3. [SOLUTIONS] More Programs with Functions

Jukebox

You are to design a program which, given a musical library, accepts a track ID, displays all the info about the song and makes recommendations for the next track to be played. The musical library is stored in a text file as follows:

`id,name,artist,album,year,plays,similar`

With each field defined as follows:

- **`id`**: a 2-character string comprised of a letter then a number which uniquely identifies the track
- **`name`**: the name of the track, which should be stored as a string
- **`artist`**: the artist performing the track, which should be stored as a string
- **`album`**: the album on which the track appears, which should be stored as a string
- **`year`**: the year the album was released, which should be stored as an integer
- **`plays`**: the number of times this track has been played, which should be stored as an integer
- **`similar`**: a collection of **`id`**s for other tracks which are similar, separated by the | character - a track can have one or more similar tracks

The data for the jukebox is stored as text as below. You can assume that none of the fields contain commas.

```
t2,Sledgehammer,Peter Gabriel,So,1986,14,e1|e6
u3,Uptown Funk,Mark Ronson ft. Bruno Mars,Uptown Funk,2015,45,i9
y6,Leave Out All The Rest,Linkin Park,Minutes To Midnight,2007,16,m7|y6|m0
e1,Come and Get Your Love,Redbone,Wovoka,1973,12,t2|e6
m7,The Phoenix,Fall Out Boy,Save Rock And Roll,2013,53,u3|m0|u5
i9,Grounds For Divorce,Elbow,The Seldom Seen Kid,2008,9,z1|
u3,Red Right Hand,Nick Cave & The Bad Seeds,Let Love In,1994,42,i9|e1
m0,Mr Brightside,The Killers,Hot Fuss,2004,104,m0
z1,Graffiti on the train,Stereophonics,Graffiti On The Train,2013,23,i9
e6,The Chain,Fleetwood Mac,Rumours,1977,45,e1|m7
i9,Party Rock Anthem,LMFAO,Sorry For Party Rocking,2011,31,u3
u5,Welcome to the Black Parade,My Chemical Romance,2006,73,k0|m0|m7
y6,In Too Deep,Sum 41,All Killer No Filler,2001,61,t2|u3|e1|i9
n0,Bohemian Rhapsody,Queen,A Night At The Opera,1975,83,e1
k0,The Sound Of Silence,Disturbed,Immortalised,2015,32,y6|m0|z1
```

Complete the following tasks:

1. Design a data type to represent a single track. Consider how multiple tracks could be easily stored and accessed.

```
track = {
  "id": "id",
  "name": "name",
  "artist": "artist",
  "album": "album",
  "year": year,
  "plays": plays,
  "similar": ["id", "id", "id"]
}
```

Multiple tracks can be stored in a list

2. Write a function that accesses the above data as a multi-line text file (i.e. each line represents a single track) and stores it in the format you have detailed above.

```
def readFile(filename):
  tracks = []
  f = open(filename, "r")
  for rawTrack in f:
    rawTrackSplit = rawTrack.split(",")
    track = {
      "id": rawTrackSplit[0],
      "name": rawTrackSplit[1],
      "artist": rawTrackSplit[2],
      "album": rawTrackSplit[3],
      "year": int(rawTrackSplit[4]),
      "plays": int(rawTrackSplit[5]),
      "similar": rawTrackSplit[6].split("|")
    }
    tracks = tracks + [track]
  f.close()
  return tracks
```

3. Write a function called **`details( id )`** that, given a track id, prints the track name, artist, album and year. Here is a sample output using the first line of the dataset:

    `Sledgehammer by Peter Gabriel from So (1986)`

```
def details(id):
  for track in tracks:
    if track["id"] == id:
      outString = track["name"]
        + " by "
        + track["artist"]
```

```
    + " from "
    + track["album"]
    + " ("
    + str(track["year"])
    + ")"
  print(outString)
```

4. Write a function called **`similar( track )`** that, given a track, prints the details of all similar tracks associated with the original. Here is a sample output using the first line of the dataset:

```
Similar tracks:
Come and Get Your Love by Redbone from Wovoka (1973)
The Chain by Fleetwood Mac from Rumours (1977)
```

```
def similar(track):
  print("Similar tracks:")
  for sId in track["similar"]:
    details(sId)
```

5. Write a function called **`play( id )`** which, given a track id, prints "Now playing:" followed by the details of the track and the similar tracks. It should also update the dataset to increment the number of plays by 1.

```
def play(id):
  print("Now playing:")
  details(id)
  for track in tracks:
    if track["id"] == id:
      similar(track)
      track["plays"] = track["plays"] + 1
```

Note: for any of the above tasks, if you wish to define supplementary functions, please do so. For example, you may wish to write a function which will handle traversing your dataset if this is functionality which you may want to reuse.

FamilyTree

This question makes use of a dictionary format for representing a person that could be used to model family trees. The format has three entries, for the person's name, their parents, and their children. An example is as follows:

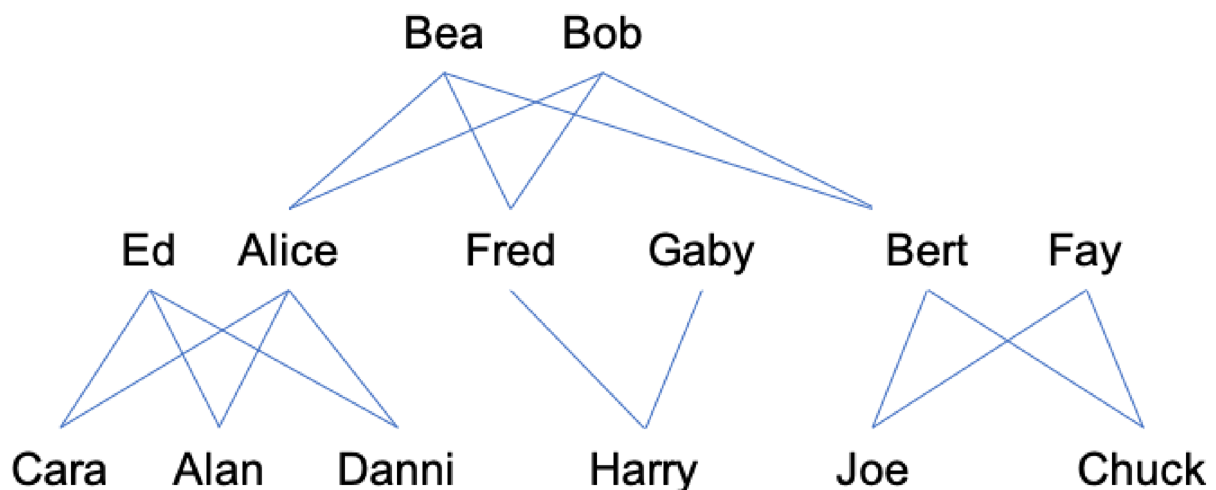
```
{ "name" : "Fred", "parents" : [], "children" : [] }
```

The parents and children entries can be empty lists, as above, in order to limit to extent of the family tree being recorded, in the case of parents, or because the person has no children, or because the person dictionary is under construction.

When parents or children are to be recorded, the relevant lists are extended. For example, if Fred has a child Harry, then this can be represented as follows, assuming the above dictionary is assigned to the variable `fred`:

```
harry = { "name" : "Harry", "parents" : [ fred ], "children" : [] }
fred[ "children" ].append( harry )
```

Using this method to link together dictionary instances, we could model a family tree such as the one below:



This would require 14 dictionaries, one for each of the family members in the tree, with all the appropriate links being recorded via updates to the `"children"` and `"parents"` entries in the relevant dictionaries.

Write functions according to the following specifications:

```
def printPerson( p ):
```
Takes a person dictionary as parameter and writes out the name of the person and the names of the parents and children. For example, if the function was called passing the dictionary for Alice *in the diagram, the following should be printed out:*

```
  Name: Alice; Parents: Bea, Bob; Children: Cara, Alan, Danni
```

```
def getNames( listPeople ):
    names = ""
    for p in listPeople:
        names = names + p[ "name" ] + ","
    return names[ :-1 ]

def printPerson( p ):
    print( "Name: " + p[ "name" ] +
           "; Parents: " + getNames( p[ "parents" ] ) +
           "; Children: " + getNames( p[ "children" ] ) )
```

```
def siblings( p ):
```
Takes a person dictionary as parameter and returns a list of the dictionary values for all siblings of the person that can be determined from the family tree. If none can be found, simply return an empty list. Step-siblings should be counted - but be sure to include any sibling only once. As an example, the siblings of Danni in the example above are Alan and Cara.

```
def siblings( p ):
    s = []
    for par in p[ "parents" ]:
        for child in par[ "children" ]:
            if child != p and child not in s:
                s.append( child )
    return s
```

```
def auntsUncles( p ):
```
Takes a person dictionary as parameter and returns a list of the dictionary values for all aunts/uncles of the person that can be determined from the family tree, defined as brothers or sisters of the person's mother or father. If none can be found, simply return an empty list. As an example, the aunts/uncles of Harry in the example above are Alice and Ben.

```
def auntsUncles( p ):
    aU = []
    for par in p[ "parents" ]:
        for gparent in par[ "parents" ]:
```

```
            unclesAuntsParent = gparent[ "children" ]
            for uAP in unclesAuntsParent:
                if uAP != par and uAP not in aU:
                    aU.append( uAP )
    return aU
```

```
def cousins( p ):
```

Takes a person dictionary as parameter and returns a list of the dictionary values for all cousins of the person that can be determined from the family tree. If none can be found, simply return an empty list. Remember that cousins can come via either parent. You should use the auntsUncles *function above in your solution. As an example, the cousins of Danni in the example above are Harry, Joe and Chuck.*

```
def cousins( p ):
    c = []
    for aU in auntsUncles( p ):
        for cousin in aU[ "children" ]:
            if cousin not in c:
                c.append( cousin )
    return c
```

```
def directAncestor( oldun, youngun ):
```

Takes two person dictionaries as parameters and returns True if the first parameter is a direct ancestor of the second parameter, False otherwise. For example, any of Bea, Bob, Ed, or Alice are direct ancestors of Cara.

```
# Non-recursive version
def directAncestor( oldun, youngun ):
    directAncestorFound = False
    ancestors = []
    ancestorsNotProcessed = [] + youngun[ "parents" ]

    while ancestorsNotProcessed != [] and not directAncestorFound:
        thisAncestor = ancestorsNotProcessed.pop( 0 )
        if thisAncestor is oldun:
            directAncestorFound = True
        else:
            ancestorsNotProcessed += thisAncestor[ "parents" ]

    return directAncestorFound
```

Finally, write a function `explore` that takes a person dictionary as parameter and enables the user to interactively explore the family tree. The function should print out the details of the person - name, and names of any children and parents recorded - and then offer the following options:

1. Move to a child - give the name of the child
2. Move to a parent - give the name of the parent
3. Print out cousins
4. Print out siblings
5. Exit

After each move, the details of the newly-found person should be printed out.
You decide how the interface should work.
Include error reporting, e.g. moving to a parent who doesn't exist.
Make use of the functions you have already written.

```
def getOption():
    print( "What would you like to do next?  Choose from:" )
    print( "Move to a child: type c then space then child name" )
    print( "Move to a parent: type p then space parent name" )
    print( "Print out cousins: type co" )
    print( "Print out siblings: type s" )
    print( "Exit: type e" )
    return( input( "Choice?: " ) )

def explore( p ):
    printPerson( p )
    choice = getOption()
    while choice != "e":
        if choice == "co":
            pCousins = cousins( p )
            names = getNames( pCousins )
            print( "Cousins: " + names )
        elif choice[ 0 ] == "c":
            childName = choice[ 2: ]
            child = {}
            for c in p[ "children" ]:
                if c[ "name" ] == childName:
                    child = c
            if child == {}:
                print( "No child of that name." )
            else:
                p = child
                printPerson( p )
```

```
elif choice[ 0 ] == "p":
    parentName = choice[ 2: ]
    parent = {}
    for par in p[ "parents" ]:
        if par[ "name" ] == parentName:
            parent = par
    if parent == {}:
        print( "No parent of that name." )
    else:
        p = parent
        printPerson( p )
elif choice[ 0 ] == "s":
    pSiblings = siblings( p )
    names = getNames( pSiblings )
    print( "Siblings: " + names )
else:
    print( "Input not understood, try again..." )
choice = getOption()
```