Name:	Reece Benson
Student No.:	16021424
Module:	Design and Analysis of Data Structures and Algorithms
Assignment No.:	Assignment 1
Due Date:	Thursday 30th November, 2017

Table of Contents

Brief

1. Identify and justify what type of data structures you will use for storing the data
2. Design a solution for ranking players according to the points they have earned (pseudocode)
3. Implement the solution for ranking players according to points earned (Python Code)
4. Explain the algorithm implemented and justify its choice
5. Design a further solution that ranks the payers based on prize money earned (pseudocode)
6. Implement the additional solution in task 5 using Python.
7. Discuss and justify your choice of entering match results.

Identify and Justify Data Structures

The data structures I will be using within my implementation will be the use of dictionaries, lists and classes. The lists will be used within the dictionaries, and the list values will contain Object references to already-initialised Classes.

Each Class is initialised during run-time of the application, and the classes I will implement are the following, followed with a description of what the class does and holds:

- *Tennis Related Classes:*
    - Season
        - Holds data about:
            - Name
            - Players (with Genders)
            - Tournaments within this Season
            - Settings
        - Handles:
            - Adding tournaments
            - Adding genders
            - Adding players
            - Overall Leader boards
    - Tournament
        - Holds data about:
            - Name
            - Season (parent) it is linked to
            - Rounds
            - Prize Money
            - Difficulty
            - File Saving State
        - Handles:
            - Adding rounds
            - Calls to generate, edit, clear, delete and input rounds
            - Defining prize money
            - Emulation of Tournament
            - Emulation of Specific Round
            - Viewing Leader board
            - Displaying Data about Tournament (Difficulty and Prize Money)

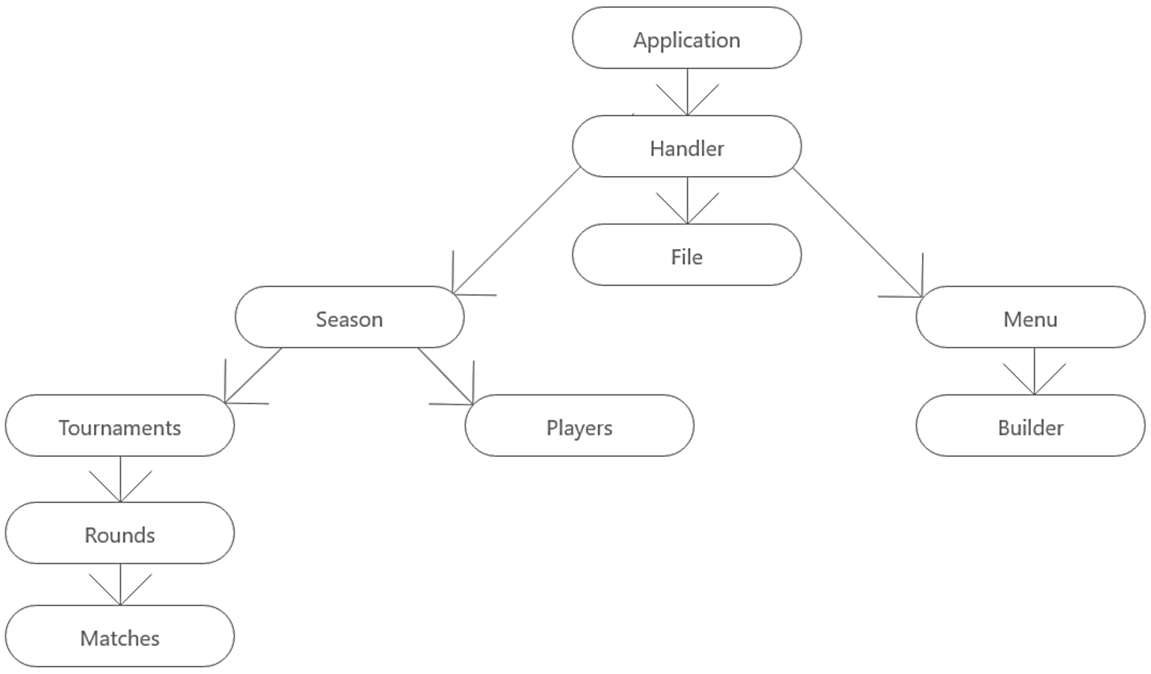
- Round
    - Holds data about:
        - Name and ID (identifier)
        - Gender
        - Tournament (parent) it is linked to
        - Previous Round (Object reference)
        - Players in this round
        - Winners in this round
        - Matches in this round
        - Matches score cap
    - Handles:
        - Adding players & winners
        - Validating round data
        - Retrieving players & winners
- Match
    - Holds data about:
        - Player A and Player B (Object reference & score)
        - Winner of the Match
        - Round (parent) it is linked to
    - Handles:
        - Validating match data
        - Retrieving Player A/B
        - Retrieving Match Winner
- Player
    - Holds data about:
        - Name
        - Gender
        - Wins
        - Score
    - Handles:
        - Increasing wins
        - Setting Score

- *Handling Classes:*
    - Handler
        - Holds data about:
            - Seasons
            - Ranking Points
        - Handles:
            - Loading of:
                - Players
                - Seasons
                - Tournaments
                - Rounds
                    - Generation
                    - Empty Project
                    - Previous Round (last state)
                - Ranking Points
            - Setting the Round Loading mode
            - Saving Rounds
            - Calculating the count of rounds
    - File
        - Holds data about:
            - *n/a*
        - Handles file saving of:
            - Updating Tournament Rounds
            - Updating Settings
            - Updating/Retrieval of Global Settings
            - Get Seasons as Raw JSON
            - Update File Saving States
    - Builder
        - Holds data about:
            - Menu (as a dictionary)
            - Tree (current menu)
            - Current Menu (string)
            - Title (title of menu in terminal)
            - Flags
                - Force Close
                - Force Reload
        - Handles:
            - Open/Reload/Close Menu
            - Add (Sub)Menu/Info/Function
            - Validation of Menu/Function
            - Set Menu
            - Input Monitoring
            - Show Menu (builds and displays menu)

- Menu
    - Holds data about:
        - *n/a*
    - Handles:
        - The initialisation of the Menu
        - Debug Info (when debug flag is True)
        - Developer Information

I have chosen to use this as my structure as I believe it is a clean and easily maintainable implementation. Due to the use of numerous classes, it allows me to reference the Object without instantiating extra memory space as it is already allocated. Furthermore, this also allowed me to retrieve data from numerous classes very easily by being able to filter through attributes within each class. This worked particularly well with the Quick Sort algorithm I implemented as I can sort through the attribute specified within the algorithm.

Being able to visualise the structure of my implementation in a tree-like manner makes it easier for development. For example:



Design a Solution for Ranking Players (pseudo code)

```
function load_ranking_points():
    WHILST file_open('./data/rankingPoints.json') REFER AS tData:
        VAR data = json.load(tData)

        if(ranking_points IS None):
            ranking_points = (NEW DICTIONARY)

        foreach VAR pts in data
              foreach VAR rank in data[pts]
                    ranking_points.append( AS_INT(pts) )


function unique_ranking_points():
    if(ranking_points_unique IS None):
        VAR ranking_points_unique = (NEW LIST)
        foreach VAR element in ranking_points
              if COUNT OF element IN ranking_points_unique IS 0
                    ranking_points_unique.append( element )

        ranking_points_unique.remove_last_element_of_array()

        ranking_points_unique.reverse_array()
    return ranking_points_unique


function add_players((array of) plyrs):
    foreach player in plyrs:
        player.set_score(TOURNAMENT_NAME, ROUND_ID, 0)

        if(player is in WINNERS_LIST):
            VAR urp = unique_ranking_points()
            player.set_score(TOURNAMENT_NAME, ROUND_ID, (urp[ROUND_ID]))

        PLAYERS_LIST.append( player )
```

```
function view_leaderboard(gender DEFAULT = None, rnd_name DEFAULT = None):
    VAR players = Quick_Sort_Algorithm(PLAYER_LIST[gender], rnd_name)
    VAR place = 1

    for i in reversed(range(length(players))):
        print("#PLACE: NAME – SCORE – SCORE MULTIPLIER")
        INCREMENT place BY 1
```

Implement the Solution for Ranking Players (python code)

```
from classes.QuickSort import quick_sort as sort

---

def load_ranking_points(self):
    with open('./data/rankingPoints.json') as tData:
        data = json.load(tData)

        if(self.ranking_points == None):
            self.ranking_points = { }

        self.ranking_points  = [ int(pts) for pts in data for rank in 
data[pts] ]

        self.ranking_points += [ 0 ] * ( self.player_count - 
len(self.ranking_points))


def unique_ranking_points(self):
    if(self.ranking_points_unique == None):
        self.ranking_points_unique = []
        [ self.ranking_points_unique.append(i) for i in self.ranking_points if 
not self.ranking_points_unique.count(i) ]

        self.ranking_points_unique.pop()

        self.ranking_points_unique.reverse()

    return self.ranking_points_unique


def add_players(self, *plyrs):
    for p in plyrs:
        p.score_set(self.parent().name(), self.name(), 0)

        if(p in self.winners()):
            urp = self._app.handler.unique_ranking_points()
            p.score_set(self.parent().name(), self.name(), (urp[(self.id() - 
1) if (len(urp)) > self.id() else (len(urp) - 1)]))

        self._players.append(p)
```

```
def view_leaderboard(self, gdr = None, rnd_name = None):
    rnd = self.round(gdr, rnd_name)
    call("cls")

    print("View Leaderboard for '{0}', Round {1}:".format(self.name(),
rnd.id()))
    print("——————————————————————————————————————————————————————————")

    srt = sort(self.season().players()[gdr], self.name())
    place = 1
    for i in reversed(range(len(srt))):
        print("#{0}: {1} – {2} – {3}".format(f"{place:02}", srt[i].name(),
"{0:03d} score".format(srt[i].score(self.name(), rnd_name) if
(srt[i].score(self.name(), rnd_name) != 0) else
srt[i].highest_score(self.name(), False)), "{0:03d} diff
score".format(int(srt[i].score(self.name(), rnd_name) * self.difficulty()) if
(srt[i].score(self.name(), rnd_name) != 0) else
int(srt[i].highest_score(self.name(), False) * self.difficulty()))))
        place += 1
```

Explain the algorithm implemented and justify its choice

```
def quick_sort(arr, attr = None):
    if(len(arr) <= 1): return arr
    else:
        piv = arr[0]
        gt  = [ e for e in arr[1:] if e.highest_score(attr, False) >
piv.highest_score(attr, False) ]
        lt  = [ e for e in arr[1:] if e.highest_score(attr, False) <=
piv.highest_score(attr, False) ]
        return quick_sort(lt, attr) + [piv] + quick_sort(gt, attr)
```

The Algorithm I had implemented is the Quick Sort Algorithm. This algorithm has the worst-case performance of O(n^2), the best-case performance of O(n log n). The Quick Sort implementation divides the array into smaller parts and then branches off to sort the individual parts. I used this algorithm due to its ability to store in-space or in-place thus meaning this method requires no extra space allocated to perform its sort. This Algorithm uses a method that is commonly referred to as "Divide and Conquer", dividing consists of taking a pivot and then the values below the pivot are stored to the left, and the values above are stored to the right. Once the sort is complete, the array is sorted from lowest values to the left, highest values to the right. In the case of my implementation, I sort by an attribute within the element and move the element from there, so the player (element) with the highest score is at the end of the array, and the lowest score is at the start of the array.

Design a Solution for Ranking on Prize Money (pseudo code)

```
function view_prize_money(gender DEFAULT = None):
    # Call our Unique Prize Money (as previously defined) to update the values
    CALL FUNCTION unique_prize_money()
    SET VAR FROM ABOVE FUNCTION AS prize_money_unique

    VAR players = Quick_Sort_Algorithm(PLAYER_LIST[gender], TOURNAMENT_NAME)
    VAR place = 1
    for i in reversed(range(length(players))):
        VAR player_money = prize_money_unique[players[i].wins()]

        print("#PLACE: NAME – £MONEY")
        INCREMENT place BY 1
```

Implement the Solution for Ranking on Prize Money (python code)

```
def view_prize_money(self, gdr = None):
    call("cls")
    self.unique_prize_money()

    print("View Prize Money for '{0}':".format(self.name()))
    print("———————————————————————————————————————————————————————————")

    srt = sort(self.season().players()[gdr], self.name())
    place = 1
    for i in reversed(range(len(srt))):
        p_prizemoney = self._prize_money_unique[srt[i].wins(self.name())]
        print("#{0}: {1} – £{2:,}".format(f"{place:02}", srt[i].name(),
p_prizemoney))
        place += 1
```

Discuss and justify your choice of entering match results

Within my assignment implementation, I have allowed a fully customisable system to entering round data for each tournament. When you start the application, you are greeted by a menu that allows you to pick the way you would like to load data: empty project, generate data, load previous data. These three options all have different functionality.

Basis of all loading methods

- You have the ability to generate, edit and clear any round, at any time.
- You can emulate the tournament, at any time.
- You can view a specific round, at any time.
- You can view the current leader board for any specific round, at any time.
- You can only view the prize money results on the final round.
- You can view the difficulty and prize money pool for each tournament, at any time.

Empty Project

- Completely erases *all* round data for each tournament from the \`seasons.json\` file. This will override the saving flag defined within the tournament. If you are picking this route, you will be limited on what data you can display.

- Using this route, you can select individual rounds and have the option to generate each round by round. You can still emulate the tournament, but only the rounds that have data and are available to emulate.

- If you were to try to view the overall leader board with no data present, you would not be able to select any options at all. You can only view the overall leader board of tournaments that have been completely played through (round 1 to the final round).

Generate Data

- This method overwrites all previous round data for each tournament from the \`seasons.json\` file and replaces it with newly generated round data. This includes randomising the players that are matched against one another, who wins and the scores of the matches. This is generated up to the round specified on the terminal window.

- Using this route, you can still do everything you could do with an Empty Project, however you will only be able to generate or manually input data for rounds that do not exist. You can still clear rounds using the Edit menu, and then go back to generating or manually inputting the rounds too.

Load from previous

- This does exactly what it says, all of the tournaments, matches, scores and winners are retrieved from the \`seasons.json\` file and loaded back into the session you were previously in. You can do all of the features listed above.

The reason why I picked to use these methods of inputting data by generation or manual input is so that the user has complete flexibility about how they would like to manage the round data. I provide two main sources of entering match results, and this can either be through:

- Generating Data up to Round *x*
- Manually inputting data, Round by Round

Generating Data is available on the load menu whereas Manual Input is not. To be able to access the manual inputting side of my implementation, you can generate up to a specific round that is not the final round. Through the menu, cycle through to the "Select Round" within a tournament and any round marked with "→" at the end of it is not yet initialised and can be generated or you can begin manual input for each match there.

If a round already has data but you want to edit the match data within, you can use the "Edit Round" menu within a tournament and select a round. Through this, you can edit the matches or clear the matches completely. When editing matches, if you change a match's score and the winner is no longer the same, the rounds after the round you selected (if any) will be cleared. This same error check happens when clearing matches – i.e. if you clear Round 2, and Round 3, 4 and 5 exist, rounds 2 to 5 will be cleared also to avoid corrupt data.

Additional Pseudo Code for Main Features

Monitoring User Input

```
def monitor_input():
    if(FORCE_CLOSE FLAG):
        CLOSE MENU

    TRY:
        VAR resp = Get User Input()

        if(resp IS "exit" or resp IS "quit" or resp IS "x"):
            raise KeyboardInterrupt (HANDLES CTRL+C, END PROGRAM)
        elif(resp IS EMPTY):
            REPEAT FUNCTION

        try:
            if(resp IS "b"):
                MENU GO BACK
            VAR req = TO_INT( resp )
            VAR req_menu = Find_Menu_Function( req )
            if(TYPE OF 'req_menu’ IS dict):
                if(req_menu.menu):
                    set_current_menu(req_menu.reference)
                    add_menu_tree(req_menu.reference)
                    show_current_menu()
                else:
                    if(Builder.is_func(req_menu['ref'])):
                        < CLEAR TERMINAL >

                        # Execute
                        VAR retStr = execute_function(req_menu.reference)

                        # Hold user (to display output from function)
                        if(retStr IS NOT "SKIP"):
                            input("\n>>> Press <Return> to continue...")

                        if (NOT FORCE_RELOAD):
                            < SHOW MENU AGAIN >
                        else:
                            < REBUILD MENU >
                    else:
                        if(ITEM DOES NOT EXIST):
                            PRINT < ERROR >
                            < SHOW MENU AGAIN >
            else:
                VAR current_menu = FIND_ITEM( the_current_menu )
                if(req IS (length_of(current_menu) + 1) and the_current_menu
IS NOT “main”):
                    < GO BACK ON MENU >
                else:
                    < SHOW MENU AGAIN >
        # Exceptions (handled from outer-scope try/except)
        exception KeyboardInterrupt, ValueError, Exception:
              < HANDLE EXCEPTION >
```

Loading Files

```
function load_seasons():
    whilst file_is_open('./data/seasons.json') referenced as tData:
        VAR data = json.load(tData)

        foreach season in the variable data:
            if(season NOT MADE):
                CALL add_season_to_list( season )

                CALL load_players( season )
                CALL load_ranking_points()
                CALL load_rounds( season )
                CALL load_tournaments( season )

                if(season HAS round_load_mode):
                    if( round_load_mode IS FUNCTION ):
                        CALL round_load_mode( )
                    else:
                        CALL load_rounds( season )
```

Manual Input

```
VAR available_players = < COPY OF players ARRAY >

while(available_players IS NOT EMPTY):
    # Match Specific Variables
    VAR winnerCount = 0

    VAR plyr_one = input("Please enter Player A: ")
    if(plyr_one does not exist): BREAK

    VAR plyr_one_score = input("Please enter the score for Player A: ")
    if not (TO_INT(plyr_one_score) BELOW OR EQUAL TO ROUND_CAP): BREAK

    VAR plyr_two = input("Please enter Player B: ")
    if(plyr_two does not exist): BREAK

    VAR plyr_two_score = input("Please enter the score for Player B: ")
    if not (TO_INT(plyr_two_score) BELOW OR EQUAL TO ROUND_CAP): BREAK

    REMOVE plyr_one AND plyr_two FROM available_players
```

Generating Rounds

```
foreach VAR gender in players:
    foreach VAR r in range(0, MAXIMUM_ROUND_CAP):
        VAR r_name = < GET ROUND NAME >

        # Default Values
        VAR round_cap = < GET SPECIFIC GENDER ROUND CAP >

        # Data to Check
        VAR prev_r = < GET PREVIOUS ROUND >

        # Check if we have a round to take data from
        if (prev_r DOES NOT EXIST):
            VAR match_cap = (length_of(ALL_PLAYERS) DIVIDED BY 2)
        else:
            VAR match_cap = (length_of(prev_r.winners()) DIVIDED BY 2)

        VAR rand_players = [ ]
        RANDOMISE ORDER OF rand_players ARRAY

        foreach PLAYER in range(length_of(rand_players) DIVIDED BY 2):
            VAR p_one = rnd_players[PLAYER * 2]
            VAR p_two = rnd_players[(PLAYER * 2) + 1]

            # Generate some scores
            VAR p_one_score = random.randint(0, round_cap - 1)
            VAR p_two_score = random.randint(0, round_cap - 1)

            # Make a random player the winner
            VAR who = random.randint(0, 1)
            if(who == 0):   p_one_score = round_cap
            else:           p_two_score = round_cap

            # Add the match
            < ADD MATCH TO ROUND >
        < ADD ROUND to TOURNAMENT >
```

File Save Handling

```
whilst file_is_open('data/seasons.json', READ_AND_WRITE) referenced as f:
    VAR data = json.load(f)

    if(season exists in data):
        if(tournament exists in data[season]["tournaments"]):
            SET data TO new_rounds

            # Seek back to SOF and write back our data
            f.write( data )
```

Loading Previous Data

```
foreach round in JSON_FILE:
    foreach gender in round:
        VAR r_name = < GET ROUND NAME FROM JSON_FILE >

        # Default Values
        VAR round_cap = < GET SPECIFIC GENDER ROUND CAP FROM JSON_FILE >

        # Data to Check
        VAR prev_r = < GET PREVIOUS ROUND [IF EXISTS] FROM JSON_FILE >

        # Check if we have a round to take data from
        if (prev_r DOES NOT EXIST):
            VAR match_cap = (length_of(ALL_PLAYERS) DIVIDED BY 2)
        else:
            VAR match_cap = (length_of(prev_r.winners()) DIVIDED BY 2)

        VAR matches = < GET MATCHES FROM JSON_FILE >

        foreach MATCH in matches:
            VAR p_one = MATCH.PLAYER_ONE
            VAR p_two = MATCH.PLAYER_TWO

            VAR p_one_score = p_one.SCORE
            VAR p_two_score = p_two.SCORE

            if p_one_score > p_two_score:
               MATCH winners APPEND p_one
            else:
               MATCH winners APPEND p_two

            < ADD MATCH TO ROUND >
        < ADD ROUND to TOURNAMENT >
```