2. [SOLUTIONS] Programs with functions

MagazineSubscribers

A magazine subscription service has a single subscriber with a first name, surname, home address and email address. Every month they will send out a magazine to the subscriber's address, with their name attached, and will send them an email to tell them that their magazine is on the way.

1. How could the data be stored to make this process easy?

Ans: in a dictionary:

```
subscriber = {
  "first_name": "first name",
  "surname": "surname",
  "home_address": "home address",
  "email_address": "email address"
}
```

Nb: a student who is adamant that a list would serve should be shown the difference between code that says `subscriber[3]` and `subscriber["email"]` in order to get the email field - the latter is much clearer!

2. The subscription service takes 10 more subscribers. How could you store this data in such a way that the above process is still manageable?

Ans: as a list of subscribers which can be iterated through

3. The subscription service grows to over 100 subscribers. Does the data structure above need to be changed significantly to accommodate this?

Ans: no, iterating over a list of 100 elements should be done in just the same way as 10 elements.

4. The subscription service has too many subscribers who input their postcode in the incorrect format or get their address lines mixed up. They wish to divide the home address field into address line 1, address line 2 and postcode so that they can be individually accessed. How would you modify the original data structure?

Ans: add a nested dictionary for addresses:

```
subscriber = {
  "first_name": "first name",
  "surname": "surname",
  "home_address": {
    "address_line_1": "address line 1",
    "address_line_2": "address line 2",
```

```
    "postcode": "postcode"
  },
  "email_address": "email_address"
}
```

Assume that the following function has already been written:

> **`emailSubscriber( first_name, surname, email_address )`**
> *This function constructs an email body using the deliveree's first and surname and sends the resulting email to their email address*

5. Given the data structures you have designed, write a program to - for every subscriber in your data structure - verify their postcode and send them an email. To verify a postcode, check that it ranges from 6 to 8 characters. If this check fails, do not email the subscriber. Use the above function to send an email.

Ans:

```
subscribers = [{...}, {...}, ...] # the subscriber list

for subscriber in subscribers:
  postcode = subscriber["home_address"]["postcode"]
  if len(postcode) >= 6 and len(postcode) <= 8:
    email(subscriber["first_name"], subscriber["surname"], subscriber["email_address"])
```

IntersectingLines

You are to write a program to determine whether a collection of lines overlap at all, specified by the (x,y) coordinates of their start and end points. For any pair of lines that overlaps, details of the two lines should be written out.

Assume the following:

- You have access to a function **`intersect( line1, line2 )`**, that takes two lines as parameters and returns True if they intersect and False otherwise.
- The lines you are to work with are already available in a variable called **`lines`**.

1. Design a data structure to hold a collection of lines.

Answer: A list of dictionaries would be appropriate. Each dictionary holds the information about a single line - the two coordinates of the start and end of the line, held as four integers. An example of a single line, with end points (3,4) and (8,9), is as follows: ***`{ "x1" : 3, "y1" : 4, "x2" : 8, "y2" : 9 }`***

2. Write a function to take four numbers, representing the coordinates of the two end points of a line, and return a instance of the data structure you have defined in (1) to represent a single line.

Answer:

```
def makeLine( x1, y1, x2, y2 ):
    return { "x1" : x1, "y1" : y1, "x2" : x2, "y2" : y2 }
```

3. Assuming **`lines`** is an instance of the data structure you designed in (1), write a short program to output the details of every pair of lines that overlap.

Answer:

```
# Check every line against every other line to see if they overlap - write out
details if they do
for line1index in range( len( lines ) - 1 ):
    for line2 in lines[ line1index + 1: ]:
        if intersect( lines[ line1index ], line2 ):
            print( "Intersect: (", lines[line1index]["x1"], ",",
                                   lines[line1index]["y1"], ") - (",
                                   lines[line1index]["y1"], ",",
                                   lines[line1index]["y2"], ") and (",
                                   line2["x1"], ",", line2["y1"], ") - (",
                                   line2["x2"], ",", line2["y2"], ")", sep = "" )
```

4. Convert the code you wrote in (2) into a function that takes as parameter a collection of lines in the format you designed in (1) and returns a collection of every pair of lines that overlap. What data structure will you use to hold the collection of pairs?

Answer:

```
def overlappingLines( lines ):
    overlaps = []
    for line1index in range( len( lines ) - 1 ):
        for line2 in lines[ line1index + 1: ]:
            if intersect( lines[ line1index ], line2 ):
                overlaps.append( (lines[ line1index ], line2) )
    return overlaps
```

HighScoreTable

In a game being developed, a high score table is required that will hold only the top five scores seen to date. Each entry holds a name and a score - a whole number. A player can only appear in the table once. The scores in the table should be retained in order, highest to lowest.

1. Design a data structure to hold the high score table

Answer: A list of dictionaries, where each dictionary represents a single high score by having two entries, one for the name of the player (a string) and one for his/her score (an integer). The list can never have more than five entries, but can have fewer. So - a high score table with only one entry would look like: `[ { “name” : “Quintin”, “score” : 78 } ]`

Assume the following functions are already available:

`isANewHighScore( scores, score )`
Takes a high score table in your format from (1) and a score. Returns True if the score should be inserted into the table, False otherwise.

`addEntry( scores, name, score )`
Takes a high score table in your format from (1), a name as a string, and a score as an integer. Updates the table with the new entry, according to the initial specification above. Assume a check has already been made to determine that this is a new high score to be inserted into the table.

`containsName( scores, name )`
Takes a high score table in your format and a name as a string. Returns -1 if the name does not appear in the table, otherwise the index into the high score table contains an entry for this name.

`printTable( scores )`
Takes a high score table in your format, and prints it out.

`playGame()`
Plays a single round of the game, returning the score gained as an integer.

2. Write a program that makes use of these functions **(assume they ARE written already)** and enables a user to play the game repeatedly - consider the following
    a. Ask for a y/n response on whether the user wants to continue playing after each game is completed.
    b. Update the high score table after each game is completed, if the score is good enough (if it is, you'll need to ask the current user for their name) and then print out the table.
    c. Write out the winner at the end.

Answer:

```
# Main program section
# --------------------

highScores = []
continue = "y"

while continue == "y":
    scoreThisGame = playGame()
    if isANewHighScore( highScores, scoreThisGame ):
        playerName = input( "Well done - a new high score - what is your
name? " )
```

```
        addEntry( highScores, playerName, scoreThisGame )
        printTable( highScores )
    continue = input( "Play again?  (y/n) : " )

print( "The winner is", highScores[ 0 ][ "name" ],
       "with a score of", highScores[ 0 ][ "score" ] )
```

3. Write the **printTable** function.

Answer:

```
def printTable( scores ):
    print( "High score table" )
    print( "----------------" )
    for entry in scores:
        print( entry[ "name" ], ":", entry[ "score" ] )
```

4. Write the **isANewHighScore** function.

Answer:

```
def isANewHighScore( scores, score ):
    return len( scores ) < maxSize or scores[ -1 ] < score
```

5. Write the **containsName** function.

Answer:

```
def containsName( scores, name ):
    pos = -1
    for i in range( len( scores ) ):
        if scores[ i ] == name:
            pos = i
    return pos
```

6. Write the **addEntry** function.

Answer:

```
def addEntry( scores, name, score ):
    newEntry = { "name" : name, "score" : score }

    nameExistsPos = containsName( scores, name )

    if nameExistsPos == -1:
        if len( scores ) == maxSize:
            del( scores[ -1 ] )
    else:
        del( scores[ namesExistsPos ] )

    pos = 0
    while pos < len( scores ) and scores[ pos ][ "score" ] > score:
```

```
        pos = pos + 1

    if pos == len( scores ):
        scores.append( newEntry )
    else:
        scores.insert( pos, newEntry )
```

AvailabilityChecker

In this example we will develop an application that performs functions on a calendar. Consider the following data structure:

```
my_day = [
  {
    "start": 9,
    "end": 11,
    "name": "Teaching",
    "private": False,
  },
  {
    "start": 12,
    "end": 14,
    "name": "Doctor's Appointment",
    "private": True,
  },
  {
    "start": 14,
    "end": 16,
    "name": "Supervisor Meeting",
    "private": False,
  }
]
```

1. Write a function called **`simpleCheck( hour )`** that, given an hour in the 24hr clock format, will check this hour against the calendar and return whether or not I am busy. If I have an appointment, return the dictionary for that appointment. If not, return False.

Ans:

```
def simple_check(hour):
  for app in my_day:
    if hour >= app["start"] and hour < app["end"]:
      return app
  return False
```

2. Write a function called **infoCheck(appointment)** that, given an appointment (as a dictionary), returns the name of the appointment, but only if it is not private. If it is a private appointment, a string should be returned indicating as such. If there is no appointment, a string indicating that I am free should be returned.

Ans:

```
def info_check(app):
  if app["private"]:
    return "Private appointment"
  else:
    return app["name"]
```

3. Write a function called **workCheck(hour)** that indicates whether or not the hour specified is during work time. Assume that I work from 8 to 18 - any time queried outwith these hours should return False, any time within should return True.

Ans:

```
def work_check(hour):
  if hour >= 8 and hour < 18:
    return True
  else:
    return False
```

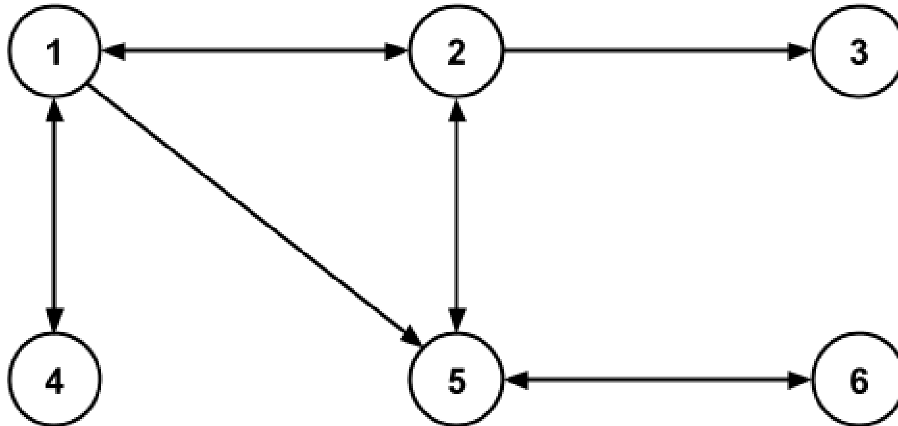
4. Construct the full application: this should have a command line interface which asks the user to input an hour. The application should then return something informative about my working schedule (something like "he's free" or "he has a private appointment" or "it's outside his working hours", etc) and keep accepting hours until a blank line is passed. Try and use the functions you have written to minimise the number of checks that you have to do.

Ans:

```
inp = int(input("Enter an hour to check: "))
while inp != "":
  if work_check(inp):
    app = simple_check(inp)
    if app != False:
      print(info_check(app))
    else:
      print("I am free")
  else:
    print("It's outside work hours")
  inp = int(input("Enter an hour to check: "))
```

Networks

In a network, nodes represent machines which are connected to each other. Each node knows which nodes it is connected to, however these connections may not be fully accurate: sometimes one node will record that it can connect to another, but the other node will not be able to respond. Below is a diagrammatic example of a network:



Some networks have a central system used to monitor and record the connections which exist in a network. This exercise will involve writing a program which mimics the central system.

1. Decide how you would represent the above network in a data structure, recording which nodes each node is connected to.

Ans:

```
nodes = {
  1: [2, 4, 5],
  2: [1, 5, 3],
  3: [],
  4: [1],
  5: [2, 6],
  6: [5]
}
```

2. The central system should be capable of listing all the nodes which a given node can connect to. Write a function **`cons( node )`** which, given a node number, returns all the nodes it can connect to as a list of numbers.

Ans:

```
def cons(node):
  return nodes[node]
```

3. As previously mentioned, some nodes assume they can connect to another node when that node can't connect to them (see the connection between nodes 2 and 3). Write a function **`comm( node1, node2 )`** which takes two node numbers and checks if each one is capable of connecting to the other. If they are it returns True, otherwise it returns False.

Ans:

```
def comm(node1, node2):
  nodes_1 = cons(node1)
  nodes_2 = cons(node2)
  if node1 in nodes_2 and node2 in nodes_1:
    return True
  else:
    return False
```

4. Write a function **`link( node1, node2 )`** which adds a link between two nodes if no link already exists. That is, each node should appear in the other's connection list. Assume that the nodes provided will already be in your data structure.

Ans:

```
def link(node1, node_id_2):
  if not comm(node1, node2):
    nodes_1 = cons(node1)
    nodes_2 = cons(node2)
    if node1 not in nodes_2:
      nodes[node2] += [node1]
    if node2 not in nodes_1:
      nodes[node1] += [node2]
```