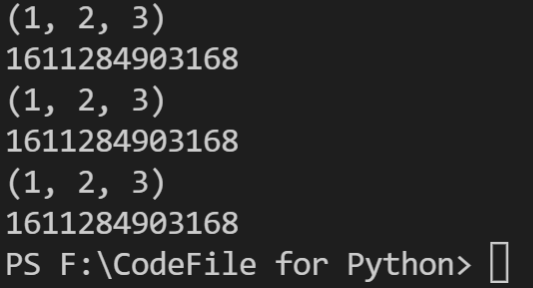


元组 不可变

代码如下：

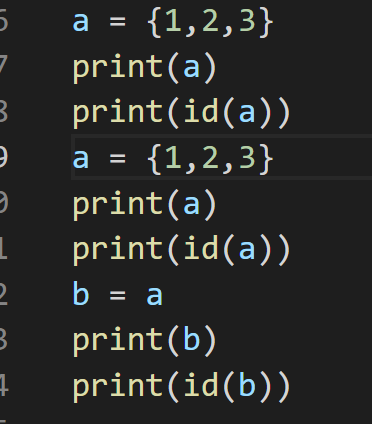


运行结果：

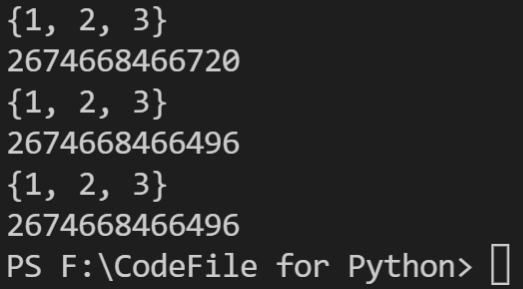


集合 可变

代码如下：

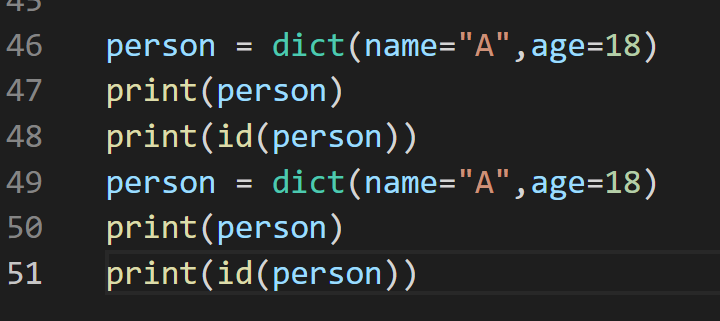


运行结果:

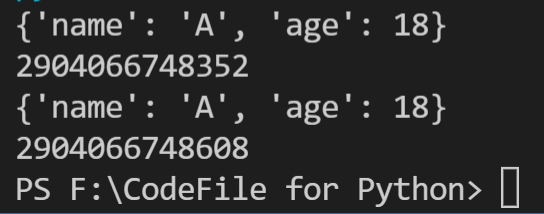


字典 可变

代码如下：



运行结果：



# 参考资料

**[1]**<https://wiki.python.org/moin/DictionaryKeys>

PS：可以自己设计报告模板，但是内容必须包括上述的几个部分，不需要写实验感想