

Assignment 5 – Dictionaries

Required Files

- `decipher_message_skeleton.py`
- `treasure_hunt_skeleton.py`
- `ancestor.py`
- And a few encrypted map files ended in `".txt"`

Dictionaries are a very useful tool in Python, being a very flexible data structure for a myriad of inputs. The purpose of this assignment is to test your understanding of Python dictionaries, and if you can apply them efficiently to certain problems. **Please be reminded that you may be penalized for inefficient code.**

Part 1: Treasure maps

Secret messages [20 marks]

On a quest for treasure, you chance upon a weird, seemingly encrypted message. What could it mean?

```
esbtr dgh abzqg! vhe ghz yzqtcjxx qx qt btgesjz cbxepj!
```

Thankfully, you always have your handy translational dictionary with you.

```
{'a': 'm', 'b': 'a', 'c': 'c', 'd': 'y', 'e': 't', 'f': 'v', 'g': 'o', 'h':  
'u', 'i': 'x', 'j': 'e', 'k': 'j', 'l': 'w', 'm': 'f', 'n': 'z', 'o': 'd', 'p':  
'l', 'q': 'i', 'r': 'k', 's': 'h', 't': 'n', 'u': 'g', 'v': 'b', 'w': 'q', 'x':  
's', 'y': 'p', 'z': 'r'}
```

Translating each letter in the encrypted message using the dictionary, you get the final message:

```
thank you mario! but our princess is in another castle!
```

Your task

Write a function, `decipher_message(msg, guide)`, that takes in two arguments to decipher the message “msg” with the dictionary “guide”. The message is a string and the guide is a dictionary. Your function should be able to translate any message with any dictionary and **return** the final deciphered message in a string.

Do note that your function should work for **any arbitrary dictionary** provided and **should not be hardcoded using the above example**. Characters in the encrypted message which are not found in the translation dictionary (keys) can be taken as-is without having to do any substitution. You are guaranteed that there will be no capital letters in the encrypted string.

Sample run:

```
>>> msgle = "esbtr dgh abzqg! vhe ghz yzqtcjxx qx qt btgesjz cbxepj!"  
  
>>> print(decipher_message(msgle, d1))  
thank you mario! but our princess is in another castle!
```

Part 2: Find the Treasure!

After a long and arduous journey, you finally obtain the treasure map. However, much to your disappointment, the treasure map is encrypted as well.

```
ZZ1DDZDDD1DZD1ZZZ1ZD111223222B;+;44411DZ11DZ
Z1D1ZZD1111ZZ111ZDD1ZD04CFF0B+BB0220Z1DDZ1ZD
Z1ZZ11DDZZZ1ZZD1DZDZ1Z400F22+;+200202Z111Z11
1ZD1ZZZ1DD11DZ1Z1Z11Z24F22222B;BB40F2111DDZD
1DDZZZZZ1DDZ1DD1D1120402444+BBB44C2DZD1ZDD
```

You turn to your handy translational dictionary once more.

```
d1 = {'D': 'W', '1': 'W', 'Z': 'W', 'C': 'T', '3': 'T', 'F': 'T', '0': '.',
      '2': '.', '4': '.', 'B': '^', '+': '^', ';': '^', 'Q': 'E', '7': 'E', '8': 'E',
      'X': 'M', 'P': 'M', '!': 'M', '(' : ':', ')' : ':', '9': ':', '*': ' ', '|': ' ',
      '#': ' '}
```

With the dictionary, you should be able to decipher the above encrypted map into something like this:

```
WWWWWWWWWWWWWWWWWWWWWW..T...^^^...WWWWWWWW
WWWWWWWWWWWWWWWWWWWWWW..TTT.^^^...WWWWWWWW
WWWWWWWWWWWWWWWWWWWWWW..T...^^^...WWWWWWWW
WWWWWWWWWWWWWWWWWWWWWW..T....^^^..T.WWWWWWW
WWWWWWWWWWWWWWWWWWWWWW.....^^^..T.WWWWWWW
```

You should have gotten the hint where the treasure is from the last question in Part 1!

Your task is to decrypt the map, and then find the treasure.

Task 1 (20 marks)






You may think, decrypting a map is the same job in the above question? Sorry, this time you need to read in a map from a **file** by yourself, and then write the new readable map into a file. Your task is to write a function

```
decode_map(mapfile,ddict,outfile)
```

Such that mapfile and outfile are two filenames. Your function will read in mapfile and use the ddict to decrypt, and then store the result into outfile. For example, after

```
decode_map('encoded_map.txt',d1,'decoded_map.txt')
```

It should read in a file 'encoded_map.txt' and write the result file into 'decoded_map.txt' in your directory.

	codeB	10/1/2019 1:16 PM	Microsoft Excel C...	1 KB
	decode_dict	10/1/2019 1:23 PM	Microsoft Excel C...	1 KB
	decoded_map	10/1/2019 1:24 PM	Text Document	2 KB
	decoded_map2	10/1/2019 1:24 PM	Text Document	2 KB
	decoded_map3	10/1/2019 1:24 PM	Text Document	2 KB

And the file 'decoded_map.txt' should contain the decrypted map like this ('answer map.txt'):

[illegible]

Task 2 (20 marks)

Write a function to read in the file you produced in the above task, and find the treasure! As you may know, the hint from the first question tells you that the treasure is in the middle of five trees that are planted in a cross shape like this:

```

.....WW
...T..W
W.TTT..I
W..T..W
      1.0.0.0

```

Write a function `find_treasure(mapfile)` that reads in the file 'mapfile' and returns the coordinates of the treasure in a tuple. For example:

```
>>> print(find_treasure('decoded_map.txt'))
(21, 25)
```

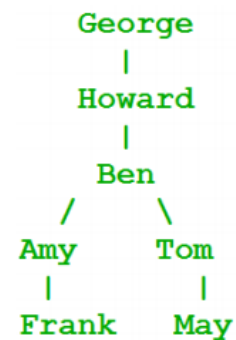
Again, note that your function should work for any arbitrary dictionary and encrypted map provided, and should not be hardcoded using the above example. Row and column indexes in the map also start from 0 – that means the first, ‘top-most’ and ‘left-most’ cell on the map will correspond to the 0th row and the 0th column. You can assume that the map will have only one location of treasure.

Part 3: Ancestry trees

All of us may come from different families, but we may share common ancestors! In this question, we'll explore the use of a dictionary to model ancestry trees.

Given some ancestry information, such as the following:

Ben is Amy's father,
Ben is also Tom's father,
Amy is Frank's mother,
Tom is May's father,
Howard is Ben's father,
George is Howard's father.



We can easily construct the 'graphical representation' of the ancestry tree (right), and we can also construct a dictionary to represent our ancestry tree:

```
parent = {'Amy': 'Ben', 'Tom': 'Ben', 'Frank': 'Amy', 'May': 'Tom', 'Ben':  
'Howard', 'Howard': 'George'}
```

Here, the keys of the dictionary represent the 'child', while the values of the dictionary represent the 'parent'. For example, 'Ben' is the ancestor of 'Amy', so our first entry is "'Amy': 'Ben'". So `parent['Amy']` will give the result 'Ben'.

To simplify this question, we assume that every name is unique, and every person has at most one parent. So, here is an example parent dictionary:

```
parent = {'Amy': 'Ben', 'May': 'Tom', 'Tom': 'Ben',  
          'Ben': 'Howard', 'Howard': 'George', 'Frank': 'Amy',  
          'Joe': 'Bill', 'Bill': 'Mary', 'Mary': 'Philip', 'Simon': 'Bill',  
          'Zoe': 'Mary'}
```

Are you my ancestor? [20 Marks]

Person A is an (direct or indirect) ancestor of Person B if Person B is considered to be one of the many descendants of Person A. In the example ancestry tree given above, Howard is an ancestor of Amy, but Amy is not an ancestor of Tom. And that person himself is NOT his own ancestor.

Write a function `is_ancestor(name1, name2, pdict)` to check if the person with name `name1` is an ancestor of the person with name `name2` in the parent dictionary `pdict`.

The first two arguments are the names of people (strings), while the third argument is the parent dictionary mentioned above. The function should return the boolean value `True` if the first person in the argument list is an ancestor of the second person, and `False` if the first person in the argument list is not an ancestor of the second person.

Sample run:

```
>>> is_ancestor('Howard', 'Amy', parent)
True
>>> is_ancestor('Amy', 'Tom', parent)
False
```

Are they related? [20 marks]

Two people are related if they share a common ancestor in the ancestry tree. In the example ancestry tree given above, Amy and Tom are related since they share a common ancestor, Ben, and Ben and Howard are related since they share a common ancestor, George. If we were to introduce a new person Luna, with no ties to anyone in the ancestry tree, Luna is considered to be not related to anyone in the ancestry tree.

Write a function `is_related(name1, name2, pdict)` to check if the person with `name1` is related to the person with `name2` in the parent dictionary `pdict`.

The first two arguments are the names of people (strings), while the third argument is the parent dictionary mentioned above. The function should return the boolean value `True` if the first person in the argument list is related to the second person, and `False` if the first person in the argument list is not related to the second person.

Sample run:

```
>>> is_related('Joe', 'Philip', parent)
True
>>> is_related('Amy', 'Philip', parent)
False
>>> parent['Ben'] = 'Philip'
>>> print(is_related('Amy', 'Philip', parent))
```