Useful Tips For Your Quidditch

# Write your basic classes with basic attributes first.

Field (based on Maze) and Seeker (based on Rat) objects would be a good place to start. Use the Rat Maze sample to see how to place your tiles to make the complete game scene. **You do not require to make the full classes at this point,** just the basic stuff to get you started.

# Make good use of the free book!

If you want to do something specific, like make a button, there will be an example somewhere in the book. Or on the Internet, because Python is very much the Language Of The Day and there are literally millions of websites out there with code examples.

# Now create your Main Thread.

Use plenty of comments and **always, always, always** make a **new version** **every time** you go back to work on your masterpiece. That way, you can always roll back if something goes horribly wrong.

# Amend your design as you go.

As you add extra attributes and methods to your classes, go back and fill in your Class Diagram to match.

# Use images wherever possible.

Pygame includes various methods for drawing all sorts of shapes (see the chapter *Memory Puzzle)* and you can use these if you like, but it’s probably a lot of extra work you don’t need to do. Find or make some appropriate pictures for your game, and make sure that they are all an easy size to multiply and divide – eg 100 x 100. Try to avoid pictures of different sizes, or odd sizes. Make good use of the transparency option so that you can have odd-shaped items that are still actually square.

# Enhancements.

If you want fancy bits like a splash screen or game over screen, do these **at the end** once you have got the main bits working.

# Reminder of the Client Brief.

Weasleys’ Wizard Wheezes have commissioned you to design and develop a digital version of Quidditch. Quidditch is a popular sport played in the Wizarding World; you can read all of the objectives and rules here - <https://en.wikipedia.org/wiki/Quidditch>

The objective of Quidditch, as with most sports, is to be the team that has gained the most points by the end of the match. Matches are played between two opposing teams of seven players riding flying broomsticks; using four balls: a Quaffle, two Bludgers, and a Golden Snitch. Centred around the use of each ball, there are four positions: the Chasers and Keeper (who play with the Quaffle), the Beaters (who play with the Bludgers), and the Seekers (who play with the Golden Snitch). Each team has three Chasers, one Keeper, two Beaters, and one Seeker. Matches are played on a large oval pitch with three ring-shaped goals of different heights on each side. It is an extremely rough but very popular [semi-contact](https://en.wikipedia.org/wiki/Semi-contact) sport, and has a fervent fan following in the Wizarding World.

Your game will be a prototype consisting of a Quidditch field surrounded by an invisible magical barrier (you can have a square pitch to start off, to make it simpler). Two human players will play as Seekers of opposing teams, who vie to be the first to collect the Golden Snitch. This immediately awards the catcher’s team 150 points and proceeds to the next level, or ends the game if the players are already on the last level.

The game will require at least two levels. Each level should be harder to solve than the previous one, either by having obstacles or a more complex starting arrangement for the pitch, or a combination of both. The program should record how many moves the players take to solve a level (hint: give each player a starting number of points, and deduct one for each move). This information should be output visually.

# System Requirements.

You **must** use an **aggregation** (whole-part) relationship – this will be covered if you make your room map from tiles.

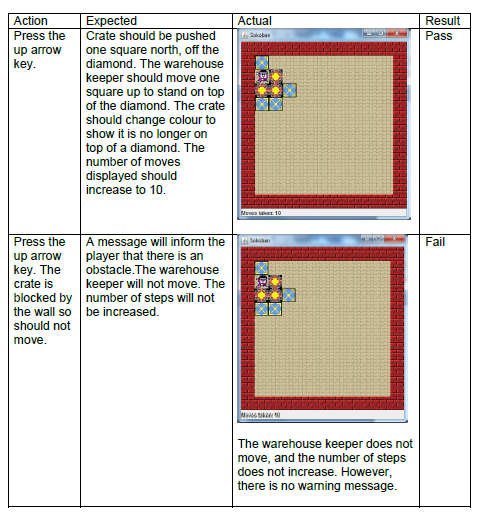
You **must** have an **inherited** class in it somewhere. This should be easy as you are using different sorts of tiles.

You **must** include a **polymorphic** method. In the *RatMaze,* there is a *moveDown()* method in both the *Rat* class and in the *Main* class. Although they are both called *moveDown()* they do completely different things. This counts as a polymorphic method.

Testing.

# Test and log as you go. I cannot emphasise this enough!

Here is an example snippet of an acceptable test plan and log. You **must** include screengrabs. You are looking at about 30 test cases **as an absolute minimum** in order to cover everything.



# List of unit Requirements.

These will be published as a Rubric on Gradebook, but in the meantime, here is what you have to do as a checklist. I have added some notes to the SQA requirements in red text.

## **Stage 1 — Implementation**

You are required to implement the program in an object oriented manner. The solution must demonstrate each of the following:

Design requirements:

 abstraction, encapsulation and information hiding used where appropriate

 inheritance used

 polymorphism used

 all class-wide variables are private to prevent content coupling

 class-wide variables are kept to a minimum to ensure a minimum of common coupling

 data coupling is used (parameter passing) in preference to content or common coupling

 program does not contain a lot of unnecessary data coupling

 classes are highly cohesive

## Implementation requirements:

 a working solution which meets the requirements of the given brief

 variables are correctly declared and initialised

 arithmetic and/or logical operators are used correctly

 a range of control structures are implemented correctly (ifs and loops)

 at least two data structures are implemented correctly (lists, dictionaries or tuples will all count, as does your map if it is set up as a matrix)

 the program contains a minimum of four classes, which contain attributes, methods and a constructor method (your Top Level and *Main* thread **do not** count as a class!)

 a minimum of three objects are created from the classes, with appropriate initial attribute values set through the constructor methods (this is a given as your program won’t work without this!)

 the program contains at least one overloaded method (this may be a constructor method and the one in your inherited class would count)

 classes are linked appropriately through association, aggregation or inheritance relationships

 parameters are passed correctly both within and between objects

 appropriate access types are defined for methods, attributes and classes

 use of pre-defined classes and/or methods from the standard object library (any of the Pygame imports counts for this)

 the program appropriately handles errors with exceptions or pre-validation

 the program code is commented appropriately throughout

## **Stage 2 — Testing**

After completing Stage 1 you are required to develop a test plan and test the completed program. The test log should identify any areas where the program fails, and detail any fixes and retests required.