[](http://www.google.co.uk/url?sa=i&rct=j&q=pates+grammar&source=images&cd=&cad=rja&uact=8&ved=&url=http://www.schooltogs.com/schooltogs&ei=JaI3VYXJBYrhaIyNgYAC&psig=AFQjCNFVEgg6wkOcJvB9BV_eWuOJO_YOEg&ust=1429795749580256)



**PATE’S GRAMMAR SCHOOL**

**COMPUTING DEPARTMENT**

**Unit 3/4 – Programming Project**

CANDIDATE NAME

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

EXAM NUMBER

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Table of Contents

[(1) Anlysis of the problem (10 Marks) 3](#_Toc454442805)

[(i) problem definition 3](#_Toc454442806)

[(ii) Stakeholders 5](#_Toc454442807)

[(iii) RESEARCH THE PROBLEM 6](#_Toc454442808)

[(iv) SPECIFIY THE PROPOSED SOLUTION 9](#_Toc454442809)

[(2) Design OF THE SOLUTION [15 marks] 11](#_Toc454442810)

[(i) DECOMPOSE THE PROBLEM 11](#_Toc454442811)

[(ii) DESCRIBE THE SOLUTION 12](#_Toc454442812)

[(iii) DESCRIBE THE APPROACH TO TESTING 13](#_Toc454442813)

[(3) DEVELOPING THE SOLUTION (25 mARKS) 14](#_Toc454442814)

[(i) INTERATIVE DEVELOPMENT PROCESS 14](#_Toc454442815)

[(ii) TESTING TO INFORM DEVELOPMENT 15](#_Toc454442816)

[(4) EVALUATION (20 mARKS) 16](#_Toc454442817)

[(I) TESTING TO INFORM EVALUATION 16](#_Toc454442818)

[(II) SUCCESS OF THE SOLUTION 17](#_Toc454442819)

[(III) DESCRIBE THE FINAL PRODUCT 18](#_Toc454442820)

[(IV) MAINTENANCE AND DEVELOPMENT 19](#_Toc454442821)

[APPENDIX A - BIBLIOGRAPHY 20](#_Toc454442822)

# (1) Anlysis of the problem (10 Marks)

## (i) problem definition

*(a) Describe and justify the features that make the problem solvable by computational methods.*

*(b) Explain why the problem is amenable to a computational approach.*

Scoring for cricket has always been a precise process; each scorer in a team must pay close attention to each ball bowled in the match, making sure that many details are recorded after each ball, which include writing the amount of runs scored, which batsman had scored the run, which bowler is bowling. As many details are needed to be written down each ball, it leads to a tedious process and one which is prone to human error. This could result in an incorrect total being displayed, therefore the overall game may be affected. Due to evolution of technology professional cricket matches are now predominantly scored using online software, leading to accurate results and fast calculation of useful statistics about the game, such as the average runs per over needed to reach the total, or estimated score to finish on based on the average runs per over. This is because a computer is excellent at storing records, holding data, and performing complex calculations efficiently and quickly. The professional game has developed and improved as a whole due to this innovation. However, lower level semi-professional/ amateur cricket still resorts to old methods of having two scorers calculate all of the statistics for a game. The aim of this project would be to create software which allows non-professional cricket games to be able to be efficiently and correctly scored.

Due to it being able to perform standard arithmetic faster than a human, I believe a cricket scorer is an ideal option to be created by a computer. Information such as the current amount of balls bowled, or amount faced by a batsman would be automatically calculated by a computer, whereas a human would have to work these out individually. Certain data can also be saved onto a database, or more simply, a text file. This is useful for saving past matches scored on the system, and loading past scores scored by a batsman. By having a history of data on the computer it gains the potential of being able to dynamically create useful statistics for analysis, such as a batsman’s average score throughout the season, or the amount of wickets taken by a bowler in the season. This is much easier and faster than a human collecting data manually through a scorebook, where they will have to calculate statistics individually, which would be a tedious and extremely long process. A computer will also be ideal as it can back up information, so if the file data gets corrupted a version can be easily restored.

## (ii) Stakeholders

1. *Identify and describe those who will have an interest in the solution explaining how the solution is appropriate to their needs (this may be named individuals, groups or persona that describes the target end user).*

This software is designed for non-professional cricket teams, such as clubs, schools and junior county setups. It is not appropriate to create this software for high level professional teams as there is already an existing solution. However in non-professional matches there are two scorers which write the score into a book, which is then put onto a website at a later date. This causes problems because it is prone to human error: they may not recognize a player and accidentally name him something else, therefore leading to incorrect data when put onto the website. The project will be developed in a way so that a scorebook is not needed, and therefore the scorers will only need to specify what happens each ball. Mostly everything else, such as balls faced, runs scored, bowlers stats etc. will be calculated automatically. When a match is finished, it will be stored on some sort of database which can be accessed at any time, at a later date. This means that they do not have to worry about human error or keeping the scorebook as the records are being stored on a database.

From a players and coaches point of view, this will benefit them a lot as it will provide them with a large amount of statistics for analysis. If, for example, a batsman changes their technique slightly or are given some advice, they can compare the scores they had before that change with the scores they have after to see if it has actually benefitted them. Coaches and selectors will be able to look at each player in the squad’s past games to decide who will be in the team based on performances.

I will be designing this as a laptop application, as it is a portable device so can be easily taken to matches. As it will be designed for 20, 45 and 50 over matches it will not be able to score U9 and U11 Taverner games, or first class games which range from 2-4 days.

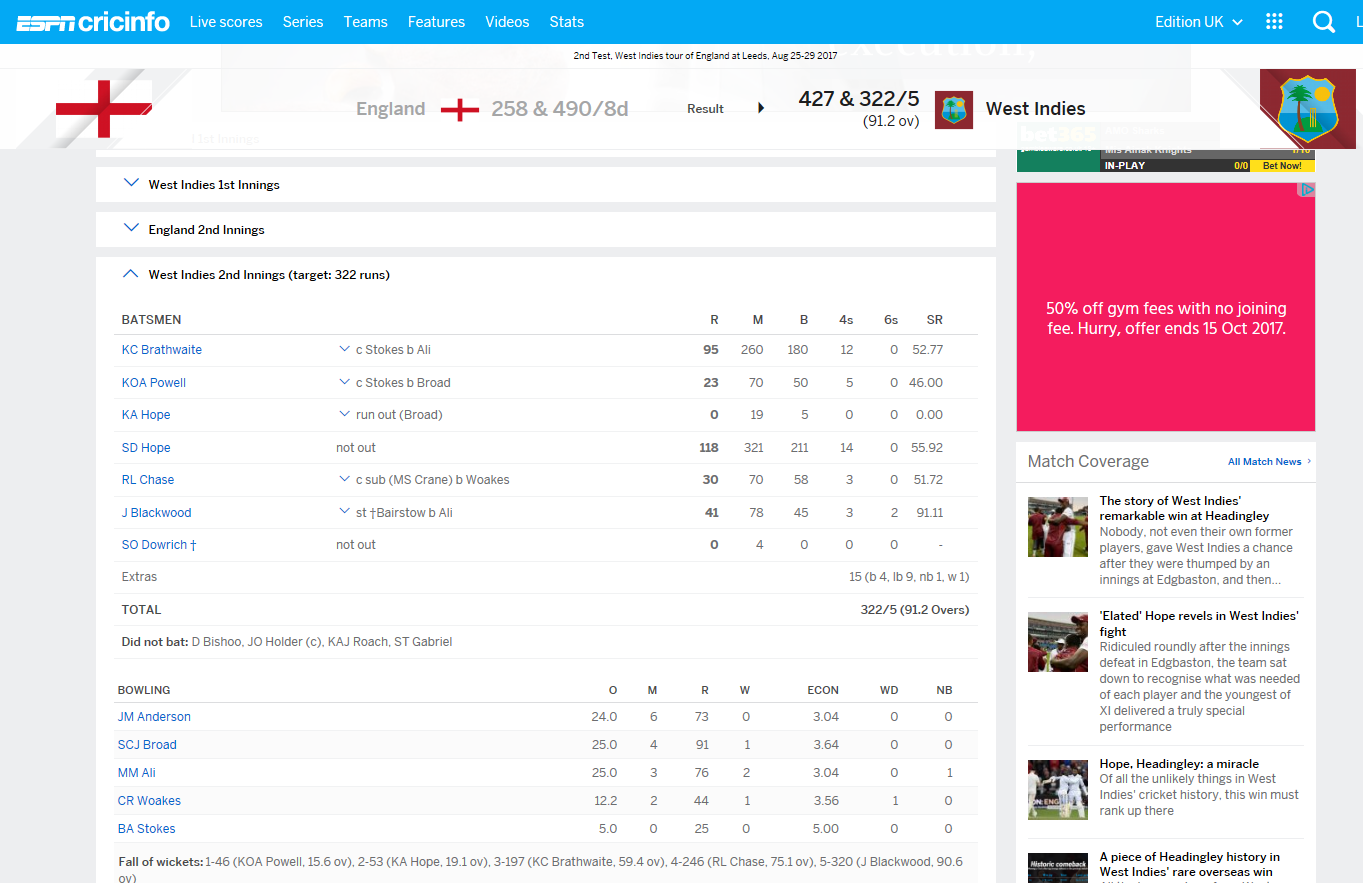
To receive feedback I have asked a classmate, Louis Scott, to review my program as it goes through each stage of development. Louis is the ideal person to choose because he is a club cricketer so his feedback will be beneficial to the development of this project. I will be presenting each iteration of my project to him and asking if any aspect of the program could be improved. I believe this is helpful to the project because exposing an end user to the program in the development cycle will enable me to optimize it to what users like.

## (iii) RESEARCH THE PROBLEM

*(a) Research the problem and solutions to similar problems to identify and justify suitable approaches to a solution.*

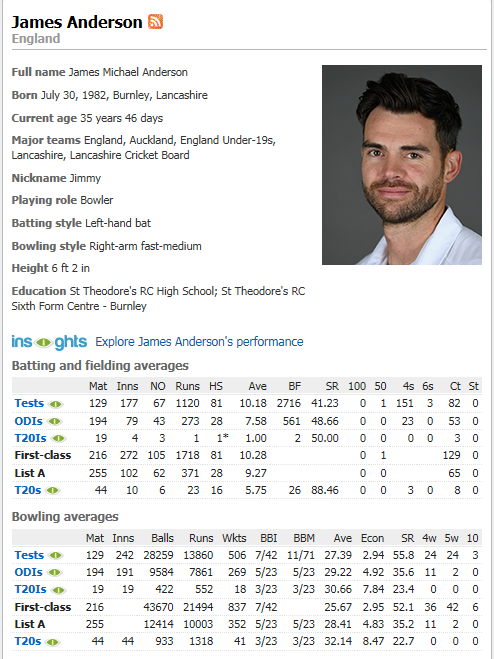
*(b) Describe the essential features of a computational solution explaining these choices.*

*(c) Explain the limitations of the proposed solution.*

A similar solution to this project is the website [*www.espncricinfo.com*](http://www.espncricinfo.com)*,* for professional cricket. There are many features of this website which will be used in this project.

The picture above shows the scorecard of a completed test match. A feature I liked was the simplicity of the heading – it gave the two scores of both of the teams, showing the winner underneath. The user is then able to click on each team and view their two innings, along with all of the batsmen and their scores, showing useful statistics on the side such as strike rate, balls faced, amount of 4s and 6s scored. Below the batsmen is the performance of the bowlers, showing amount of overs bowled, wickets taken, economy etc. I would like to add some of these to this project as these calculations take advantage of the computer’s speed. There is also an option to click on each player taking them to their respective statistics throughout the season and career, which would also be a useful feature to add. The aesthetic of the scorecard is pleasing to the eye and I will be looking to use a similar colour scheme for mine.

A feature which would be beyond the scope of this project is live scoring. This requires various networking techniques, such as setting up a server, and the scorers using the system would have to require Wi-Fi, which may be hard to access if they are travelling to various clubs which don’t have it. Another feature which I will not put in is the option to add test matches, because club level teams do not play more than 50 overs. Adding a season feature to the program may take too long to implement however if there is enough time then I would look to add it to my program.



The picture above shows the statistics page linked to the player on the scorecard. I am not a fan of this layout as it feels overwhelming when you look through the stats - something as simple as finding total wickets taken requires you to look hard through the table to find it. I would like to abstract these details and only put relevant information when a user clicks on the player, such as wickets taken in current season, bowling and batting average, and total runs scored.

**Limitations**

For the program to run the user will need Python 3+ installed, as I am developing it as a .py file. Limitations of this piece of software includes creating the software for a Windows based PC (ideally the user would download the software onto a laptop so they can carry it around to matches). Preferably I would have liked this to be an app for tablets, however I do not have access to an Android phone; I also am not familiar with Objective-C/Swift so development for an iPhone/iPad is not possible. A feature which I would’ve liked to add would have been live scoring, where the majority of the application is on a website and it can be accessed from any device. However, it is beyond the scope of this project as it will take a long amount of time; also I would have to register and pay for web hosting and a domain. The program will not hold data such as strike rates or amount of balls faced/bowled. The player’s age will also not update every year. As the player information is only stored in a JSON file it will not be encrypted as this would take too long to implement.

A finished version of the program will successfully add/delete/read players from a file, display statistics and score the match.

### *Abstraction and visualisation*

The key objects in my project will be:

* An instance of a player, which holds a unique id pointing to the data of a player stored in a JSON file.
* A team, which holds data about 11 players and unique terms such as a team name
* A match, which holds data about the current game, teams, and amount of players
* When a new match is created the 2 teams are searched and accessed from a JSON file, which contains all of the players in the team and a number corresponding to their batting number. The batting team will be displayed according to the order specified in the JSON file, and the bowling team will show the statistics of the current bowlers bowling.
* A ‘State’ object, which will hold necessary functions for all screens used in the program
* Lists which hold the players in a match, which will be read when a match starts

* Lists which hold the players stored in the json file so it can be displayed in the browse players screen

### *Thinking ahead*

The inputs required for this program will be:

* A mouse, so the user can browse through the program clicking buttons, adding to the score, accessing data, etc.
* A Keyboard, so the user has the option of adding player data onto the system by text input

The outputs used this program will be:

* A physical screen, so the user can see what is happening
* A scorecard, to give the general overview of the match
* A screen for adding player details onto the program’s system
* A menu screen to allow the user to access each part of the program
* A screen which displays a certain player’s statistics, requested by the user
* A screen which displays all of the players stored in the JSON file
* A screen which displays all of the teams stored in the JSON file

### *Thinking procedurally*

The states I will need in my program would be:

* A menu state, which is shown at launch as it will display a number of options which the user can click on to take to a certain state.
* An ‘in-play’ state, which is after the user creates a new match. The user will be able to add to the score on a ball-by-ball basis. Also, if there has been an error to previous matches, when in this state the user can edit any other previous matches.
* A browsing state, where the user can access scorecards from certain matches to look at certain data/statistics about the game. In this state it will not be possible for the user to edit information about a match. On startup this will be the default state to prevent unwanted changes in data.
* An edit state on the player creation screen, where the user can add players to the system.

As this is a GUI application, there will be 4 essential functions in each state:

* A function which will be executed when the state is created, *init()*
* A function which will be run continuously when the state is active, *update()*
* A function which will be run continuously to render the graphics on the screen, *render()*
* A function which will check for input events such as polling the mouse position, or polling for a keyboard being pressed, *pollForEvents()*

I will be using a technique of OOP in order to template a state class called inheritance. All states will ‘inherit’ from this base class so all functions listed above will be included in each child class.

In order to be able to switch states I will need to create a state manager. The job of the state manager will be to render the currently active screen, and allow for the states to be changed at any time. It will contain 5 main functions:

* A function which will be executed when the class is created, init(). This function will take in one parameter: the window that the application is running on so the render function can be called. In this function a local variable will also be declared which sets the current state to null.
* A function which will set the current state to a valid state.
* A function which will run the current state’s *update()* function
* A function which will run the current state’s *render()* function
* A function which will run the current state’s *pollForEvents()* function

This system is essential for the type of program I’m creating because I need to be able to instantiate, draw and update certain elements to the screen for each state. For example the menu state:

* On creation of the state, the buttons on the screen will need to be declared at a certain position. Also, this is where we can load in textures and images to customize the look of the menu.
* We need the update function because it will be necessary to check whether the user has sent a form of input to the program, such as clicking on a button to create a new match.
* We also need the render function to actually draw the elements of the menu (program, colors etc.) to the screen so the user will be able to see it.

*Render() and Update()* are called continuously because the program needs to check every second if a graphic has changed, or if the user has clicked on something. It would not work if we only called those functions once.

### *Thinking logically*

As I am running two functions in each state as many times per second, as well as a function run only once, I will need to find some sort of loop to enable me to do this. A simple solution, in pseudocode, would be:

Function entryPoint()

createWindow()

currentState.init()

while(window.isOpen())

currentState.update()

currentState.render()

end while

end function

The main entry point will instantiate a window, and then initialize the current state. It will then check to see if the window is open, and if it is then it will keep on running update and render. This seems like a suitable solution to handle the main loop. In the update function, I will check for user input. To do this, an if statement is needed in the update function:

Function update()

If(keyboard.hasPressed(“ESC”))

closeWindow()

end if

end function

As update is being called continuously, this if statement will be continually checked, therefore the program will be polling for user input correctly.

### *Thinking concurrently*

* *•Are there aspects of the game where more than one thing happens at once? The classic example is the sound player using multi-channel sound to play different sound effects and background music at the same time as updating the game logic.*

There will be many processes running concurrently in this program which I will need to consider.

The menu state will have to render each button, whilst checking for input. Also, when checking for input the button will change colour if the mouse is hovered/clicked on the button.

Similar to the menu state, the player browse state will have to check for input and whether the mouse is hovered/clicked on text. When the state is loaded, the program will have to process and find all of the player names in the JSON data located outside of the program in a file, whilst rendering the screen. The player statistics state will also have to query data from the JSON file whilst rendering the information on the screen.

The match state will have to do a lot of concurrent processing. It will first have to load player data from a team in a JSON file, then have to render the information onto the screen. When the match is underway, it will have to constantly keep updating which batsmen are batting, how many runs they are on, the total score of the team; whilst also checking for mouse input to see if the ‘next ball’ button was pressed. In the player creation screen the program will need to update the screen whilst altering the player JSON file data.

## (iv) SPECIFIY THE PROPOSED SOLUTION

*(a) Specify and justify the solution requirements including hardware and software configuration (if appropriate).*

*(b) Identify and justify measurable success criteria for the proposed solution.*

Aesthetic Requirements:

* The window resolution will be 1280/720 and it will not be resizable.
* The colour scheme I will be using is a light green background with a mixture of other light colours for banners and buttons
* The colour scheme will allow me to use white text in every part of the program
* When hovered over, the buttons will turn slightly brighter which indicates to the user more clearly that they are hovering over a button. Also, when buttons are pressed the colour will turn brighter again so the user will know that they are clicking on a button
* The text box will have a grey-white background so the user can easily distinguish it from the background of the program
* The scorecard will be in the shape of a rectangle with a pale green background. Within the scorecard will be player names equally spaced to look like a scorecard
* The font will be a mixture of Roboto bold, normal and thin

Input Requirements:

* Navigation of the program will be carried out through clicking on buttons displayed on the screen
* There will be a text entry field for the user to add a player on the system

Processing

* The software will need to calculate various statistics and information:
  + Total runs scored
  + Total wickets taken
  + Matches played
  + Age
* The software will also need to be able to parse and create JSON files, so player data can be saved and read on termination and creation of the program

Outputs

* The menu will contain three buttons:
  + Create new match
  + Browse players
  + Quit
* In create new match:
  + There will be a list of teams stored in the JSON file which will be displayed on the screen for the user to be able to select two teams for a matchup
  + After the user selects two teams, an option will come up which will ask the user the team which will bat first
* On the match screen:
  + There will be a set of buttons labelled 0-6 so the user can click on the amount of runs scored on the current ball.
  + There will be a button for when the team takes a wicket
  + There will be a button used when the ball has been bowled to add the current score/wickets to the total score/wickets
  + There will be text displaying the teams total score, current score of the ball, innings number, target score and the scorecard for batsmen and bowlers
* In browse matches:
  + There will be a search function to view the matches played in the current season
  + The user will be able to click on these matches to view the scorecard for them
* In browse players:
  + This screen will show all of the players on the JSON file
  + The user will be able to click on each player to view their statistics for the current season
* Scorecard display:
  + The scorecard will be contained inside of a rounded rectangle. The batting section will have grid lines which splits the rectangle into rows, to separate each player, displaying their runs and balls faced on the right hand side of the row. This will be similar for the bowlers, except showing overs bowled, runs conceded and wickets taken
  + To calculate runs per ball, the user will click on a certain button to add the amount of specified runs. There will be another button to confirm that the ball has been bowled so the runs can be added onto the score.
  + The buttons on the match screen will be labelled 0-6. The confirm button will have a “+” sign. There will also be some text representing the amount of runs that has been scored on the current ball so the user can check if they’ve input the runs correctly.
* Player Statistics:
  + This screen will feature all of the player’s data stored in the JSON file. It will display the requested player’s name and their respective stats such as wickets, runs, averages, matches played, matches won. A rough mockup of this is show below

**Success Criteria**

The program will:

* Add a player to a JSON file containing name, age and wickets/runs scored
* Read a list of all players stored in a JSON file
* Delete a player from a JSON file
* Store teams in a JSON file
* Score matches, where the user can select teams read from a JSON file
* Respond to user input including mouse presses and keyboard presses
* Contain a menu system which the user can navigate through elegantly

# (2) Design OF THE SOLUTION [15 marks]

## Choice of Language

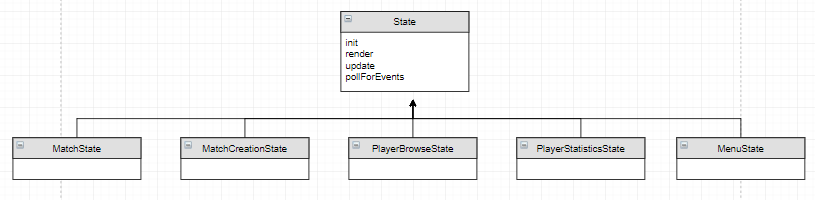
For this project the language I will be using is Python. There are many reasons to this; Python has a library, named JSON, which can efficiently and effectively parse JSON data into a python array. The reason I am using JSON rather than storing player data in a string is the flexibility of it: I am able to edit the data in a text file without changing the code in my program. Also, if the user were to make changes to the players, they would not save on exit of the program if the data stored was in a string. JSON also contains handy data structures such as storing dictionaries, arrays and strings, so I can have all of the players in a string array which allows for customizability and ease of access. Python also has simple file handling operations so reading the JSON data into a python string will be a relatively trivial process. When the JSON data is imported into an array of strings in python, the JSON library can detect which pieces of data are arrays, lists and dictionaries from a JSON file so this will allow me to easily query data from a string and process it to my liking.

As this is an application which requires GUI elements and drawing/rendering to the screen, I will need a library which will allow me to do this. Python has the ideal solution for this: Pygame. Pygame allows me to easily create renderable windows with event handling so I can create visual displays without too much hassle. Pygame also contains useful functions relevant to my program such as rendering images and displaying text to the screen, which is mostly the GUI functionality I need. With features such as mouse event polling, this will allow me to implement buttons and input, which is needed in my program.

Python is also object oriented, which is ideal for my project as I am going to have many objects which exist, such as a screen state. Classes are also useful for state handling. I can make a state manager class which contains useful functions such as changeState() to easily swap between screens. By using classes it will reduce the amount of code I need, with features such as inheritance. For example, I can have a template class called state, with render(), update() and pollEvents(). I can then create different states (menuState, scoreState) as classes which inherit the base state class. This will allow me to add extra features to each state, giving me a lot of customizability whilst reducing the amount of code needed. If I were to implement this without using classes it would require a lot more code and may become messy to use.

## Inheritance

To show how I will use object oriented programming, I will use an inheritance diagram to describe the structure of each screen:



Inheritance is when a class is based on another one, using the same implementation. It allows me to easily create a class which is built upon another class. There will be a parent class named state which will contain all the necessary functions in order for a screen state to display on the screen. Each ‘child’ class under State will inherit from this class, which will save me from describing what each function does in each state I program. The procedures used in every class inheriting State are:

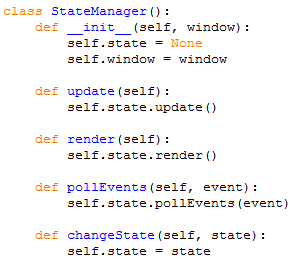
*init(stateManager) –* This is a constructor; it is called when the class is instantiated. Any variables needing initialization before use for any states will be declared in this method. The method takes in an object of StateManager as a parameter – this is so that each state can be changed and can be drawn to the screen.

*render(window) –* This method is for drawing to the screen*.* It takes in a parameter called window – this is a reference to the window created by pygame so that it knows which window to draw to the screen.

*update() –* This is the method which will be called 60 times per second by pygame’s inbuilt clock. Any logic decisions, such as checking if a button is pressed, will be undergone in this method.

*pollForEvents(event) –* This is the method for declaring any object which is interactive, such as a button, to be able to check for events. It takes in a parameter named event which is a pygame variable which returns any input event(keyboard/mouse) and lists it.

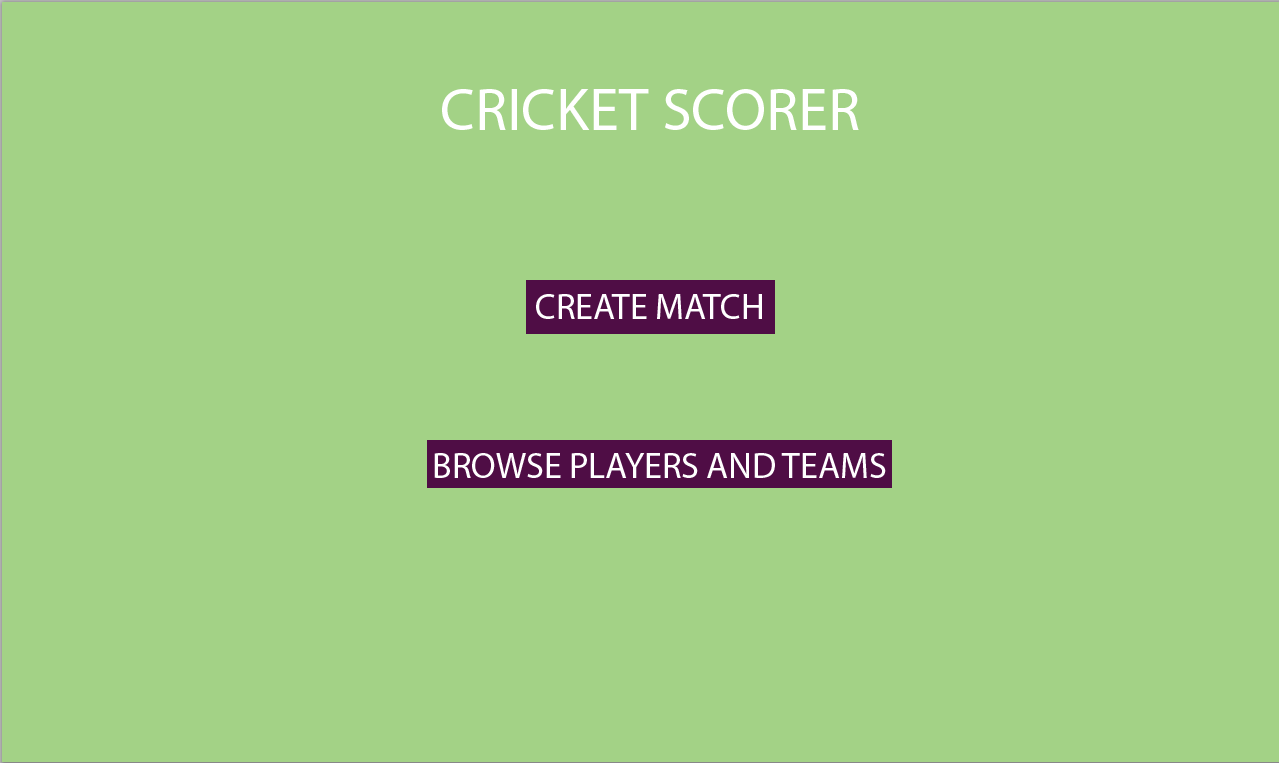
The way each state will be drawn to the screen is using a StateManager class. This class will take in one parameter when it is instantiated, which will be a State class. This is why inheritance is useful in this situation: because each child class inherits the state class, I can pass in any state I want into StateManager, giving me flexibility when debugging, so I don’t have to browse through each state in the program to get to a specific one. The StateManager will have similar functions to the State class; *init, update, render* and *pollForEvents* however it will have another function, *changeState*. This takes in a state in the parameter which changes the state that is rendering in *render*, allowing me to easily move screens with one function. Below is a snippet of code showing the StateManager class:



In the following part of this write up I will break down the program into each screen state, describing what each part does, showing a mockup of the screen, a flowchart and pseudocode. I have broken each flowchart down into a simple process describing what the purpose of each screen does. In order to focus on describing each screen, I have abstracted irrelevant information such as the color of the background, font of text etc.

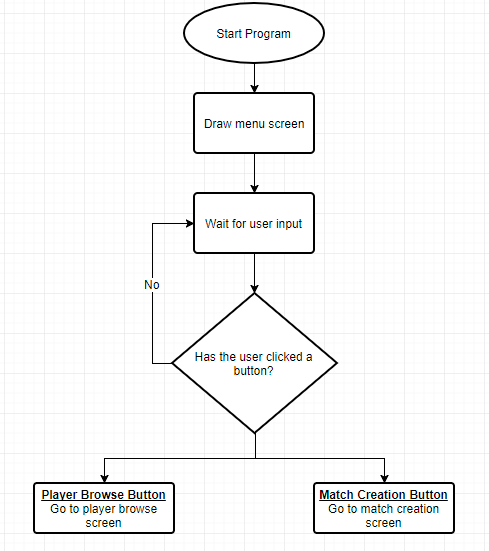
## Menu Screen

### Mockup

This is the screen which will be the entry point for the program. The user will be able to click two buttons – one going to the player browsing screen and one which goes to the match creation screen. All screens will inherit from a State superclass, as explained above. Below is a mockup of what the menu screen will look like:

### flowchart

The flowchart below shows the abstracted procedure that the main menu will have. Every screen in the program will have the first three procedures: drawing to the screen, waiting for user input and waiting until the user has clicked a specific button. To save time making multiple conditional diagrams on a flowchart to determine which button the user has pressed, I will display the multiple choices as a case statement. This is achieved by branching multiple processes out from one conditional statement.



Example of a case statement

### Variables & Methods

These are the essential variables/methods needed for the menu screen:

*Render() –* inherited from state

*Update()* – inherited from state

*PollForEvents() –* inherited from state

PlayerButton – object of class Button which represents the button to go to the player screen

MatchCreationButton – object of class Button which represents the button to go to the match creation screen

### PSEUDOCODE

*CLASS MENUSCREEN INHERITS STATE:*

*FUNCTION RENDER:*

*DRAW SCREEN*

*END FUNCTION*

*FUNCTION UPDATE:*

*IF PLAYERBUTTON.PRESSED:*

*CHANGESCREEN(PLAYERBROWSE)*

*IF MATCHCREATIONBUTTON.PRESSED:*

*CHANGESCREEN(MATCHCREATION)*

*END FUNCTION*

*FUNCTION POLLFOREVENTS:*

*PLAYERBUTTON.LISTENEVENTS*

*MATCHCREATIONBUTTON.LISTENEVENTS*

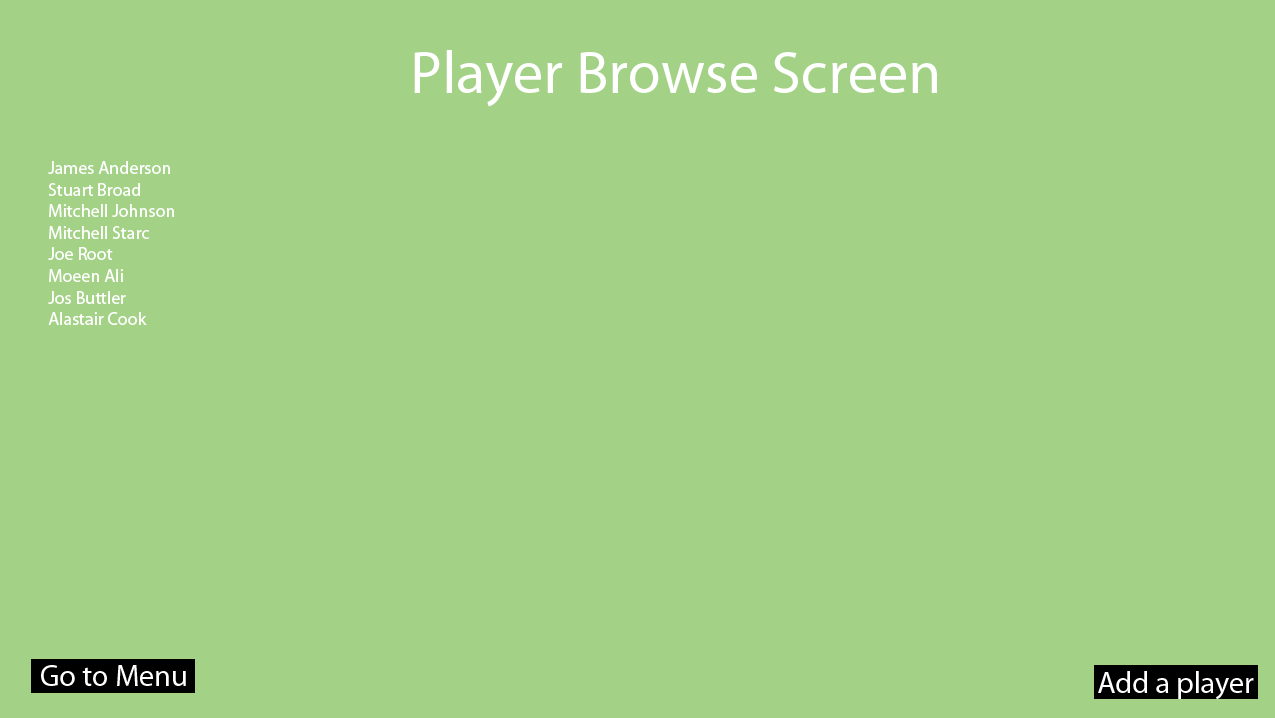
*END FUNCTION*

Although the function init will be written, the menu screen does not make use of it so I have discarded it in my pseudocode.

## Player Browse Screen

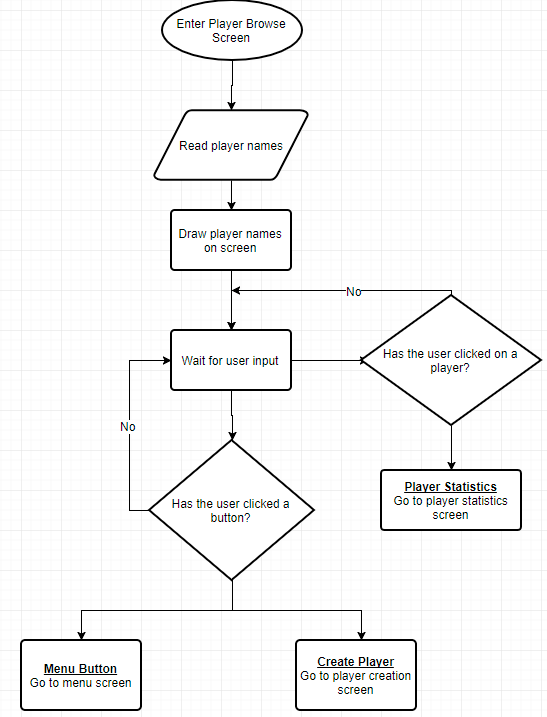
### Mockup

This is the screen where it displays all of the players stored in the JSON file. The user is able to click on each player which will change state to go to the player’s statistics screen. The user will also be able to add a player to the JSON file, using a button which will take it to the player creation screen. There is also a button which allows the user to go back to the menu. Below is a mockup of the screen:



### Flowchart

This flowchart is slightly different to the menu screen as it has to read data from a JSON file, and display each player read onto the screen. I will be using a case statement in each flowchart for the button checks as it will be easier to read.



### Variables & Methods

These are the essential variables/methods needed for the Player Browse screen:

*Render() –* inherited from state

*Update()* – inherited from state

*PollForEvents() –* inherited from state

playerData – string taken from JSON file holding the player data

MenuButton – Button object placed in order for the user to navigate back to the menu screen

CreatePlayerButton – Button object placed in order for the user to navigate to the create player screen

### Pseudocode

*CLASS PLAYERBROWSESCREEN INHERITS STATE:*

*FUNCTION INIT:*

*ARRAY PLAYERS = READFROMJSON(“DATA.JSON”)*

*END FUNCTION*

*FUNCTION RENDER:*

*FOR PLAYER IN PLAYERS:*

*DRAW PLAYER*

*DRAW MENUBUTTON*

*DRAW PLAYERCREATEBUTTON*

*END FUNCTION*

*FUNCTION UPDATE:*

*FOR PLAYER IN PLAYERS:  
 IF PLAYER.ISPRESSED:*

*CHANGESCREEN(PLAYERSTATISTICS)*

*IF MENUBUTTON.ISPRESSED:*

*CHANGESCREEN(MENU)*

*IF PLAYERCREATEBUTTON.ISPRESSED:*

*CHANGESCREEN(PLAYERCREATION)*

*END FUNCTION*

*FUNCTION POLLEVENTS:*

*FOR PLAYER IN PLAYERS:*

*PLAYER.LISTENEVENTS*

*MENUBUTTON.LISTENEVENTS*

*PLAYERCREATEBUTTON.LISTENEVENTS*

## Player Statistics Screen

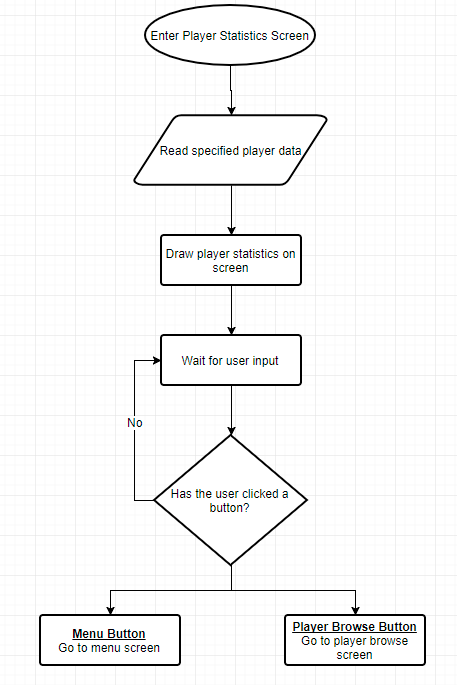
### Mockup

This is the screen for browsing the statistics for a player. It will be fairly similar to the player browse screen because I am just retrieving a specific player’s information from the JSON file and displaying it on the screen. There will also be two buttons, one for navigating back to the menu and one which navigates back to the player browse screen. Below is a rough mockup of the player statistics screen:



### Flowchart

Shown below is a flowchart of the player statistics screen. It is a fairly simple and straightforward screen to display so there is not much to show:



### Variables & Methods

These are the essential variables/methods needed for the Player Statistics screen:

*Render() –* inherited from state

*Update()* – inherited from state

*PollForEvents() –* inherited from state

playerData – string file holding the specified player data from the JSON file

MenuButton – Button object placed in order for the user to navigate back to the menu screen

PlayerBrowseButton – Button object placed in order for the user to navigate to the browse player screen

### Pseudocode

*CLASS PLAYERSTATISTICS INHERITS STATE:*

*FUNCTION INIT(PLAYER):*

*READSTATSFROMJSON(PLAYER)*

*END FUNCTION*

*FUNCTION RENDER:*

*DRAW PLAYER STATISTICS*

*DRAW MENU BUTTON*

*DRAW PLAYERBROWSE BUTTON*

*END FUNCTION*

*FUNCTION UPDATE:*

*IF MENUBUTTON.ISPRESSED:*

*CHANGESCREEN(MENU)*

*IF PLAYERBROWSEBUTTON.ISPRESSED:*

*CHANGESCREEN(PLAYERBROWSE)*

It is worth noting that when this class is instantiated is that it takes a parameter, which is a string referring to the player in the JSON file to be searched.

## Player Creation Screen

### Mockup

This is the screen where the user can enter data to be stored into the JSON file. It will constitute of a few buttons and text input, so the user can enter in their credentials. Below is a rough mockup of the screen:

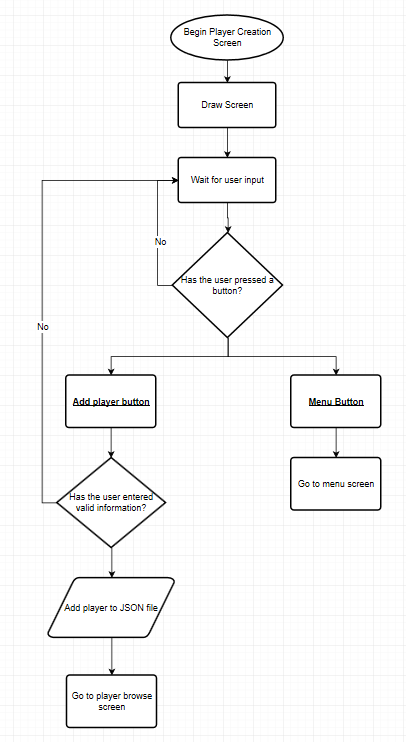


As there are no libraries for text boxes and UI in pygame, I will have to create my own text input class. This means it will be left with limitations, therefore I will need to come up with a simple system to allow the user to enter the age, full name and team on one line. To achieve this I will ask the user to enter the details in a specific way so I can parse the data and add it to the JSON file correctly. The way I will ask the user for data is to input the following:

*Full name:age country*

### Flowchart

The flowchart below describes how the player creation screen will function. There is a check to see if the player has entered valid data using the pattern shown above. If the verification fails then nothing will happen. If it succeeds, then the data will be parsed and added to the JSON file, then the program will be sent to the player browse state so the user can see the newly added player.



### Variables & Methods

These are the essential variables/methods needed for the Player Creation screen:

*Render() –* inherited from state

*Update()* – inherited from state

*PollForEvents() –* inherited from state

AddPlayerButton – Button object in order for the user to add a player to the JSON file

MenuButton – Button object in order for the user to navigate back to the menu

### Pseudocode

*CLASS PLAYERCREATIONSCREEN INHERITS STATE:*

*FUNCTION INIT():*

*INITIALISE TEXTINPUT*

*INITIALISE BUTTONS*

*END FUNCTION*

*FUNCTION RENDER():*

*DRAW SCREEN*

*DRAW TEXTINPUT*

*END FUNCTION*

*FUNCTION UPDATE():*

*IF ADDPLAYERBUTTON.ISPRESSED():*

*IF TEXTINPUT.DATA.ISVALID():*

*ADDTOJSON(PLAYER)*

*CHANGESCREEN(PLAYERBROWSE)*

*END IF*

*END IF*

*IF MENUBUTTON.ISPRESSED():*

*CHANGESCREEN(MENUSCREEN)*

*END IF*

*END FUNCTION*

*FUNCTION POLLEVENTS():*

*TEXTINPUT.POLLEVENTS()*

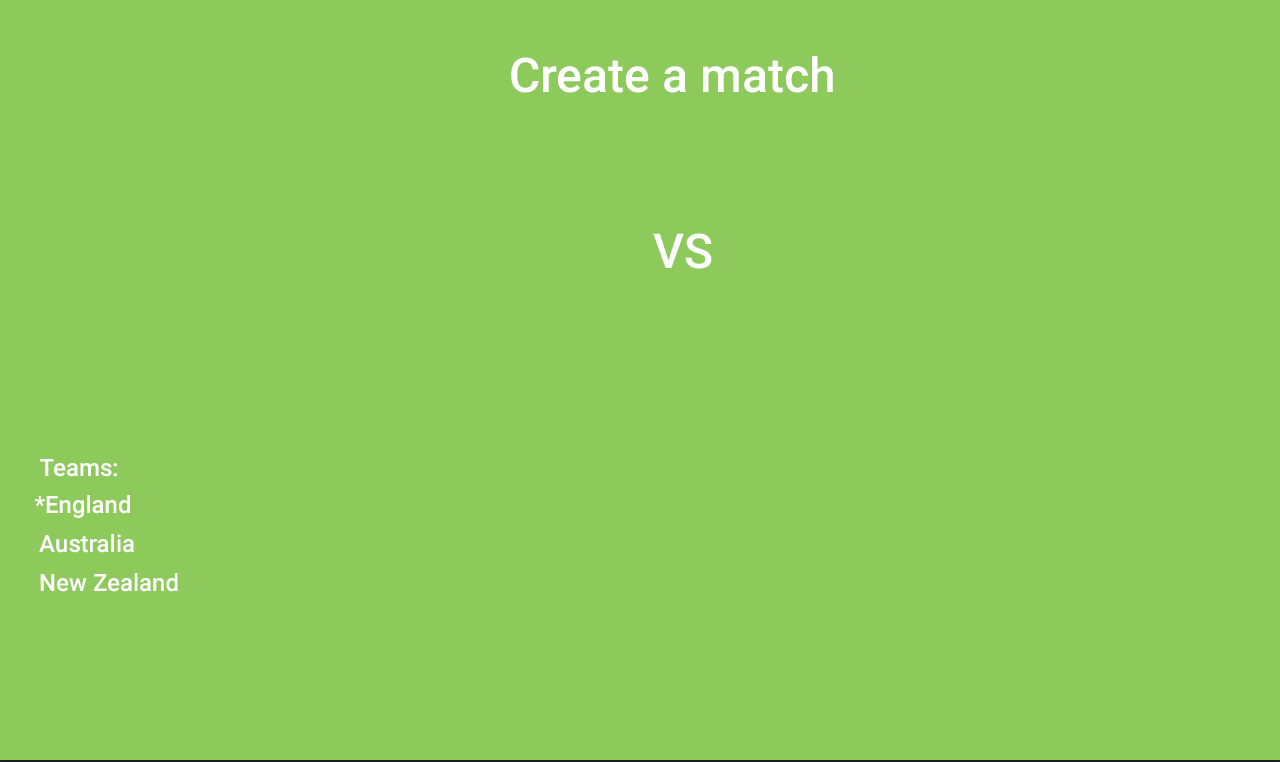
*ADDPLAYERBUTTON.POLLEVENTS()*

*MENUBUTTON.POLLEVENTS()*

*END FUNCTION*

## Match Creation Screen

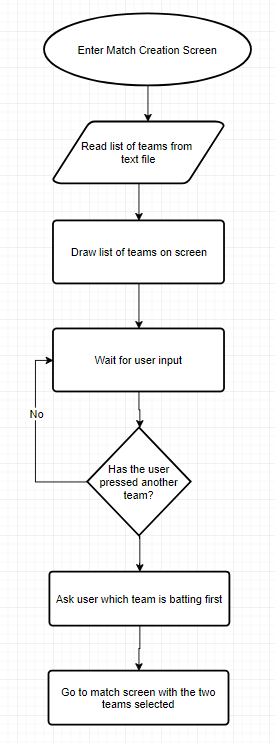
### Mockup

This will be the screen where a player can choose which team is playing each other. The teams will be read from the JSON file, and displayed on the screen accordingly. They will also have to input which team is batting first so the actual match state will know which team to display. Below is a rough mockup of the screen:

When two teams are selected they will appear here

The asterisk indicates the current team selected

### Flowchart

The flowchart on the left describes how the match screen will function. Firstly, teams will be read from a file and displayed onto the screen. When a user pressed two teams, another block of text shows up which asks which will be batting first. After this the scene changes to the main match screen with the two teams selected.

### Variables & Methods

These are the essential variables/methods needed for the Match Creation screen:

*Render() –* inherited from state

*Update()* – inherited from state

*PollForEvents() –* inherited from state

teamData – string holding teams stored in the JSON file so the list of teams to select will be displayed on the screen

### Pseudocode

*CLASS MATCHCREATIONSCREEN INHERITS STATE:*

*FUNCTION INIT():*

*READDATAFROMJSON(TEAMS)*

*END FUNCTION*

*FUNCTION RENDER():*

*DRAW TEAMS ON SCREEN*

*IF TEAMS.SELECTED == TRUE:*

*DRAW TEXT ASKING THE USER FOR ORDER OF BATTING*

*END IF*

*END FUNCTION*

*FUNCTION UPDATE():*

*IF TWO TEAMS SELECTED:*

*TEAMS.SELECTED = TRUE*

*END IF*

*IF TEAMS.SELECTED == TRUE:*

*IF ORDERSELECTED:*

*CHANGESCREEN(MATCHSCREEN)*

*END IF*

*END IF*

*END FUNCTION*

*FUNCTION POLLEVENTS():*

*FOR TEAM IN TEAMS:*

*TEAM.LISTENEVENTS()*

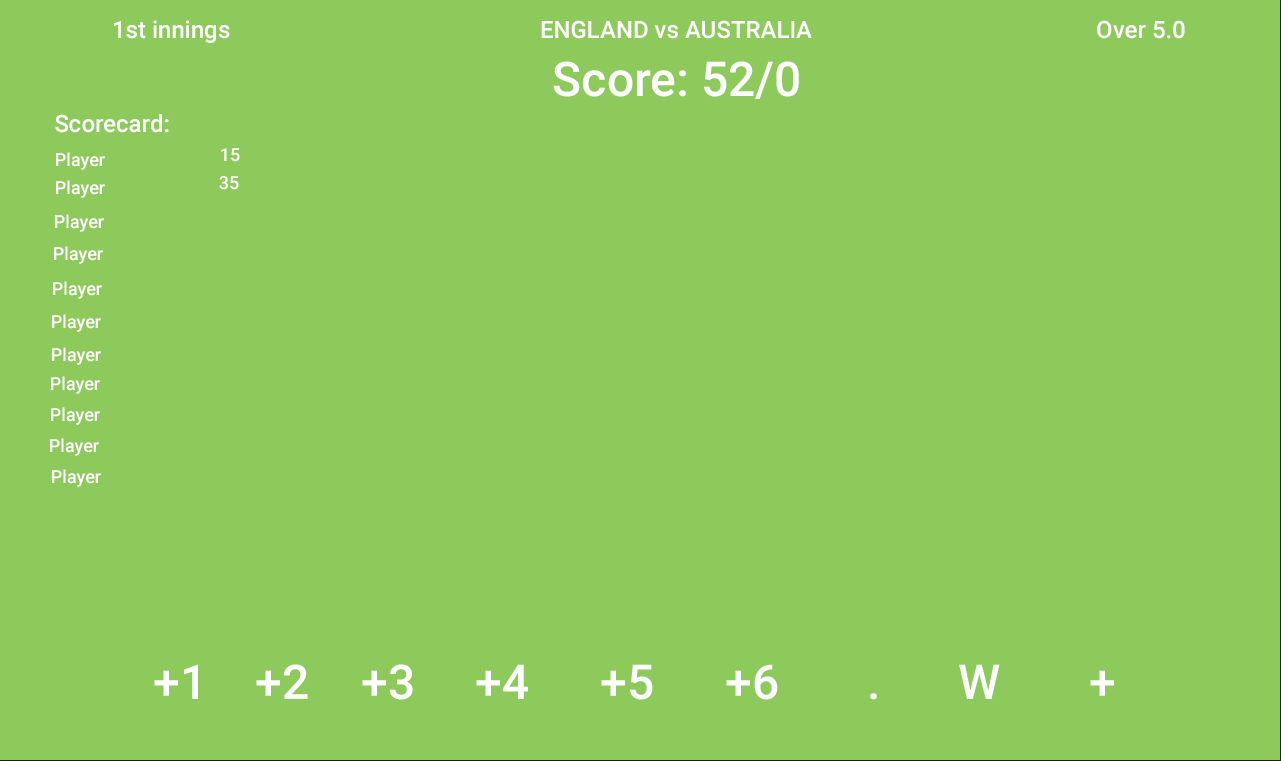
*END FOR*

*END FUNCTION*

## Match Screen

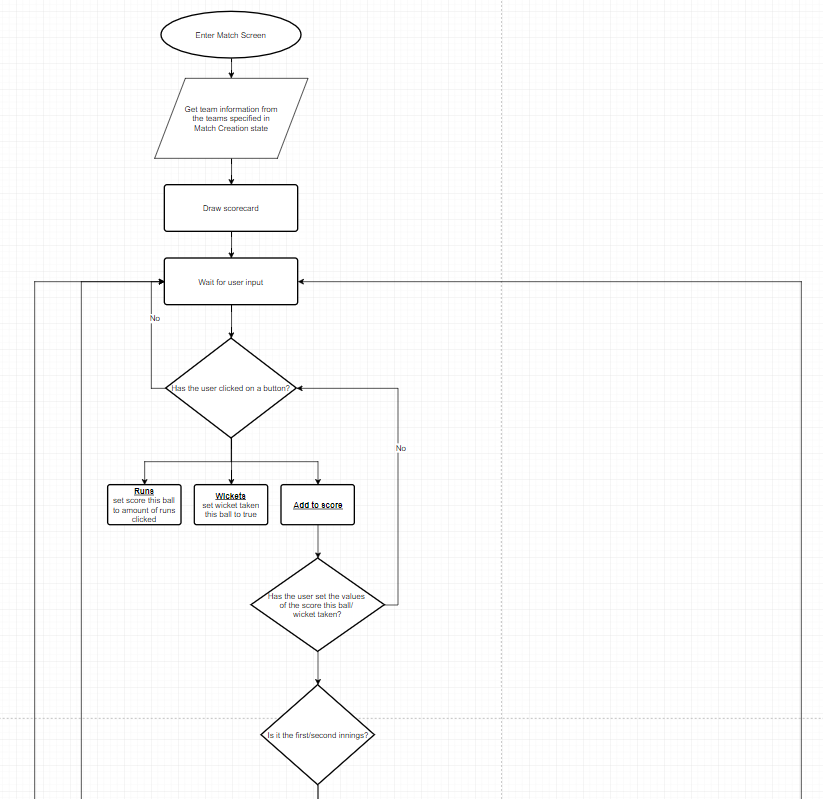
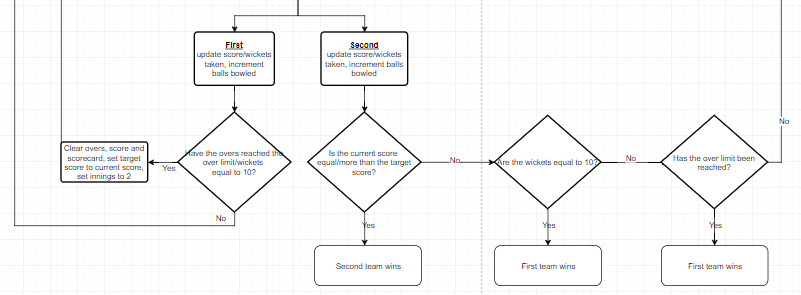
### Mockup

This will be the most complicated screen in the program because it will be scoring the match. It will be keeping track of most things an actual scorer does – updating individual batsmen’s score, which batsmen is facing, updating the total score and target score, etc. Below is a rough mockup of what the screen will look like:



Buttons for the scorer

### Flowchart

The picture below shows the structure of the class and how it will be implemented. It will be a very large class so I have tried to simplify it as much as possible to only include relevant information integral to the function of the program:

### Pseudocode/Variables & Methods

This is a large class so I will define a few key match variables before I write the pseudocode to describe what the purpose of each one is. These are not all the variables used in the class, only the ones which are relevant to the problem.

*Innings –* an integer to keep track of which innings it is. Can only be 1 or 2.

*scoreThisBall –* an integer to hold the score of the ball being bowled

*overs –* an integer to show the amount of overs currently bowled. Can only be between 0 and *maxOvers.*

*maxOvers –* an integer to show the maximum amount of overs to be bowled in the match. Can only be 20, 40, 45 or 50.

*wicketTaken –* a Boolean to check whether a wicket was taken on the current ball.

*totalScore –* an integer to keep count of the total score in the innings.

*targetScore –* an integer to hold the target score needed. It is set to 0 until the 2nd innings has started.

*totalWickets –* an integer to hold the amount of wickets currently taken.

*BEGIN MATCHSCREEN INHERITS STATE:*

*FUNCTION INIT():*

*READFROMJSON(TEAMS)*

*INITIALISE MATCH VARIABLES*

*END FUNCTION*

*FUNCTION RENDER():*

*DRAW SCORECARD*

*DRAW BUTTONS*

*IF INNINGS == 2:*

*DRAW TARGETSCORE*

*END FUNCTION*

*FUNCTION UPDATE():*

*IF ADDTOSCORE.ISPRESSED() & SCORETHISBALL.ISSET:*

*IF WICKETTAKEN == TRUE:*

*WICKETTAKEN = WICKETTAKEN + 1*

*IF WICKETTAKEN == 10:*

*IF INNINGS == 1:*

*INNINGS == 2*

*END IF*

*ELSE IF INNINGS == 2:*

*CHANGESCREEN(ENDMATCHSCREEN(TEAM1))*

*END IF*

*END IF*

*END IF*

*TOTALSCORE = TOTALSCORE + SCORETHISBALL*

*OVERS = OVERS + 0.1*

*IF OVERS == MAXOVERS:*

*IF INNINGS == 1:*

*INNINGS == 2*

*END IF*

*IF INNINGS == 2:*

*IF TOTALSCORE > TARGETSCORE:*

*CHANGESCREEN(ENDMATCHSCREEN(TEAM2))*

*END IF*

*ELSE IF TOTALSCORE == TARGETSCORE:*

*CHANGESCREEN(ENDMATCHSCREEN(DRAW))*

*END IF*

*ELSE:*

*CHANGESCREEN(ENDMATCHSCREEN(TEAM1))*

*END IF*

*END IF*

*IF INNINGS == 2 & TOTALSCORE > TARGETSCORE:*

*CHANGESCREEN(ENDMATCHSCREEN(TEAM2))*

*END IF*

*END IF*

*END FUNCTION*

*FUNCTION POLLEVENTS():*

*BUTTONS.LISTENEVENTS()*

## The Utility File

### Overview

In collaboration with the screen states, I will need to write a few classes for functions I have not described yet. This includes a text class, a button class, and a text input class. This is because they are not written in pygame therefore I will need to write my own. All of these classes will be stored in a ‘utility’ file which makes it easier to organize as if I were to put it in the same file as all the state classes then it would be hard to navigate to them.

### The Button Class

This class will be responsible for loading in an image, checking to see if it has pressed, and returning a Boolean with the correct answer. The variables required are:

*x, y –* two integers that are taken from the constructor to determine where the button will be placed

*image –* a pygame image file which loads the image of the button into memory

*pressed –* a Boolean to determine whether the button is pressed or not

Below is pseudocode written to describe how the class will work:

*CLASS BUTTON:*

*FUNCTION INIT(IMAGE, X, Y)*

*INITIALISE CONSTRUCTOR VARIABLES*

*PRESSED = FALSE*

*END FUNCTION*

*FUNCTION RENDER():*

*DRAW IMAGE AT X,Y*

*END FUNCTION*

*FUNCTION POLLEVENTS()*

*GET MOUSE POSITION*

*IF MOUSE.X > X & MOUSE.X < X + IMAGE.WIDTH AND MOUSE.Y > IMAGE.Y AND MOUSE.Y < IMAGE.Y + Y:*

*IF MOUSE.ISPRESSED:*

*PRESSED = TRUE*

*ELSE:*

*PRESSED = FALSE*

*END FUNCTION*

The code above takes in an image and an x and y coordinate from the constructor to display an image onto the screen. It then polls for the mouse coordinates in *PollEvents()* and checks to see if the x and y coordinates are between the top left of the button and the bottom right of the button, by checking if it is greater than the x and y position of the image and smaller than the x and y position plus the image x and y.

### The Text Class

This simple class will be used in all of the screens, in order to display text converted from a string. The variables used in this class will be:

*x, y –* the x and y position of where the text shall be placed on the screen

*text –* a string which holds the text to be displayed onto the screen

As pygame has a function for converting a string into renderable text on the screen, this class will be fairly simple to implement. Below is the pseudocode for the class:

*CLASS TEXTRENDERER:*

*FUNCTION INIT()*

*INITIALISE PYGAME.FONT*

*END FUNCTION*

*FUNCTION DRAWTEXT(TEXT, X, Y):*

*PYGAME.FONT.RENDER(TEXT, X, Y)*

*END FUNCTION*

### The Text Input Class

This class will be used in the player creation screen, where a user can input a player name, age and team. The variables required are:

*x, y –* the x and y position of where the text input class shall be placed on the screen

*textBuffer –* a list of characters which holds the key typed on the keyboard

*textRenderer –* an instance of the TextRenderer class which I have written above

The logic for this class is fairly simple. I will poll for a user pressing a key using pygame’s *event* variable, which will return the numerical ASCII value of the key. I will then use python’s *chr()* method which will convert the ordinal value into a character. This will then be added to the *textBuffer* list and rendered using the *textRenderer’s drawText()* function. If the user presses the backspace key, then the textBuffer will be popped. Below is the pseudocode for the program:

*CLASS TEXTINPUT:*

*FUNCTION INIT(x, y):*

*INITIALISE X, Y*

*END FUNCTION*

*FUNCTION RENDER():*

*FOR I IN RANGE(LENGTH(TEXTBUFFER)):*

*TEXTRENDERER.DRAWTEXT(TEXTBUFFER[I])*

*END FOR*

*END FUNCTION*

*FUNCTION POLLEVENTS():*

*IF EVENT.KEYPRESSED:*

*IF EVENT.KEY == BACKSPACE:*

*TEXTBUFFER.POP()*

*ELSE:*

*TEXTBUFFER.APPEND(CHR(EVENT.KEY))*

*END IF*

*END IF*

*END FUNCTION*

## Saving data

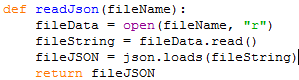
### Approach

As explained in the analysis, I will be using JSON to store player data as it is widely used and supported by Python with a JSON library to allow me to easily interact with these files. I will implement a procedural approach to this part of the code as it will be a fairly short part of the section and won’t be reused.

### Implementation

The methods I will use are:

readJson(filename)

This function will return a variable which holds all of the data in the JSON file in an array. Firstly, it will take in a string named filename as a parameter to get the file name of the JSON file. It will then use Python’s inbuilt function, open, to retrieve the contents of the file. After, it will convert the contents into a string file using .read() on the file variable and finally convert it into an array using json.loads().

The variable returned is fileJSON

saveFile(fileJSON)

This function will take in fileJSON and overwrite the file originally retrieved from readJson to the current contents stored in fileJSON. It will use a function from the json library, json.dump, to overwrite the file.

addPlayer(fileJSON, playerName, playerAge, playerTeam)

This procedure will create a player and store it in the JSON file with appropriate formatting. It takes in the fileJSON variable, as well as a player name, age and team. Using these parameters, it will append the data to the fileJSON variable and add the player name, age and team.



This procedure links in with the program well because now in the player creation screen, when the data is parsed, I can call this procedure in that subroutine, easily adding a player with one line of code.

## Testing

### Text Input

In order to verify that adding players to the JSON file works, I will implement a test plan for development. I will create a table which will have test values as an input, and decide which a valid, borderline and invalid entry is:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test data | Test type | Predicted outcome | Actual outcome | Pass/Fail |
| Mitchell Johnson:36 Australia | valid | Saves to JSON file |  |  |
| Joe Root:26 England | valid | Saves to JSON file |  |  |
| Dawasd afaf:421 Paris | borderline | Saves to JSON file |  |  |
| John:19 England | Invalid | Rejects text input |  |  |

As explained in the text input section the required valid input for adding a player is the following:

Firstname Lastname:Age Team

Therefore following this pattern it means that the first two test data to be used are valid, and should add the player to the JSON file as required. Although the third test data is not an actual name, it still follows the same pattern specified therefore it still should work. However it is unlikely that a user will input something like this so it is borderline. The last test data should return invalid because the program requires a last name too, so it should just reject the text input.

### Match Creation

In the Match Creation screen the user is displayed a collection of teams stored in the JSON file. In order to create a valid match there must be a check to see if the selected teams are different, as two same teams cannot play each other. If two of the same teams are selected then the user will be notified that it is an error. The test data below are examples of two teams which can be selected from the JSON file.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test data | Test type | Predicted outcome | Actual outcome | Pass/Fail |
| ‘England’ and ‘New Zealand’ | valid | Goes to match screen |  |  |
| ‘Australia’ and ‘Gloucestershire’ | borderline | Goes to match screen |  |  |
| ‘West Indies’ and ‘West Indies’ | invalid | Stays at Match creation screen |  |  |

The first test data should return valid as two different teams are selected therefore it goes to the match screen. Although the second test data will go to the match screen, it is borderline because the teams selected are a mix of a national and county team, which is unlikely to ever happen.

### Match State

In the match state, the user will press a button corresponding to the amount of runs scored on the current ball, and then press another button to confirm it, and increment the ball counter by one. Therefore, if the user presses the ‘confirm’ button without pressing any current score buttons, then an error must be handled in order for the program to behave as expected. Test data will be created in order to check if the user presses a button before. The test data simulates the sequence of buttons pressed:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test data | Test type | Predicted outcome | Actual outcome | Pass/Fail |
| Button ‘4’ -> Button ‘Confirm’ | valid | Increments ball counter by 1 |  |  |
| Button ‘0’ -> Button ‘Confirm’ | valid | Increments ball counter by one |  |  |
| Button ‘Confirm’ -> Button ‘3’ | invalid | Ball counter is not incremented |  |  |
| Button ‘Confirm’ | Invalid | Ball counter is not incremented |  |  |

The first two test data should be valid as a current score button is pressed prior to the confirm button. Following this pattern, the last two data should be invalid as confirm is pressed first therefore the ball counter should remain unchanged.

Another test plan for the match state is to check see if the winning team is calculated correctly. There are many different ways a team can win so these combinations need to be tested:

Try and reduce this diagram to it’s smallest component parts.

You may wish to reference some of the computational think in this section by refereeing to:

* Abstraction – what have you discarded?
* Thinking ahead – what problems/limitations can you foresee?
* Thinking concurrently – quickly dismiss any thought of using cores independently
* Reference the next section to thinking procedurally

## (ii) DESCRIBE THE SOLUTION

*(a) Explain and justify the structure of the solution.*

*(b) Describe the parts of the solution using algorithms justifying how these algorithms form a complete solution to the problem.*

*(c) Describe usability features to be included in the solution.*

*(d) Identify key variables / data structures / classes justifying choices and any necessary validation.*

*(e) Identified and justified the test data to be used during the iterative development of the solution.*

**Menu Screen**

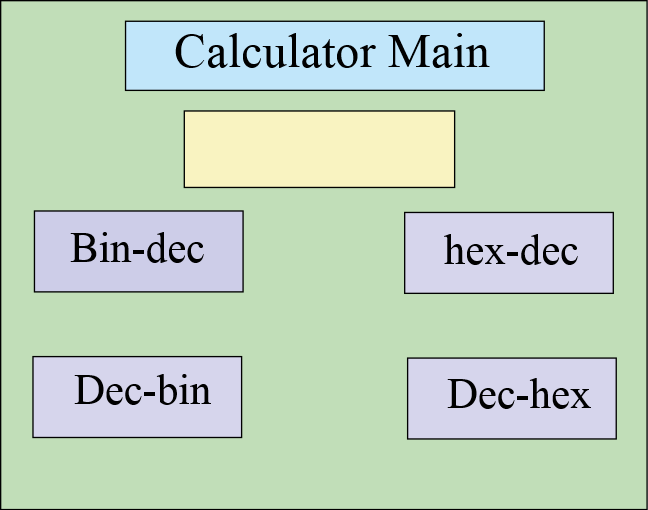
The menu screen will contain two buttons: one which will go to the player browsing screen, and one which goes to the match creation screen.

Explain any connections, dependencies or links between parts of you design e.g. the main page will have a login drawing on data from users table.

Modularise you work e.g. complete these elements for each of your modules (screens)

Using the example above – the calculator

**Calculator – Main Page**

****

The main page will contain the options for my binary calculator. I will have four options that the user is allowed to choose (bin-den, den-bin, hex-den, den-hex) and they can enter the number onto the front page. It will also store their number in a file for later use. There is a title, data entry for number buttons……

Remember to list all of the variable/functions used, their purpose, data types and validtions

**Functions and variable used:**

Var click\_me Boolean 0 or 1 validation – lookup check 0 or 1

Var sci\_bin String (3) – validation – length check (3 chars)

Func\_conv\_bd(number) function that converts binary numbers to denary numbers

**Flow chart to show the process**

START

Bin to den?

Den to bin?

Remember to include routines for the validations

*……………………….etc*

**Pseudo code**

Proc calculator\_main

Open main\_page

If x < 7

Open random\_message(“this must be a binary number”

End if

………………………………….etc

In order to test the functionality of this page, I will test the following items

**Button1**

Should close the main page and open the denary – binary converter

**Button 2**

Should close the main page and open the binary – denary converter

………………………etc

You need to plan the tests that you will use as you are developing this module – it can be quite short – tests should be designed to demonstrate that your system is working – particularly validations and the logic of the module.

**Test Plan for development**

**For var\_number**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test data | Test type | Predicted outcome | Actual outcome | Pass/Fail |
| 010100101 | valid | Saves number |  |  |
| 01010100101001010100 | borderline | Saves number |  |  |
| 000000000000 | boarderline | Saves number |  |  |
| bob | Invalid | Rejects text |  |  |

………..etc

## (iii) DESCRIBE THE APPROACH TO TESTING

1. *Identify the test data to be used during the iterative development and post development phases and justify the choice of this test data.*

This is the overall testing for the system and should be a test plan that proves that you have completed the system and met all of your success criteria. You need to describe your methods of testing – these could include alpha, beta, white box, black box, top down & bottom up. There are several ways in this should be achieved:

1. System test – derive three scenarios that need to be completed, e.g. a customer needs their details entered, they wish to make a purchase and get their receipt. Choose one to be valid, the next to be borderline and the last to have invalid data in it.
2. Ask you user to test your system. Give them two tasks to do so that they can experience the usability – acceptance testing (black box)
3. White box testing – stress test variables & functions to try and expose weaknesses, e.g.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Test data | Test type | Predicted outcome | Actual outcome | Pass/Fail |
| 010100101 | valid | Saves number |  |  |
| 01010100101001010100 | borderline | Saves number |  |  |
| 000000000000 | borderline | Saves number |  |  |
| bob | Invalid | Rejects text |  |  |

# (3) DEVELOPING THE SOLUTION (25 mARKS)

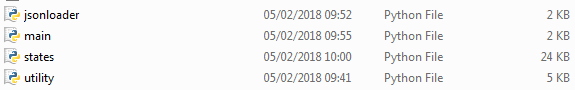
## (i) INTERATIVE DEVELOPMENT PROCESS

*(a) Provide annotated evidence of each stage of the iterative development process justifying any decision made.*

*(b) Provide annotated evidence of prototype solutions justifying any decision made*.

### Overview

Initially I started to make the main loop of the program, and create the State and StateManager classes. I separated my code into 4 files to make it easier to work shown below:



*jsonloader –* will contain code responsible for interacting with the JSON file holding player and team data

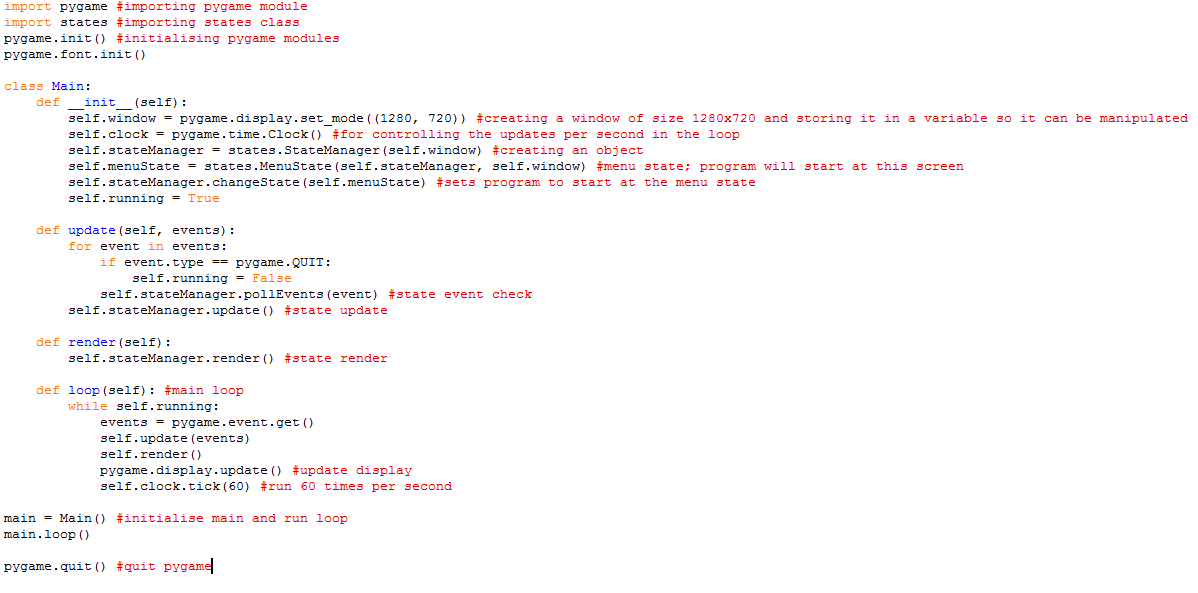
*main –* will contain the main loop of the program, as well as the creation of the window/pygame modules. The entry point of the program will be this file.

*states –* holds the code for all of the states in the program

*utility* – contains all of the helper classes used in each state of the program such as text, text input and buttons

The main loop will be run at a controlled 60 times per second therefore it runs at a same speed on all computers. In the loop it will call the currently running state’s update, render and event check loop.

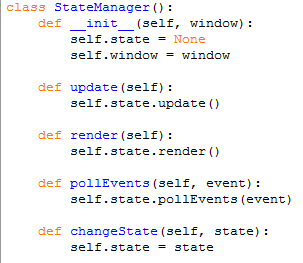
### Main Class

This is the main.py file which is the starting point of the program. This file will remain mostly untouched from now on apart from entering the program in a different state for testing purposes. Firstly a window is created using the pygame function display.set\_mode((x, y)) to make a 1280 x 720 screen. Another inbuilt function is the pygame clock which controls the updates per second, which I will set to 60. I then create a statemanager object which will be explained below, which contains the changeState function allowing me to set the state for when the program launches. Here I set the starting screen to be a menu state.

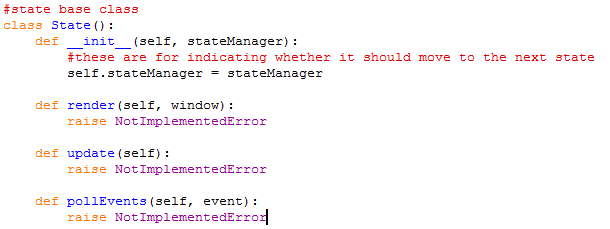
The loop is the main part of how the program; it runs off a simple condition to check if a Boolean, running is true. While running is true, the currently run state is updated, rendered and event checked 60 times per second using the clock.tick(x) function from pygame’s clock class. The last part of the code simply initializes the main class and runs the loop. Once the program is exited pygame quits and the window closes.

### State and StateManager Class

To start to work on the screens for my program, I firstly need to create a State and a StateManager class, as explained in the design section. These classes do not require much code, as the State class is an interface and the StateManager class only requires a few methods.

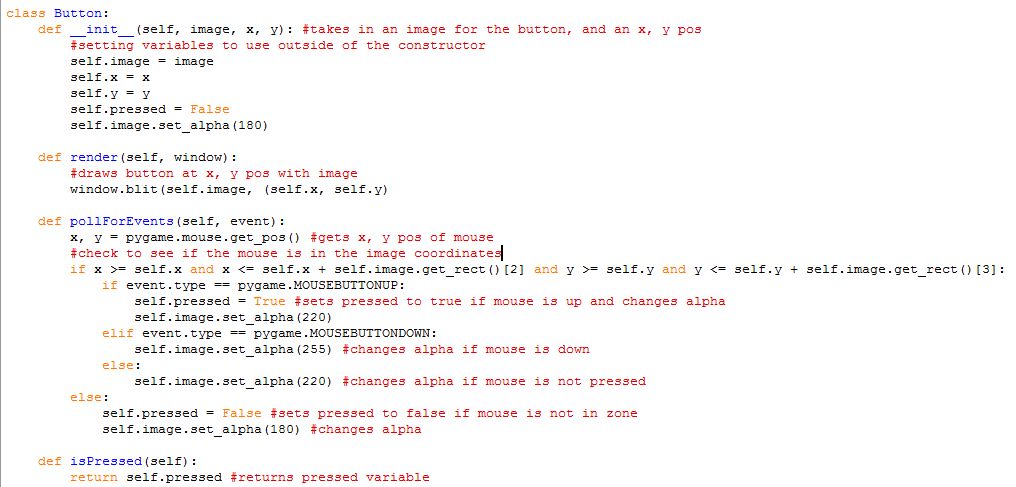
Firstly, the StateManager takes in the window as a parameter. This is so that the states can use this variable to draw to the screen. It then sets a variable, state, to none. This variable holds the current state being updated, rendered and event checked. This class is instantiated in the main file and the procedures are run in the loop shown above in the main file.

The second class needed before starting work on the states is the State class. As explained in the design section, the State class will be a parent class to all states as they will be inherited.



The code above shows the State class. Firstly, it takes in a parameter called stateManager, which is a StateManager object. This is so that when there is a situation where the state needs to be changed, eg if the user clicks on a button to go to the menu screen, then I can call the changeState() method in the stateManager class anywhere in a class which inherits state. The other three methods are the standard rendering, update and event check methods. The render method takes in the window parameter so it can be drawn to. None of these methods do anything, they only return a NotImplementedError, which means that all classes inheriting this one must contain these 3 methods otherwise it will return an error.

### Button and Text Renderer Class

In order to work with text and button interaction in pygame easily, I decided to make classes for buttons and text rendering to make drawing text/creating buttons trivial.

The Button class firstly takes in an image, x and y position. This is so that I can use the window.blit(image, x, y) pygame method to draw an image of a button on the screen at specified coordinates. The Boolean pressed holds the value for if the button is pressed.

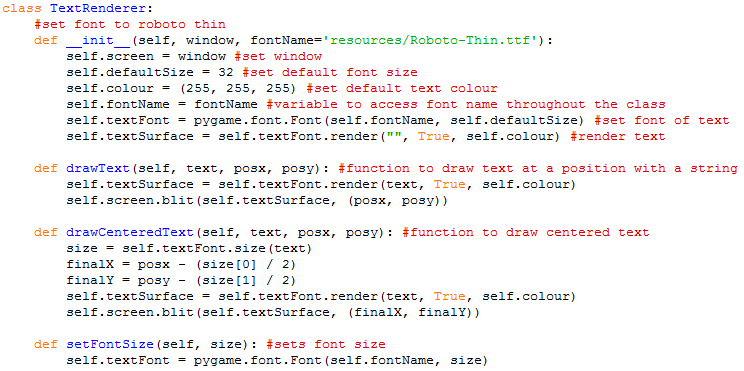
The render method draws the image to the screen using window.blit() from Pygame’s inbuilt functions.

The main logic of the code lies in the event check method. The way it changes the pressed variable value is to check the mouse position. Pygame.mouse.get\_pos() returns a tuple with the x and y coordinates of the mouse on the window. This is where storing the x and y value of the button is important, because I can check if the mouse position is between the x, y value and the image width/height added to the x, y value. To get the image width/height I can use an inbuilt Pygame function image.get\_rect(). This returns an array with size 4, containing the x, y, width and height of the image. Therefore I can index the width/height of the image. If the mouse position lies in the image, it then goes on to check multiple possibilities. The first piece of selection is to check if the mouse has been pressed and is now coming up. If this is the case, then this means the user has already pressed the button and has released it, therefore pressed can be set to true. I also change the alpha value of the image with the choices; this is because it slightly changes the appearance of the image so the user knows if they are hovering/pressing/not pressing the image. The second piece of selection is to check if the mouse is down. If this is true then the alpha simply changes to indicate that the user is pressing down on the button. The last statement changes the alpha to notify the user that they are just hovering over the button; they are not pressing down or up. The last else statement runs if the mouse is not in the image area. This means that the user is not pressing on the button, so pressed is set to false, and the alpha is changed to indicate that they are not hovering over the button.

The last function, isPressed(), returns the pressed boolean in order to check if a button is pressed in a state.

The way the button class will work is that it will first be instantiated in the constructor of the state class to place the button in. I will then call the render function in the state’s render function, and the event check function in the state’s event check function. In the state’s update loop, this is where the button will be checked to see if it is pressed, using the isPressed() function. This is where I can put the code which executes when the button is pressed.

The TextRenderer class will be present in every state as they all require text to be drawn to the window. Below is the text renderer class:

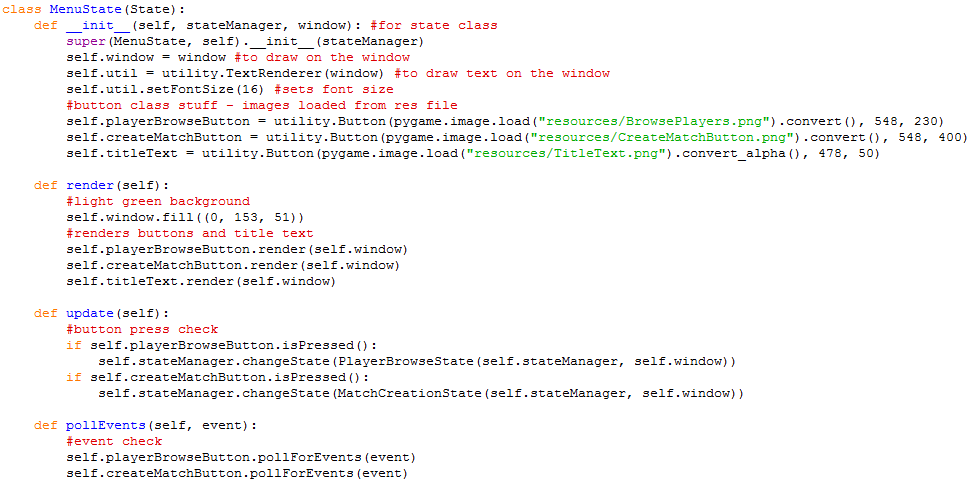


The constructor takes 2 arguments: the window so it can draw text to the screen, and the font location. This is an optional argument as I will be using Roboto thin for most text on the screen. In the constructor I set some standard values such as font size, text colour and font name. I also make a variable called textSurface which is responsible for drawing the text onto the screen. There is nothing to be drawn on creation of the object so it currently contains an empty string.

The function drawText() and drawCenteredText() both achieve similar objectives – they use Pygame’s functions to allow me to draw text easily to the screen without worrying about passing in the window or other unnecessary things, just through a simple custom function. The function drawCenteredText() draws the text at an x, y position relative to the center of the string to be drawn to the screen, whereas the origin of the normal drawText() function is at the top left of the text. I did this to make it easier to position text on the screen when drawing titles etc.

The function setFontSize simply changes the font size of the next string to be drawn to the screen.

### Menu State

The first state I have decided to implement is the menu screen, as it is a fairly simple screen to make and it won’t take long to complete.

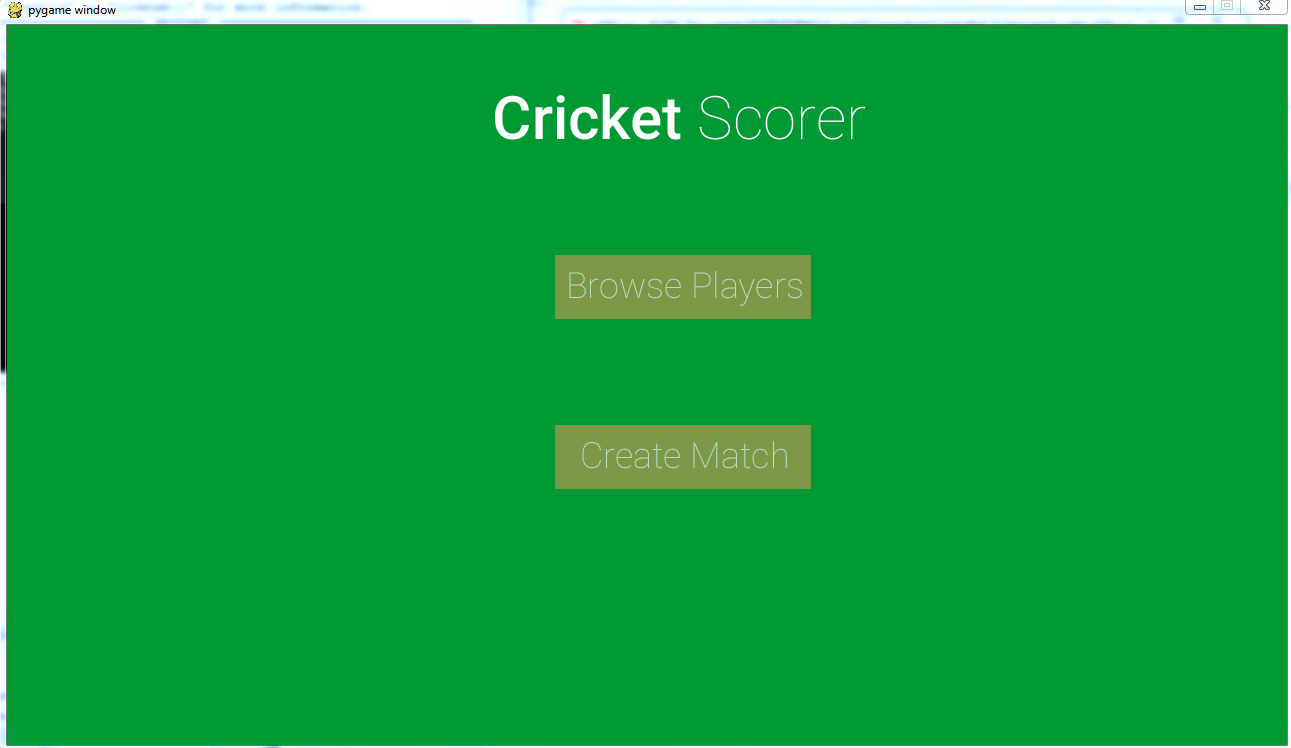
The Menu State inherits the parent class State to make sure all methods are included. The arguments for the constructor are the stateManager and window, so I can draw to the screen and change state inside the class. Util, playerBrowseButton, createMatchButton and titleText are all objects of the Button and TextRenderer class.

It is worth noting that titleText is not supposed to be a button, it is just a logo. However I have used the Button class to allow me to easily display an image to the screen because I have not created a function to draw an image to the screen, and it is faster to just use my Button class rather than using the built in python functions to draw an image.

The render function fills the background with a light green colour as mentioned in my success criteria, and renders the buttons.

The update function demonstrates how the Button class works; it checks to see if the button is pressed using an if statement, and if it is true then it changes state using the StateManager object function.

The event check function calls the button event check functions, so the pressed value of both buttons will turn to true if it is pressed and in the button. This is what the menu screen looks like with the code used above:



Example of Button class

As shown it has two buttons directing the user to the player browse screen or the match creation screen.

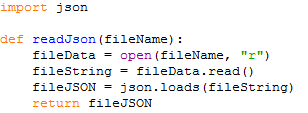
### File Handling

The next section of the program I decided to work on was the interaction of the JSON file with the scorer. In order to create an efficient solution I need to figure out a consistent way of storing the player information in the file. A system was devised to store the players in the following way:



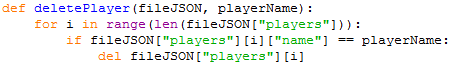
I inserted a few random player names in the file for testing purposes; the user will be able to add players to the system without needing to enter the JSON file. All players stored will be contained in a list, with each separate object in the list containing player information: name, runs scored, age, matches played, team and wickets taken. I can then use the python library json to access and manipulate this data.

The jsonloader file handles everything to do with the interaction and manipulation of this file. As it is not something that needs to be duplicated, I will implement it procedurally rather than using classes as there are not many functions needed to be written.

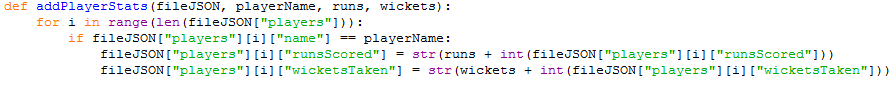
Firstly I import the json library. The readJson function returns a multi-dimensional array containing all of the data read from a filename specified in the parameters. It firstly opens the file, converts it into a string and finally parses the json into an array using json.loads().

This procedure will be used in the add player screen. It is a function to automatically append an object in the json file containing the player’s name, age and team. The matches, runs and wickets are set to 0 as default. It also takes in the json file to add the player to, which will be the json file storing all of the player information. As the json data is a list, it is easy to add to the list of players; I can just use .append().

The next function to add in the jsonloader file is a function to delete a player, as the user should be able to delete a player as specified in the success criteria.

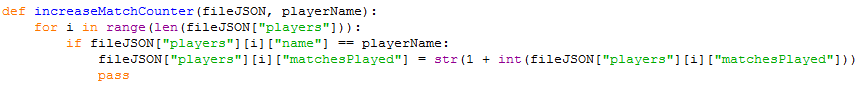


The code above shows the complete function which will delete a player. It takes in 2 parameters; the first is the json file data, and the player name so that it knows which player to delete. The way it finds the player is that it iterates through each object in the ‘players’ array of the json file and if the name matches the specified player name, then that object selected is deleted.

The next function to program is a function to add player stats to the json file. It will be used when the match has finished to save the stats of the match to the json file. It will need to take in the player name, runs scored in the match and wickets taken. Below is the code:

Firstly, similar to the delete player function, it searches through each object in the players array of the json file to find one with the same name as the one specified in the parameter. It then adds the amount of the runs and wickets specified in the parameter to the respective player’s runsScored and wicketsTaken value in the json file.

The next function to create is to increase the matches played value for all the players involved in a match. In order for this to work, it will be called iteratively at the match ending screen on all of the players involved in the match. The function will take in a player name. Below is the finished code:

The code searches the json file for the player by iterating through each one until the name matches the specified player name. When it is found, the matchesPlayed value is incremented by 1.

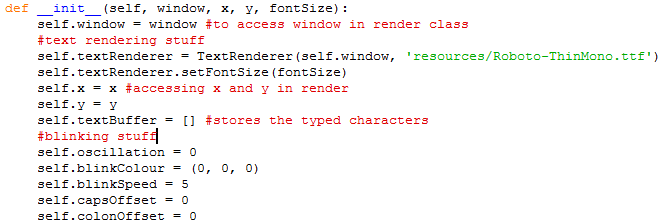
The last function to write in the jsonloader file is one which saves the file. Below is the code:



This function simply overwrites the json file containing the player data with the updated json data variable used in all functions.

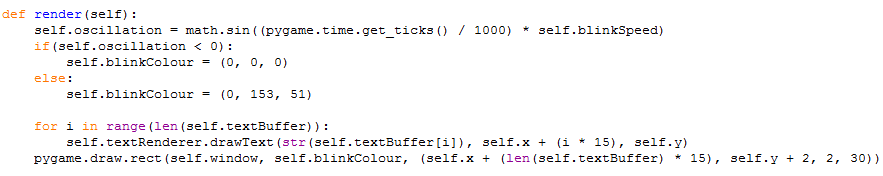
Most of these functions will be used in my player statistics screen, which will be created next. Whilst writing this part of the code, I realized that if the json file stored large amounts of players, then the browse players screen would be filled with a lot of text, which means that some of it would overflow. Therefore I decided that I would make a search function instead, where the user can type in a name of the player and they will be directed to their statistics screen.

In order to make a search system, I need to make a TextInput class which will be responsible for keyboard input, and holding the string that the user has typed. It will take in 4 parameters: the window to be drawn to, the x and y pos, and the font size. Below is the code from the constructor of the TextInput class:

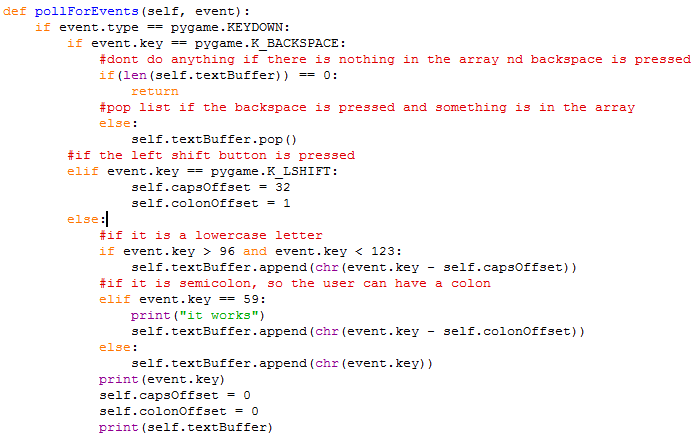


Firstly, I initialize some necessary variables such as the window for use in the render method, and the text rendering class so that the user can see the text typed on the screen. There is another variable, an array, called textBuffer which will store the characters typed on the keyboard, and there will be a function which converts the array of characters into a string so that the string typed is retrievable.



The function above converts the text buffer into a string by using python’s .join() function.

The render function will be responsible for drawing the text typed onto the screen at coordinates specified. It is also used for drawing a blinking cursor to let the user know where they are typing. The blinking cursor works by using a sine function on the time elapsed which constantly changes, therefore a value will oscillate between -1 and 1. Then it simply changed colour based on the value; if it is less than 0, then it turns black and if it is greater, then it turns the background colour, which is a light green.

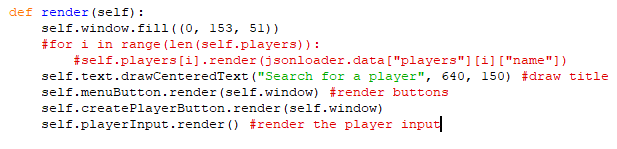
The main part of the code will take place in the event check function, as this is where code is executed when a key is pressed.

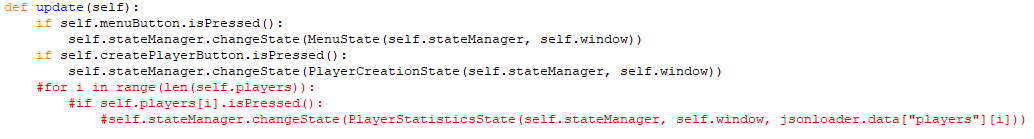
The way the input works is that I get the ascii value of the key pressed, using event.key from Pygame’s inbuilt function, and use chr() to convert it into a character which is then appended to the textBuffer list. To allow capital letters I had to make an offset to the ascii value, as the capital letters are 32 apart from each other. When the shift button/caps button is pressed, then this offset is applied.

Before creating the player statistics screen, I need to make a player search screen with the text input created above.

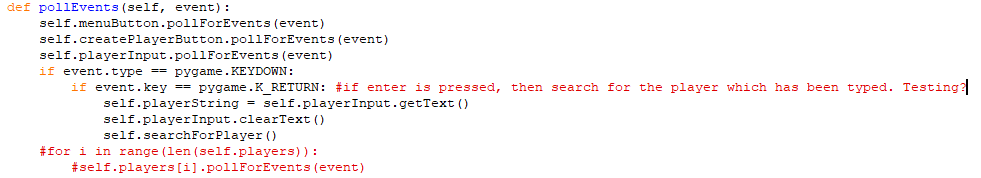
### Player Browse/Statistics screen

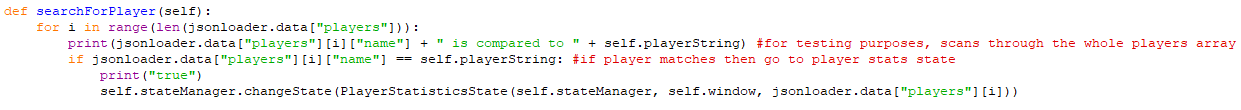
As usual the player search state class needs to inherit state, and pass in a window/statemanager in the parameters for it to work. I also create a textrenderer object, and two buttons which navigate to the menu and the create player screen. This is also where the TextInput class is used. The code commented out is the code written previously, before I decided to make a search system, where it displays each player stored in the JSON file on the screen.



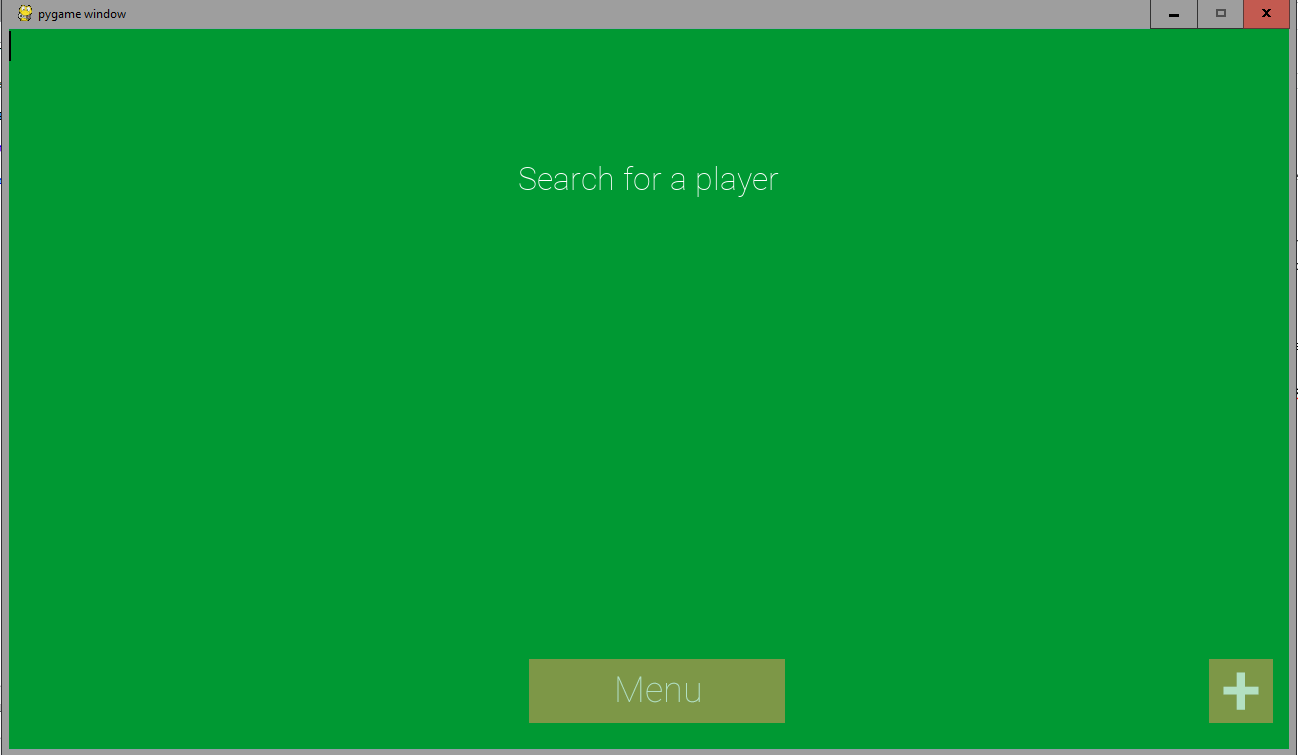
The next part of the class is the render function. This is a fairly simple function; it just draws the buttons, text and text input to the screen.

The update function is also fairly trivial. It runs to check if either the menu or the create player button is pressed, and if it is, then it calls the statemanager function to change state to another class. Before making this a search system, I treated all the player text like buttons, where they can be clicked, which is why the commented out code is written.



The pollevents function firstly calls the button and text input function. The next part checks if the user has pressed enter and checks to see if the player typed in the text input is in the json file. This is done by using Python’s event.key variable, which holds the keycode of the keys being pressed. If the key pressed is the return key, then it calls the function in textinput to get the string being typed (which has been explained in the textinput class section) and is stored in a variable called playerString. A function in the PlayerBrowse class called searchforplayer gets called which searches for the player in a separate function. I did this in order to modularize the code a bit, as it would start to look a bit messy if everything was done in the pollforevents function. The text is also cleared using a clearText function I added to the textinput class. The searchforplayer function is shown below:

It firstly loops through all the players in the json file using the jsonloader.data array, and compares it to the self.playerString variable. If they match, then the statemanager changeState function is called to navigate to the player statistics state of the selected player. Currently the statistics state is empty so it can’t be shown, however the player browse state is shown below:



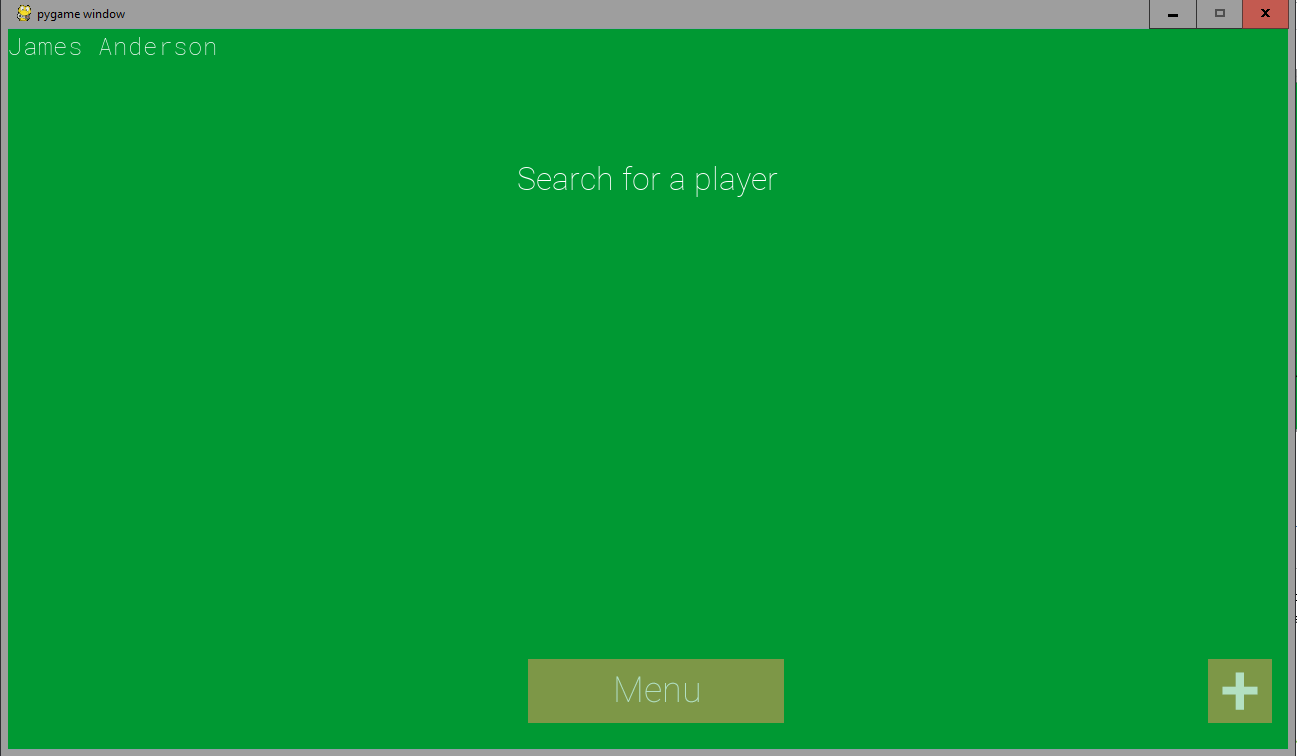
playerInput(TextInput object)

createPlayerButton

menuButton

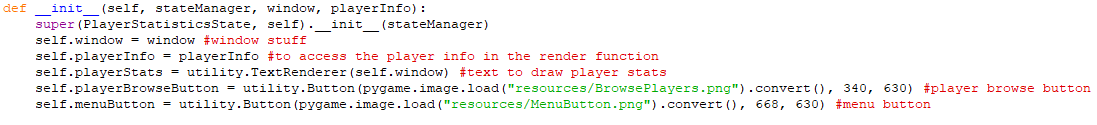
TextRenderer text

When player has typed:



playerInput(TextInput object)

As this text input was not originally planned, I did not write a testing plan in the design section for this state. However after writing the player statistics state I will be testing the text input too as I feel it is a suitable one to carry out.

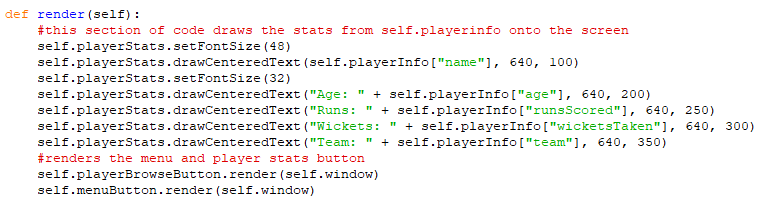
The player statistics state will be a fairly small class as it is just taking in a player and parsing their stats from the JSON file onto the screen. It will contain age, runs, wickets and their team. Below is the init function:

The class takes in an additional parameter – which is an array holding the player stats. In the player browse class’s function searchForPlayer I showed how it works:

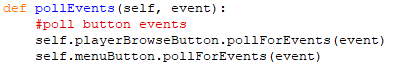


It passes the jsonloader.data value of the player typed onto the screen. This means that it will hold four variables: age, runsScored, wicketsTaken and team.

The init firstly assigns the window variable so we can draw to the screen. Then, the playerInfo variable is passed on to the class so I can use it in other functions such as render, where it will be required. I then create a textrenderer object, and two buttons, which will navigate to the player browse/menu screen when pressed. The next part of the class, the render function, is shown below:

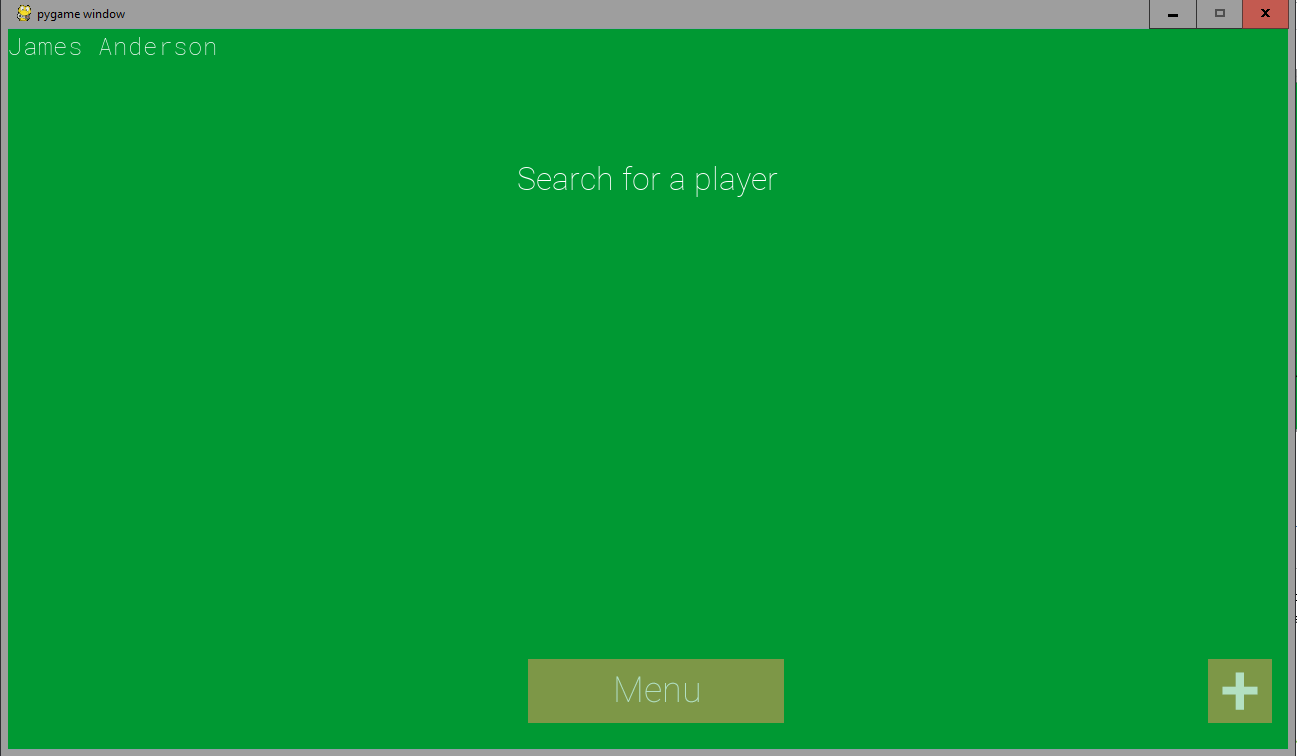


This class draws the stats onto the screen. It firstly draws the player name, accessed by the playerInfo array, in a slightly larger font to act as a title. Then, the age, runs, wickets and team are shown. The last part of the function is where the two buttons, to the player browse and menu screen, are rendered.



The last function, pollEvents, just calls an event check on the two buttons which are being rendered onto the screen.

To demonstrate how all of this works, I will annotate screenshots from the player browse to the statistics screen:



TextRenderer text

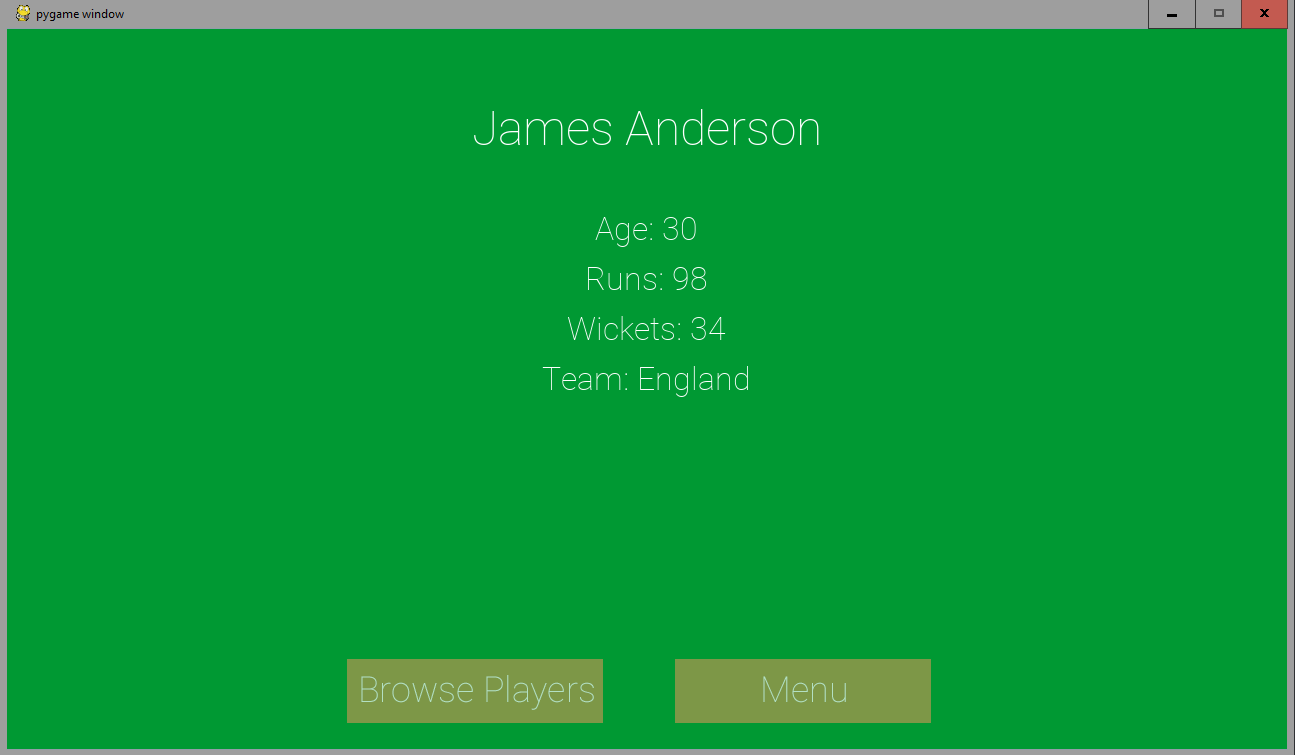
menuButton

createPlayerButton

playerInput(TextInput object)

When enter is pressed, it will pass this part of the json file to the player statistics state class:





menuButton

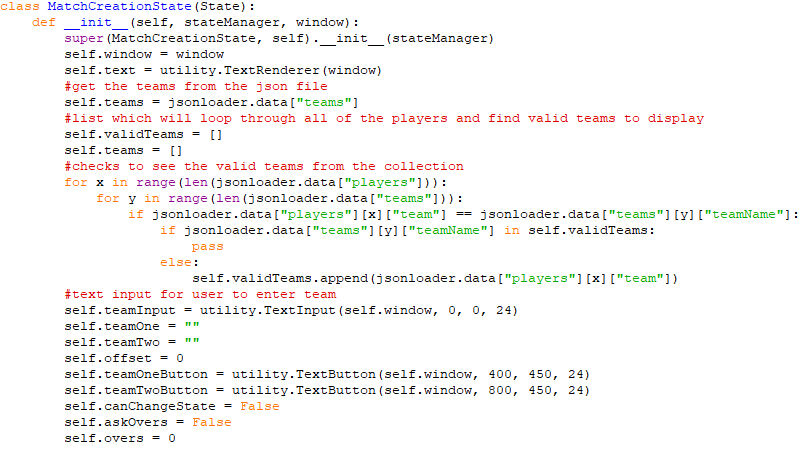
playerBrowseButton

Values taken from json file as shown above

TextRenderer text

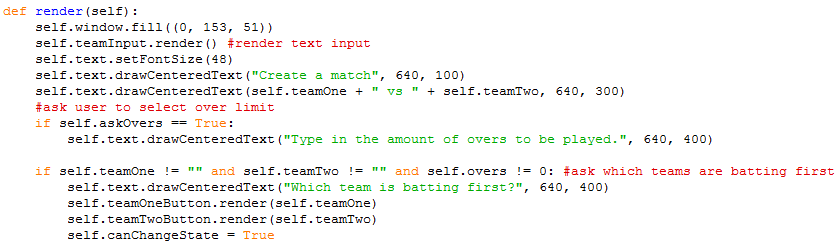
### Match Creation State

This is the screen before the match state where the user will be able to select the teams involved in the match. Similar to the browse player screen I changed my approach to the user typing in the team rather than showing a list of teams, as there may be a lot of teams stored on the JSON file, and it may flow off the screen. The init function of the match creation state class is shown below:



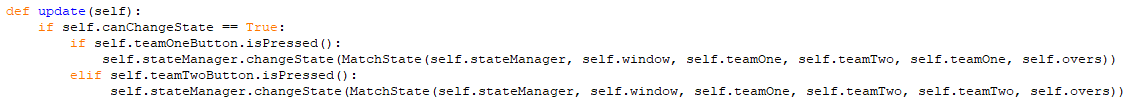
This function initializes the variables to be used in the class. Firstly I make a variable holding the window and the text renderer. I then get the list of teams by retrieving them from the json data file. In order to minimize the amount of teams to be searched I decided to make a validTeams variable, which runs a loop through the players in the json file and only adds teams to the list which actually involve players in them. I also create an instance of the text input class so that the user can search for the respective teams. Some other essential variables are also created, explained below:

* teamOne, teamTwo – strings which hold the name of the teams which will be selected
* offset – for selecting with arrow keys however is obsolete now with the new approach to selecting teams
* teamOneButton, teamTwoButton – for selecting which team is batting first
* canChangeState, askOvers – for asking the user about overs, explained in more detail below
* overs – variable to set the amount of overs. Will be passed as a parameter in the MatchState class.

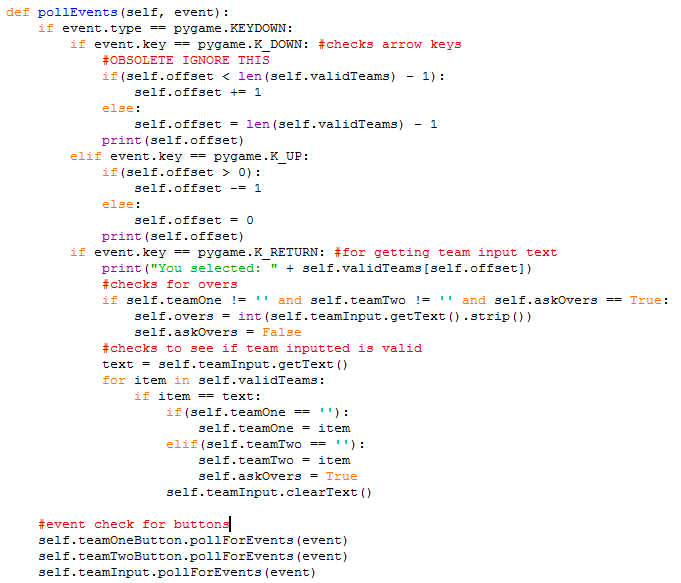


The render function will draw the teams and buttons onto the screen. It firstly asks the user to type in two teams, which will be added to the teamOne and teamTwo strings. Afterwards, askOvers is set to true which then allows the user to enter the amount of overs they want to play. This value will then be read from the text input and set to overs.

When the overs have been set, and the teamOne/teamTwo variable are not empty, the user is asked which team are batting first. This allows the user to click on the respective team, and then the canChangeState variable is set to true, which allows the program to change to the MatchState.



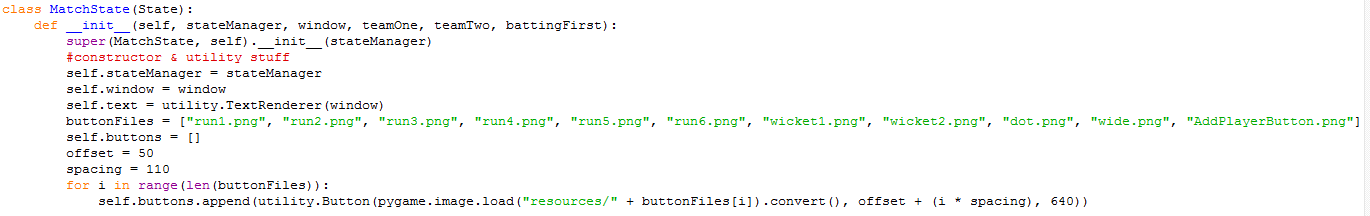
The update function checks to see if the team one/two buttons have been pressed, and if so then the state manager changes state to the match state, configuring which team is batting first by setting the teamOne and teamTwo variables as parameters for the MatchState class.



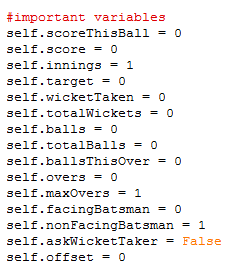
This is the event check function. It will process the text input that the user types, which are the two teams and the amount of overs to be played. The first part of the code is not needed anymore, as it was the old method of listing all of the teams and selecting them with arrow keys. The way it processes the text input is that it checks if the enter key has been pressed. The string of the team typed is then stored into a variable named text, which is compared to the validTeams list created in the init function, which contains all teams in the JSON file with players in them. It then firstly fills the teamOne variable with the team name. To fill the second name, it checks if the teamOne variable is not empty and if it is true then teamTwo is equal to text. When enter is pressed, the text is also cleared so that the user knows the program has processed the text.

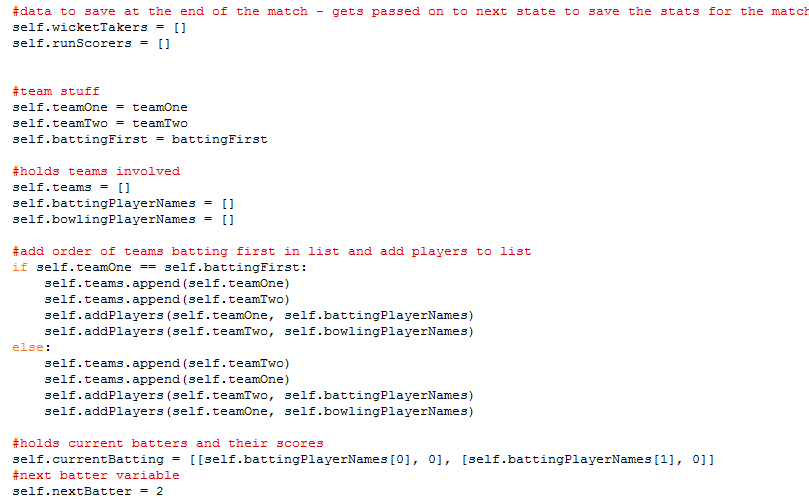
After teamTwo is filled, askOvers is set to true which allows the text from the render function to ask the user the amount of overs to be played. When the user has entered a suitable number, it is converted to an int and the overs variable initialized in the init function is set to this value so that it can be passed in the MatchState parameter. As the overs variable has been set to something other than 0, and the teamOne/teamTwo variables are not empty, the user can now select which batsman is batting first, and they can switch to the MatchState screen.

### Match State

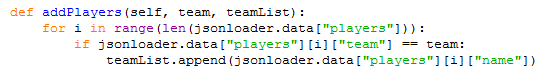
This will be the main state of the program, where the user can score a cricket match. The constructor will need to take in several inputs; the two teams which are playing, and some sort of way in identifying which team is batting first. It will also take in the usual state manager and window instances.

This piece of code is the first part of the constructor. It does the usual creation of state manager and window variables, as well as a text variable. The next variable holds an array of all button files, in order, to be rendered on the screen. I have done this because for a screen using a lot of images it is more efficient to store them in an array and initialise them in a for loop, which is what I do below the array. The offset and spacing variables are for adjusting how far apart each other the buttons are and how far off from the left it should start.

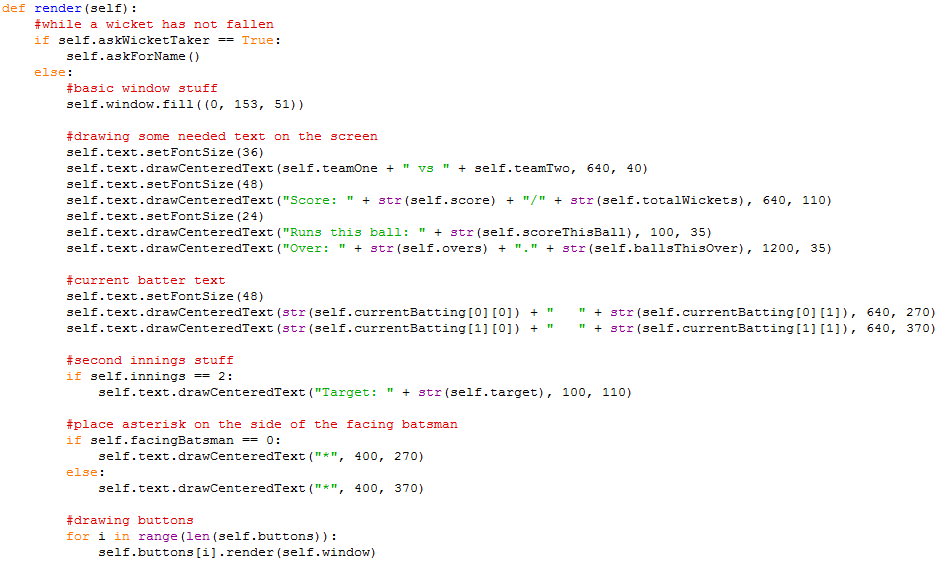
The next part of the init function is a collection of variables needed for the scoring. These were mostly included in the design section however a few have been added which were not thought of before, such as askWicketTaker.



The two arrays above are going to be an array storing the players who have taken a wicket/scored runs. In the match ending state these variables will be processed and saved to the json file holding player statistics. The following 3 variables are for accessing which team is batting first in other functions. Team one and team two are also stored because when there is an innings change, the team currently batting will need to swap. The team, battingPlayerNames and bowlingPlayerNames are all for drawing information to the screen which will be shown in the render function. Self.teams is appended in the following part of code, which appends the two teams in order of batting first. I also made a function to quickly add players to a specified list, called addPlayers. I also create a 2d array called currentBatting, which stores the name of the player currently batting as well as their score. At the start this is the first two elements in the battingPlayerNames list and both are set to 0. There is also a nextBatter variable as a pointer to the next item in the battingPlayerNames list.



As explained above, I created a quick small function to help out with adding player names from the json file to a specified list. It simply searches the json file for a team name matching the one specified as an argument, and appends it to a list also specified as an argument.

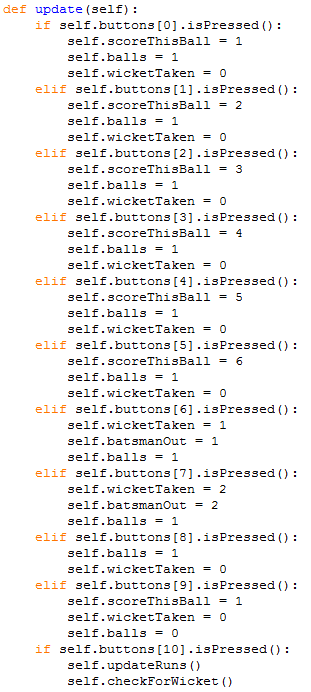
This render function runs off an if loop, as when a wicket falls, the screen will change and it will ask the user which player took the wicket. This screen is small so it doesn’t need a class of its own; I have made a function called askForName which will handle asking the user which player got the wicket.

The main scoring screen shows a number of information:

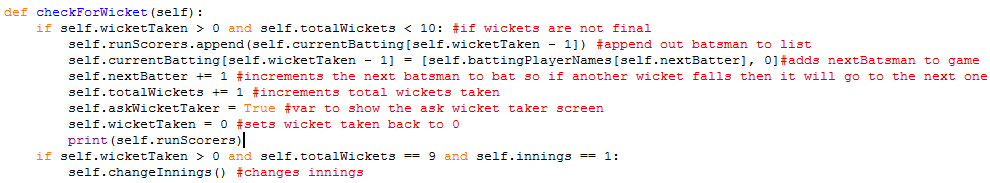
* Teams involved
* Score
* Wickets
* Overs
* Target (if it is the second innings)
* Current batsmen
* Buttons for updating the score

In order for the user to know which batsman is facing, an asterisk will be drawn next to their name, which is done commonly in professional digital scorecards. This is done by a simple check to see which batsman is facing using self.facingBatsman, then drawing an asterisk at a position just to the right of the name. The buttons are also drawn using a for loop as they are all stored in an array, as discussed above.

The second innings compromises of the batting team chasing a target in order to win the game. Therefore it should be possible to see the target on the screen. This is done by getting the target variable and only displaying it on the screen if it is the second innings.

The code on the left is the update function in the match class. It is mostly just checking for button input through accessing each one from an array. As explained in the design section, the user will have to click a button corresponding to what has happened in the game(e.g if there has been a run, then the user will press the ‘1’ button), then press a button which updates the total runs and moves onto the next ball.

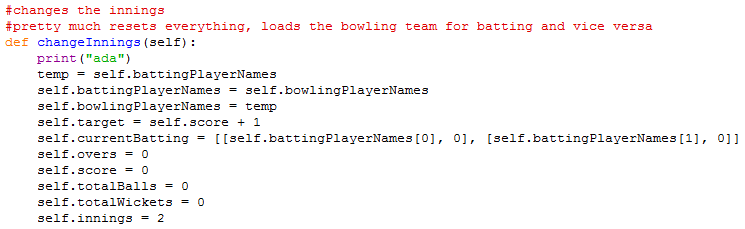
There are 11 buttons on this screen. The first 6 elements in the array are for runs; if these buttons are pressed then the value of balls changes to 1, as this means a ball has been bowled, and scorethisball changes to the value of the runs on the button. Wickettaken is also set to 0, as this variable is for the index of the batsman which is out, which are separate buttons. The next two elements in the array hold the wicket buttons. The user will press the first wicket button if the dismissed batsman’s name is displayed first, whereas if the second batsman is out, then the next button should be pressed. The 9th button will be used when no runs have been scored. The 10th button is for an extra ball, where there have been runs scored on a wide, byes, no ball etc. The last button is the one where the user will press to update the runs/wickets and move onto the next ball. Two functions are run, one which updates the runs/wickets and one which checks for a wicket. Whilst writing this I originally wrote it in the update function however it became too large therefore I decided to modularize it into separate functions.



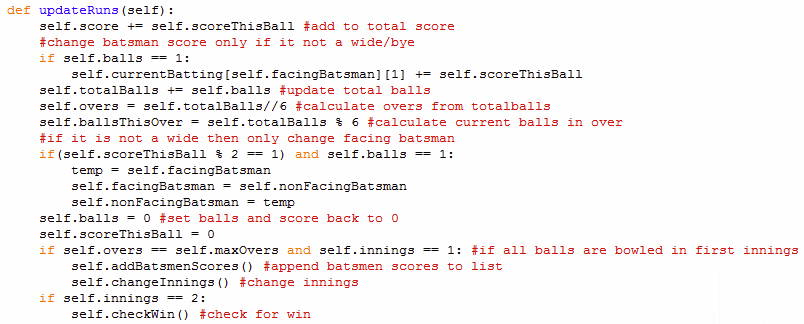
There are many things to take into account when a wicket has fallen; whether it is the last wicket, which innings it is, and which batsman is facing next. The program also needs to know which player got the wicket, so it can be updated on the stats json file. The first if statement checks to see if it is not the last wicket. If the condition is true, then it finds the index of the batsman which has been dismissed, from the variable wicketTaken, set on the wicket button pressed, explained above in the update function and adds this to the runScorers list. The runScorers list is for the updating of stats on the json file. It then increments nextBatter and totalWickets.

The next Boolean, askWicketTaker, is set to true. This Boolean will change the screen temporarily and ask the user which player got the wicket. I have not added this function, but it is checked in the render method, and I will create a sepearate function called askForName to update the wicketTakers list.

If the wickets reach 10 and it is the first innings, then a function called changeInnings is called, which will be explained below.



The changeinnings function resets most of the variables shown to the user such as overs, score and wickets. It also sets the target score variable defined in the init function, which will be the check to determine the outcome of the game. I then swap the battingPlayerNames with bowlingPlayerNames to update the batsmen/bowlers, as an innings change is where the batting team swap with the bowling team.

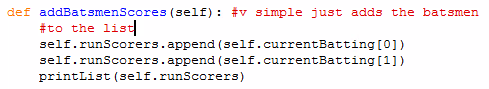
Going back to the update function, the second function called when the last button pressed is updateRuns, shown below:

The updateRuns functions handles the possibilities of what could happen when the ball has been bowled. Firstly, it updates the score by incrementing the current score of the ball to the total score. It then changes the batsmen’s score if ball is equal to 1, because if ball is 0 then this means an extra has been bowled, such as a no ball using the extras button. Extras do not count towards the batsmen’s score. The total balls bowled then gets updated. In cricket scorers a common way to display the amount of balls bowled is to have an over counter. This is where the number of overs already bowled are displayed followed by a full stop, then the amount of balls currently bowled in the over. For example, if it was the 2nd ball of the 5th over, then the over counter would be: 4.1. This is because 4 overs and one ball have already been bowled. The way I implemented this into my program is to use two equations determining the overs bowled and the balls bowled. The first equation, is totalballs//6 stored in a variable called overs. This returns an integer which is the amount of times 6 goes into totalballs. For example if totalballs was equal to 4, then overs would be equal to 0, which is correct as no overs have been bowled yet. The second equation is totalballs%6 which is assigned to the variable ballsThisOver. This equation grabs the remainder of totalballs if it was divided by 6, therefore counting the number of balls bowled in the over.

The next part of the code checks to see if the facing batsmen should change. If there are an odd number of runs scored, such as 1 or 3, then this means the batsman have swapped ends. Therefore an easy way to test if the batsmen facing should switch is to check if the current score this ball is odd. If it is odd, then the facing and non facing batsman variables are swapped.

The next two lines of code change the balls and current score this ball back to 0 in preparation for the next ball to be bowled.

The final part of the code is to check if all overs have been bowled whilst also being the first innings. If it is true then changeInnings is called and a small function called addBatsmenScores is called.



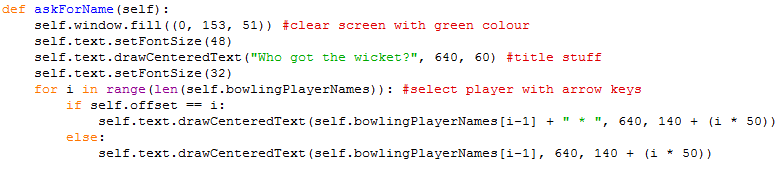
This function just adds the current batters names and scores to the runScorers list for later use.

Finally the last part of updateRuns checks to see if is the second innings. If it is then it runs a function which checks to see if the game has finished.

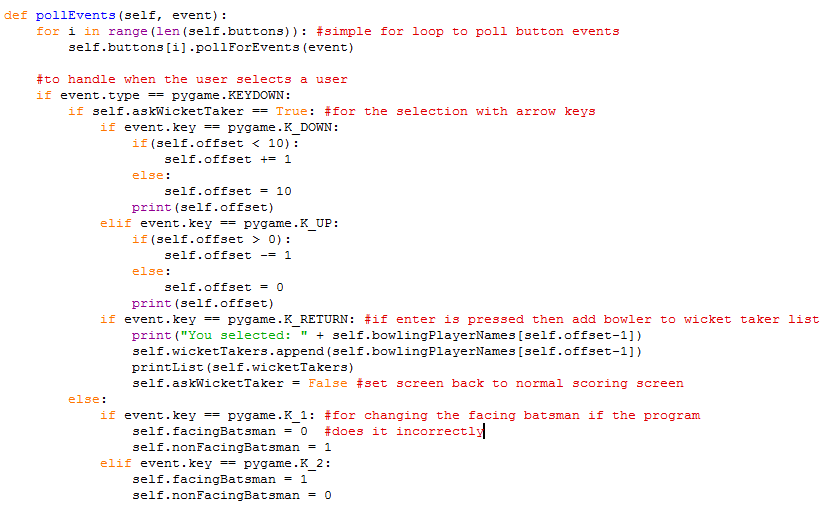
#### Annotate code here

The function checkWin is run every ball in the second innings. This is because it needs to check to see if there has been a potential outcome to the game. The first if statement checks if all the overs have been bowled and if it is the second innings. This then branches out into 3 possibilities – a win for the batting team, a win for the bowling team, or a tie. The first if statement is to check for a tie. As the target is one more than the first innings score, the variable score is compared to target – 1. If it is true then the statemanager is called to switch to a new small state, called MatchWinState, which will be explained after this section. It takes in 3 extra parameters, the winning team, the wicket takers and the run scorers. This is where the wicketTakers and runScorers list is used, to pass the statistics to the next state in order for it to parse the information into the JSON file. The last if statement in the first branch checks to see if the team batting second has won. If they have, then the winning team parameter from the MatchWinState class will be the team batting second.

The second if statement checks to see if the team batting second has won before the overs have finished. This is a possibility to consider as they can score the runs before the over limit has been reached. If it is true then addBatsmenScores is run to add the currently batting players to the batting player list. Afterwards the statemanager changes state to the MatchWinState.



This function will be responsible for asking the user which player got the wicket. It firstly clears the screen with a green colour, and then draws some text asking the user who got the wicket. The user then selects the list of players rendered on to the screen. They are rendered by using the bowlingPlayerNames list and rendering each name iteratively. The players are selected with the arrow keys, and the currently selected one will have an asterisk by their name.

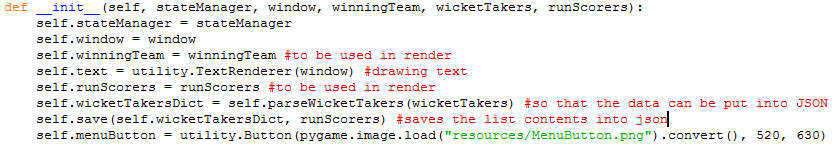


The last method in the MatchState class is the event check. Firstly it polls the button events for each one on the screen by iteratively calling pollForEvents on each element in the button list. The next part of the code handles the key pressing. Firstly, it checks if askWicketTaker is true – this part of the code is for selecting the player who got a wicket on the askWicketTaker screen. It is done by incrementing the offset value each time the down arrow is pressed, and decrementing for the up arrow. This offset value is then the pointer for accessing the bowling player names in the array, which is done when the enter key is pressed; wicketTakers appends this string value which is the bowler’s name. The askWicketTaker Boolean is then set back to false so that the screen returns to the scoring state.

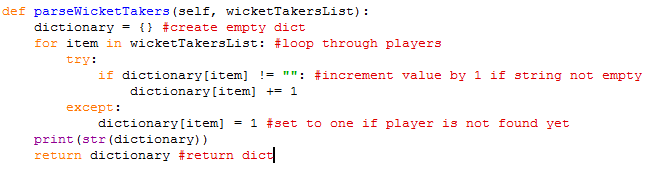
The last part of the pollEvents method is to allow the user to change which batsman is facing. This feature has been added if the automatic facing does not work properly. If the user presses 1 on their keyboard, then the facing batsman is the first name whereas if they press 2 then the facing batsmen is the second name.

### Match Finish State

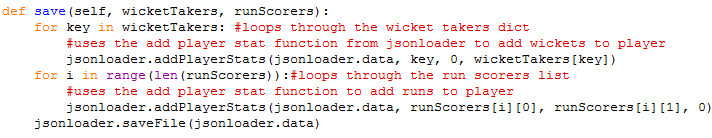
The last state to be created is the screen after the match has been finished. It will be a visually simple screen, as it will display the stats of the match. The stats will be the wicket takers and run scorers; it will also display whoever has won the match on the screen so the user knows which team has won. The code below is the init function of the match finish class:



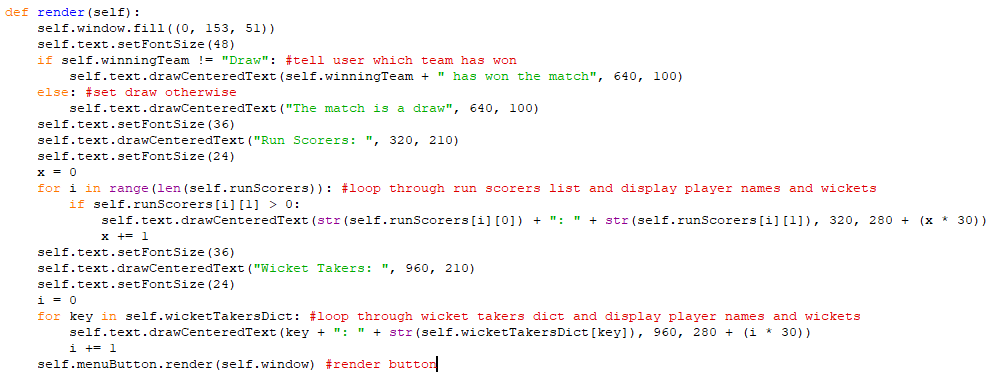
The class takes in a few extra parameters: the winning team, the list of wicket takers and the list of run scorers. Firstly, the statemanager and window are set as variables as usual. I also create a winning team variable as that needs to be accessed in other functions. A runscorers variable is also created in order to parse these values into the json data file. The run scorers is a 2d list, as for each element, it contains the name of the player, and the amount of runs. Therefore to parse this data to the json file it should be fairly simple by indexing. However, the wicket takers list will be slightly harder. This is because each time a player takes a wicket, their name is added to the list, leading to duplicates. In order to make it easier to count the amount of wickets each player has, I decided to make a separate dictionary, where the key is the players name, and the value is the amount of wickets that the play has. I have created a function which returns a dictionary to handle this called parseWicketTakers. I create a variable called wicketTakersDict which holds the dictionary of the wicket takers. The next function, save, writes the wicket taker and run scorer values from the dictionary and list to the json file, which will be shortly explained below. I also create a menu button so that the user can navigate back to the menu when they have finished with the screen.



The parseWicketTakers function takes in a list, which is the wicket takers list passed through as a parameter from the match state class. Firstly an empty dictionary is created. It is then filled by iterating through the wicket takers list and adding the strings as a key. If the value of the key is already 1, then the value is incremented by 1. The final part of the code is to return the dictionary, so that a variable can be assigned to it, which is what happens in the init method.

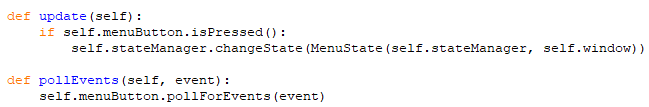


The save method takes in the wicket takers and run scorers list. It then makes use of the jsonloader function addPlayerStats which increments an integer to either the wickets taken or runs scored to the specified player in the json file. The last line of code makes sure the json file is saved after changes are made to it, so that if the user exits the program, the stats will still be saved.



The render function tells the user the team that has won, and also renders the menu button/displays the stats. It firstly checks to see if the winningTeam variable is not equal to draw, which is passed through as a parameter in the constructor. By using the winningTeam variable it displays to the user which team has won. If the match is a draw, then the text just says that it has been drawn.

The next part of the code lists the run scorers and wicket takers. This is achieved by iterating through the list/dictionary and drawing each player who has at least one wicket/run. The final part of the code renders the menu button.



The last two functions are very small; the update function is only needed to check if the menu button is pressed. If it is then the statemanager changes state to the MenuState class.

The pollEvents just runs an event check on the menu button so that the user is able to click on the button.

## (ii) TESTING TO INFORM DEVELOPMENT

*(a) Provide annotated evidence for testing at each stage justifying the reason for the test.*

*(b) Provide annotated evidence of any remedial actions taken justifying the decision made.*

# (4) EVALUATION (20 mARKS)

## (I) TESTING TO INFORM EVALUATION

*(a) Provide annotated evidence of testing the solution of robustness at the end of the development process.*

*(b) Provide annotated evidence of usability testing (user feedback).*

## (II) SUCCESS OF THE SOLUTION

*(a) Use the test evidence from the development and post development process to evaluate the solution against the success criteria from the analysis.*

## (III) DESCRIBE THE FINAL PRODUCT

*(a) Provide annotated evidence of the usability features from the design, commenting on their effectiveness.*

## (IV) MAINTENANCE AND DEVELOPMENT

(*a) Discuss the maintainability of the solution.*

*(b) Discuss potential further development of the solution.*

## APPENDIX A - BIBLIOGRAPHY