7 Basics

7.1 RoboCup Environment

7.2 Code Structure

The WitsFc Codebase is built using **python** and contains many files and folders. Although this can be overwhelming you only really need to consider a few particular ones. The sections below outline those particular files.

The execution flow works as follows:

1. `start.sh` is executed using `./start.sh`
2. This script will call `Run_Player.py` for each of your players on the field in its own thread.
3. `Run_Player.py` will instantiate an `Agent.py` object
4. `Run_Player.py` will then call `player.think_and_send()` inside an infinite loop until the process is terminated
    1. Inside `player.think_and_send()` the `strategyData` object will be initialised, we then handle the cases where the player needs to stand up. Finally `select_skill` is called.
    2. Inside `select_skill` we choose between two actions `move` and `kickTarget`. This decision making process will be up to you and what you implement.
    3. Lastly this information is sent to the server and cycle repeats from step 4 above.

7.3 Changing Teamname

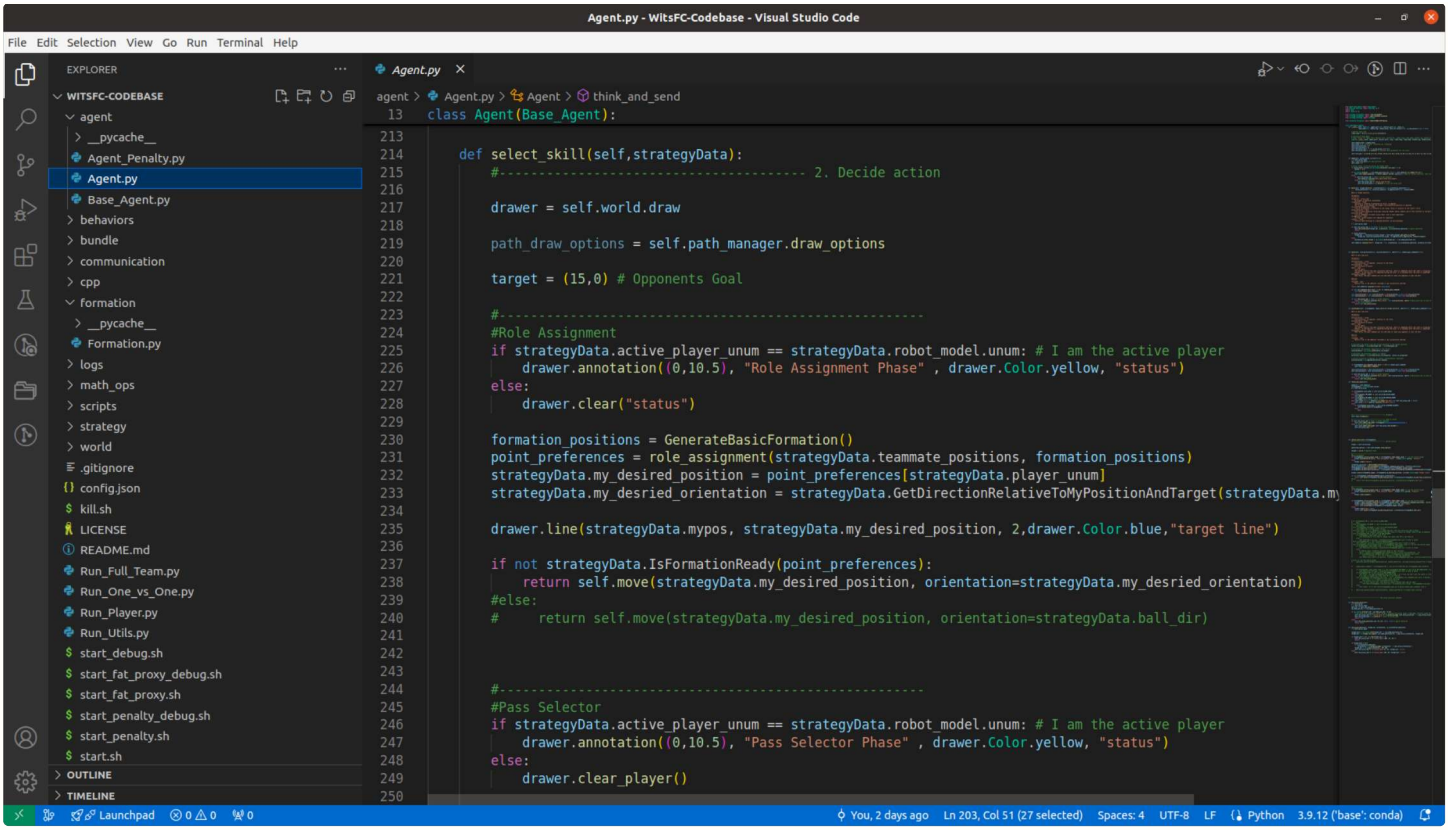
First steps should be to change the teamname that displays when the code runs. Note there is a limited number of characters allowed for this. In order to change the name you will need to update the `StudentName` variable to your name in the following files:

- `config.json`
- `start_debug.sh`
- `start.sh`

The following sections detail different class files within the codebase as well as why they are important.

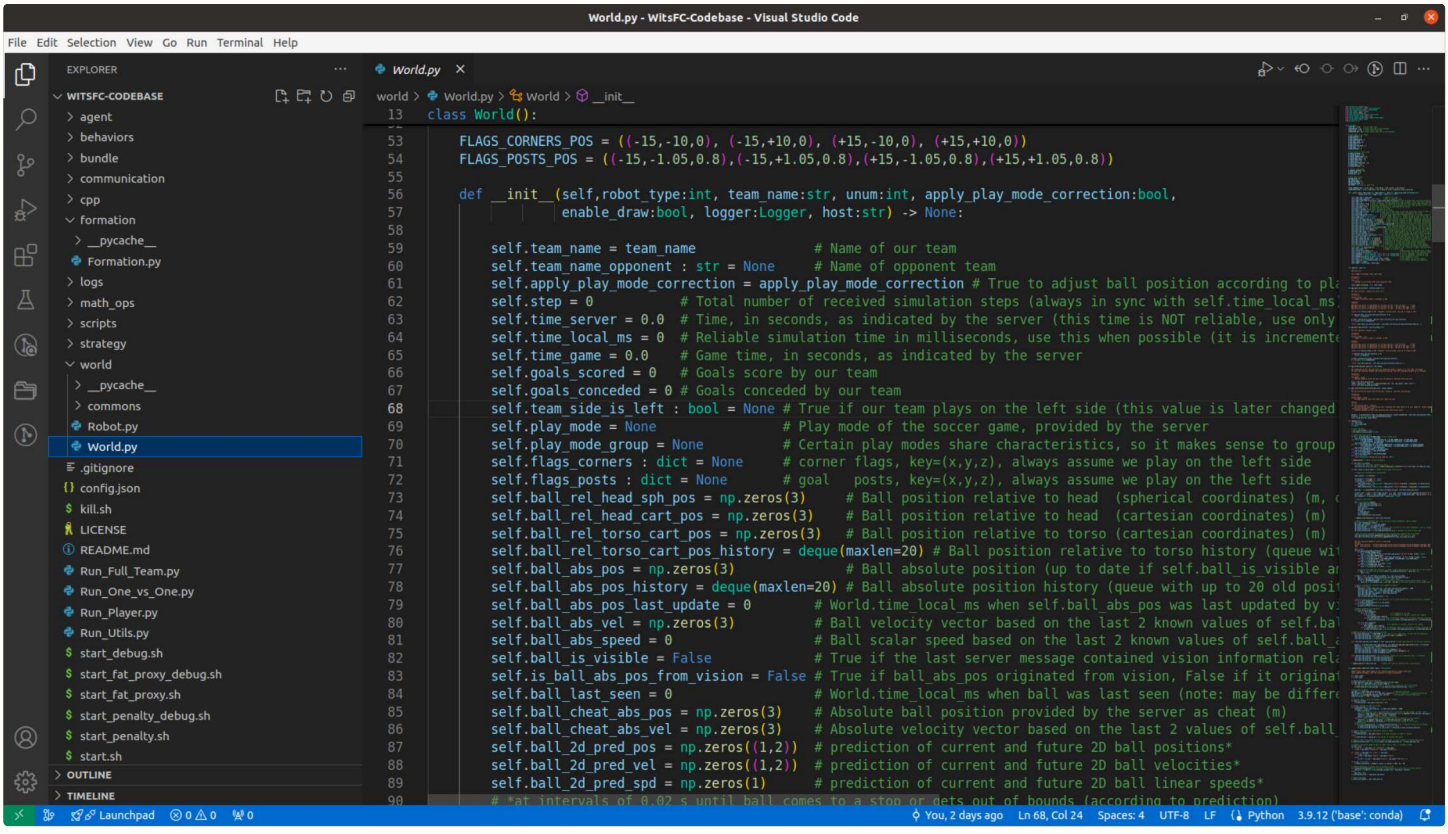
7.3.1 Agent.py

This file found in `WitsFcCodebase/agent/` contains the primary function in which you will be working, this being `select_skill` as well as the implementations of the `move` and `kickTarget` action, although these should not be altered.



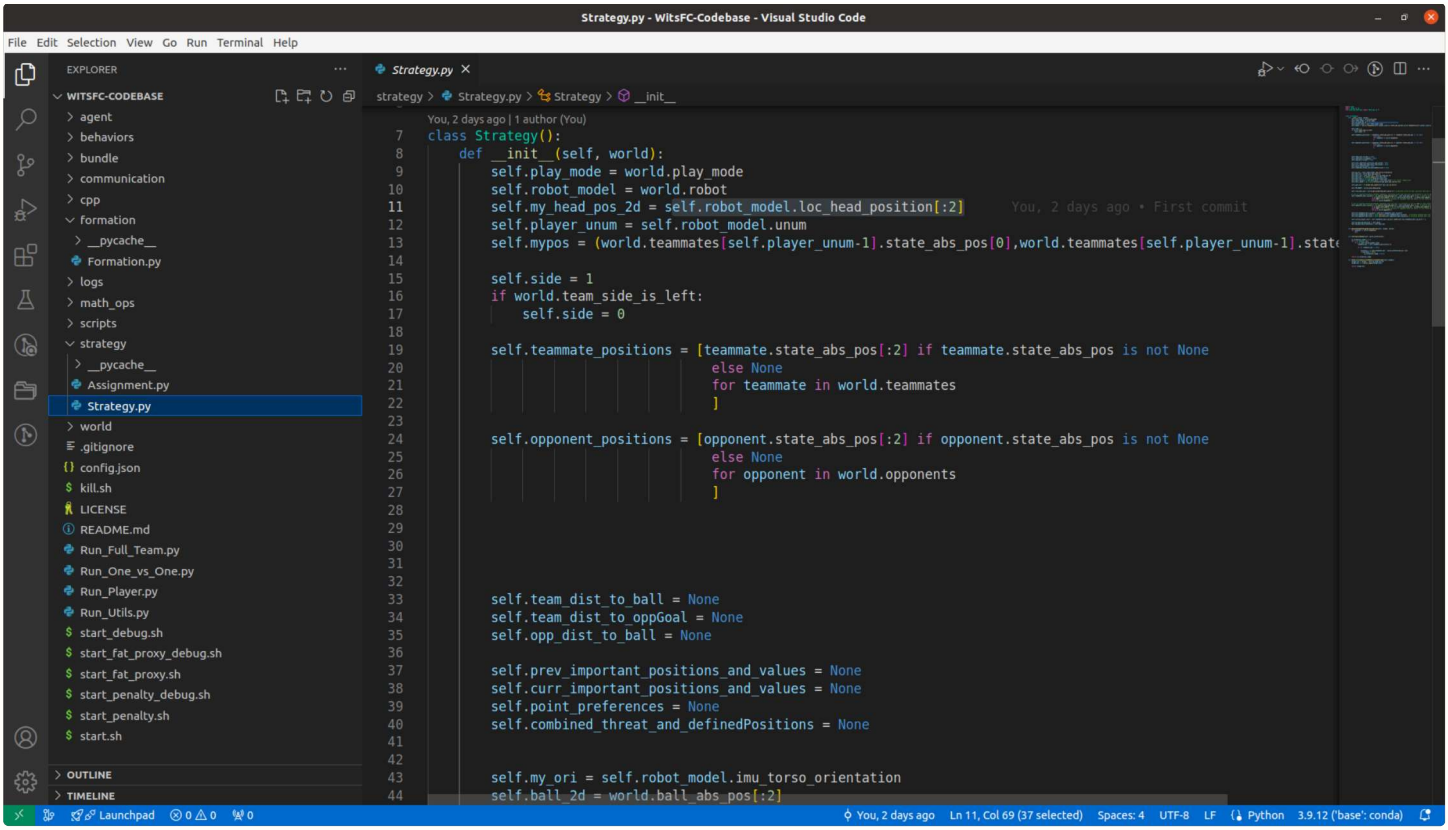
7.3.2 World.py

This file found in `WitsFcCodebase/world/` contains all the information regarding the current state of the game. This includes information like the current score and game mode as well as the information about each individual player on the field.



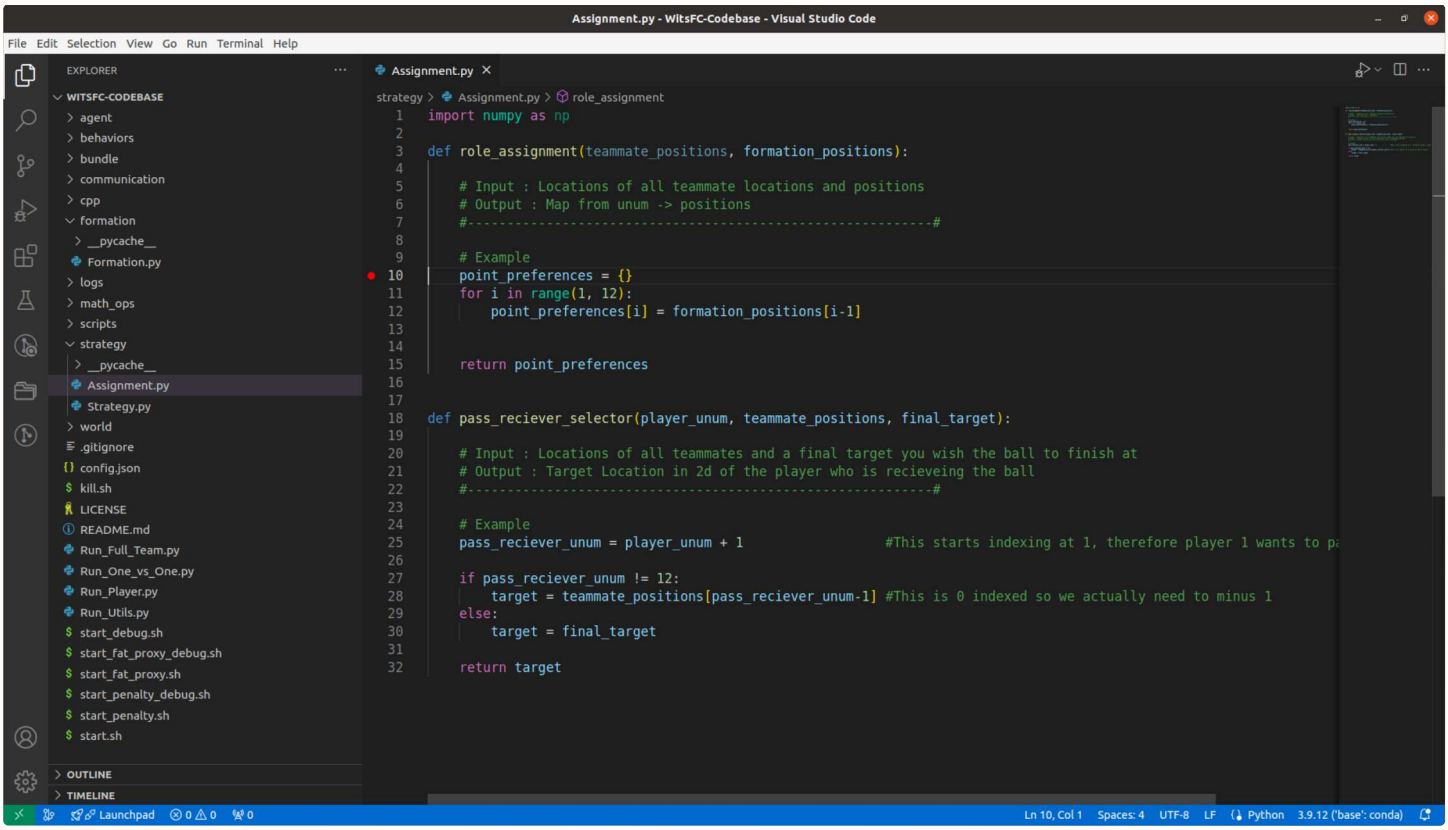
7.3.3 Strategy.py

This file found in `WitsFcCodebase/strategy/` contains the implementation of the strategy class which contains useful datastructures for decision making such as a list of (x,y) coordinates called `teammate_positions` which correspond to the positions of all your teammates. This class can be considered a subset of the `World` class with extra variables useful for designing an intelligent soccer team. This class also has some useful methods and is where you should implement all your helper functions.



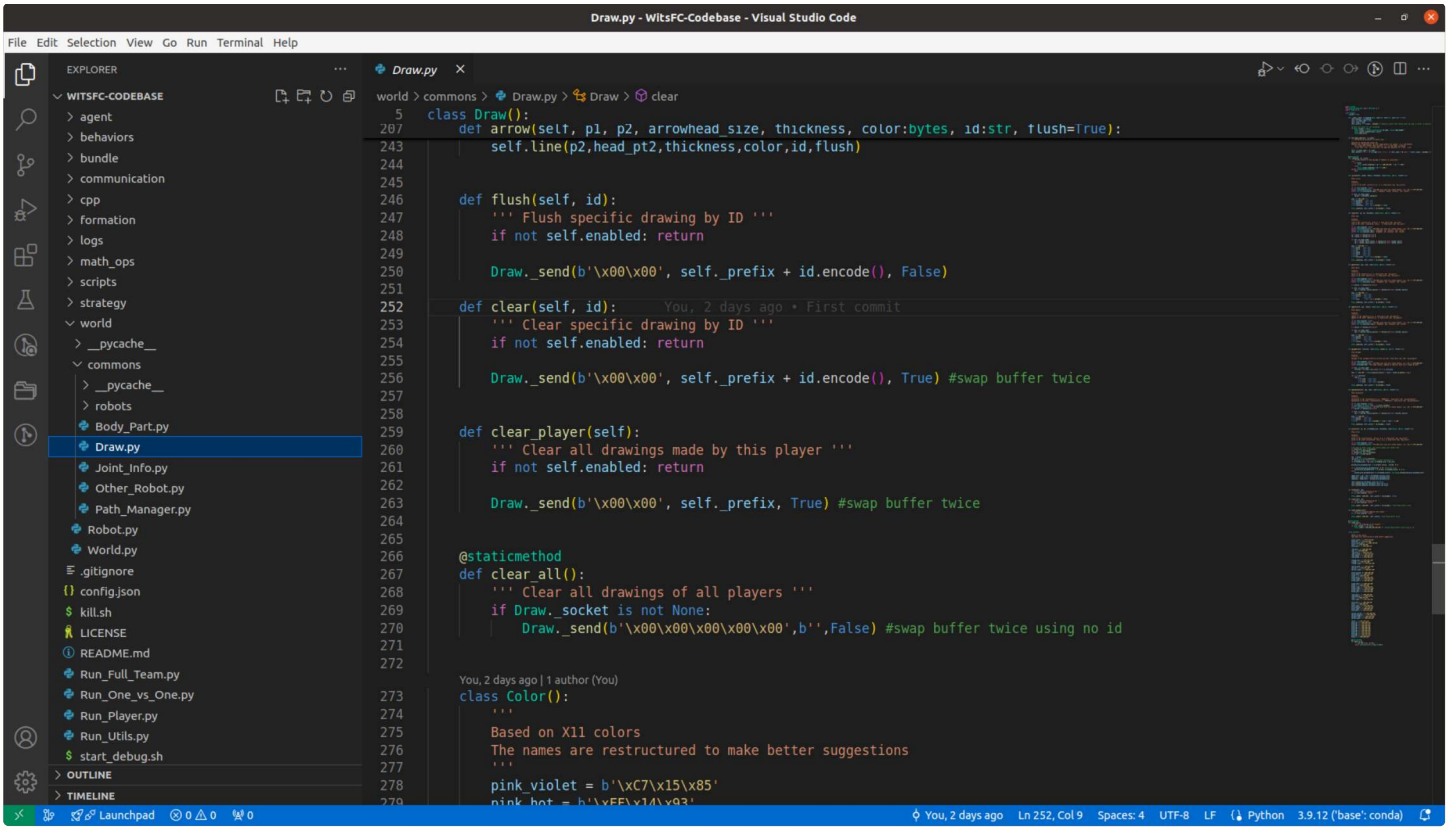
7.3.4 Assignment.py

This file found in `WitsFcCodebase/strategy/` contains the function prototypes that you will need to implement for Submission 1.



7.3.5 Draw.py

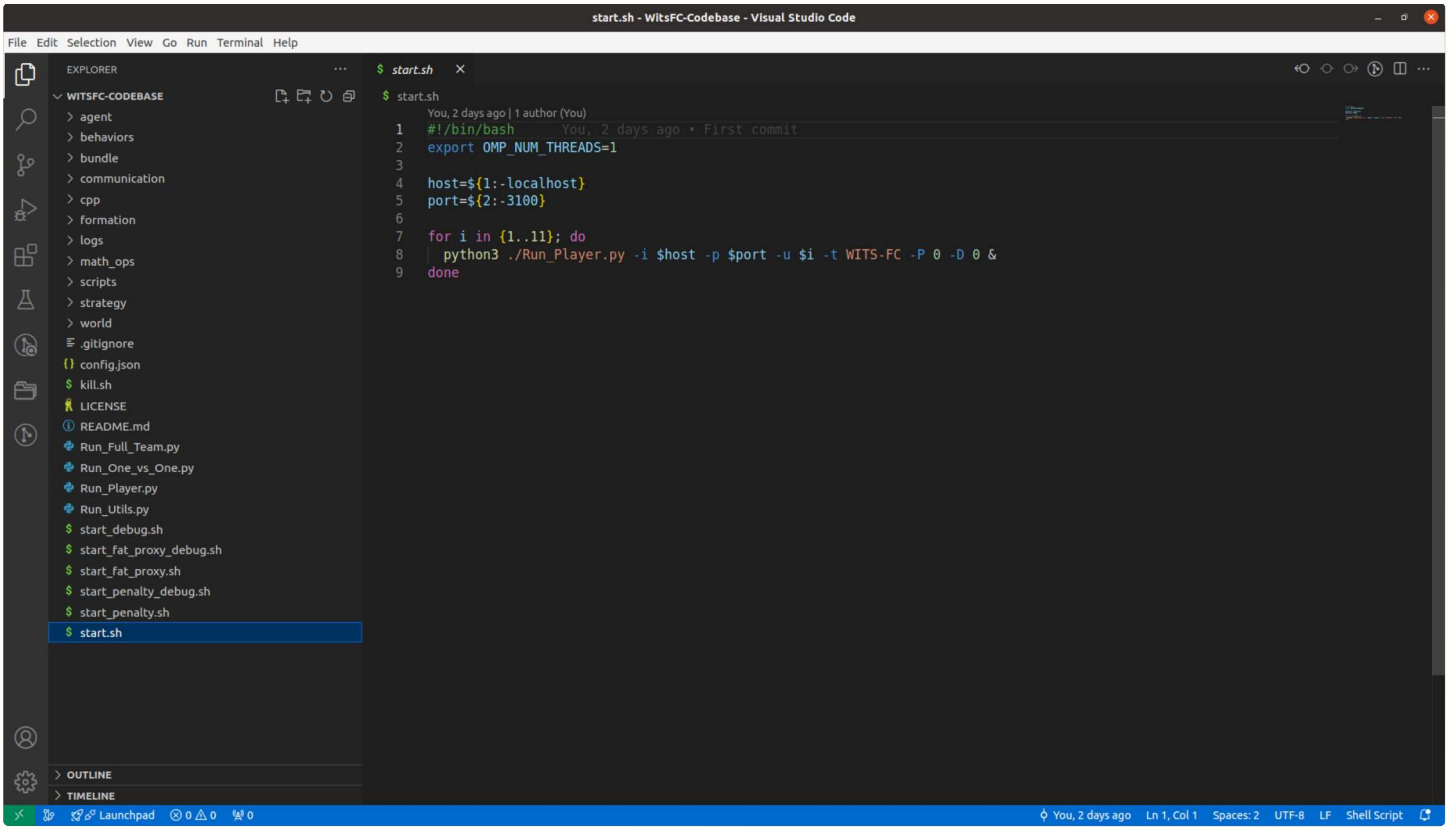
It is very useful to use drawn visualisation to depict the behaviour in which you are trying to implement. This file found in `WitsFcCodebase/world/commons/` contains the functions which you can call inside your code to draw objects like `text`, `lines`, `circles` ect.



7.3.6 Run Scripts

In order to run your team whether for testing or when we evaluate them in the tournament we utilise run scripts. There are a number of these provided which can be used for different things

- `start.sh` - This is the default script that can be used to run a single team using `./start.sh` in the terminal.
- `start_debug.sh` - This performs the same operation as `start.sh`, however, it will allow drawings to be rendered. This if you want to the drawings from `draw.py` you need to run this script using `./start_debug.sh`
- `start_debug_teammates.sh` - This launches 4 players and should be run in conjunction to the debugging process described in the debugging section.
- `Run_Player.py` - You can also run an individual player directly by running `python Run_Player.py` in the terminal. This can be useful for debugging purposes.
- `Run_Two_Teams.py` - This functions the same as `Run_Player.py` but for 5 players and for both teams. However, this just runs your current team as the opponent. **However be adviced each of these python process will be run on the same thread and will have major performance issues.**
- `Run_Full_Team.py` - This functions the same as `Run_Player.py` but for 5 players. **However be adviced each of these python process will be run on the same thread and will have major performance issues.**



7.4 Primitive Actions

Below describes, **the only two actions** you have available to use, which will need to be returned from the `select_skill` function. Meaning at some point in your `select_skill` function you must either return `move()` or `kickTarget()`. Failing to do so will result in undesired behaviours such as crashing or falling over.

7.4.1 move

The move action allows a player to walk towards a given target. This is handled by calling the `move` function given a `target_2d` which is an array of values. For example `(10,0)` if we called `move((10,2))` the agent would move to coordinate `x=10` and `y=2`. This is basic usage of the function, however, it can be expanded to include more components such as the ones shown below:

```
def move(self, target_2d=(0,0), orientation=None, is_orientation_absolute=True,
         avoid_obstacles=True, priority_unums=[], is_aggressive=False, timeout=3000):
    '''
    Walk to target position

    Parameters
    ----------
    target_2d : array_like
        2D target in absolute coordinates
    orientation : float
        absolute or relative orientation of torso, in degrees
        set to None to go towards the target (is_orientation_absolute is ignored)
    is_orientation_absolute : bool
        True if orientation is relative to the field, False if relative to the robot's torso
    avoid_obstacles : bool
        True to avoid obstacles using path planning (maybe reduce timeout arg if this function is called multiple
    priority_unums : list
        list of teammates to avoid (since their role is more important)
    is_aggressive : bool
        if True, safety margins are reduced for opponents
    timeout : float
        restrict path planning to a maximum duration (in microseconds)
    '''
```

You will see in the template code provided that there are a few calls to the `move` function found in the `select_skill`.

```
if not strategyData.IsFormationReady(point_preferences):
    return self.move(strategyData.my_desired_position, orientation=strategyData.my_desried_orientation)
```

In the above example we tell the player to walk to the location assigned to `my_desired_position` while maintaining an orientation facing `my_desired_orientation`.

7.4.2 kickTarget

The second action relates to kicking the ball. This is performed by making a call to the `KickTarget` function. This function takes as input the `strategyData` object, a 2D array corresponding to `mypos_2d` as well as 2D array corresponding to a kick target location represented by `target_2d`.

```
def kickTarget(self, strategyData, mypos_2d=(0,0),target_2d=(0,0), abort=False, enable_pass_command=False):
    '''
    Walk to ball and kick

    Parameters
    ----------
    strategyData : object
        containing game state variables
    mypos_2d : array_like
        2D target in absolute coordinates
    target_2d : array_like
        2D target in absolute coordinates
    abort : bool
        True to abort.
        The method returns True upon successful abortion, which is immediate while the robot is aligning itself.
        However, if the abortion is requested during the kick, it is delayed until the kick is completed.
    avoid_pass_command : bool
        When False, the pass command will be used when at least one opponent is near the ball

    Returns
    -------
    finished : bool
        Returns True if the behavior finished or was successfully aborted.
    '''
```

You will see in the template code provided that there are a few calls to the `kickTarget` function found in the `select_skill`.

```
#---------------------------------------------------
# Example Behaviour
target = (15,0) # Opponents Goal

if strategyData.active_player_unum == strategyData.robot_model.unum: # I am the active player
    drawer.annotation((0,10.5), "Pass Selector Phase" , drawer.Color.yellow, "status")
else:
    drawer.clear_player()

if strategyData.active_player_unum == strategyData.robot_model.unum: # I am the active player
    pass_reciever_unum = strategyData.player_unum + 1 # This starts indexing at 1, therefore player 1 wants to pass to player 2
    if pass_reciever_unum != 6:
        target = strategyData.teammate_positions[pass_reciever_unum-1] # This is 0 indexed so we actually need to minus 1
    else:
        target = (15,0)

    drawer.line(strategyData.mypos, target, 2,drawer.Color.red,"pass line")
    return self.kickTarget(strategyData,strategyData.mypos,target)
else:
    drawer.clear("pass line")
    return self.move(strategyData.my_desired_position, orientation=strategyData.ball_dir)
```

Here we determine a pass recipient (`pass_reciever_unum`) to be the player whose player number is one plus the closest players number. Calculated as:

```
pass_reciever_unum = strategyData.player_unum + 1 # This starts indexing at 1, therefore player 1 wants to pass to player 2
```

From that we determine the `target` to be the location of the `pass_reciever_unum` found in `teammate_postions`. Calculated as:

```
target = strategyData.teammate_positions[pass_reciever_unum-1] # This is 0 indexed so we actually need to minus 1
```

Finally we kick the ball into the goal is Player 11 has the ball as seen below:

```
target = (15,0)
```

If the player returning the `kickTarget` action is not within range to actually successfully perform a kick then the player will first walk towards the ball until they can kick. This is handled for you and does not need to be taken into consideration.

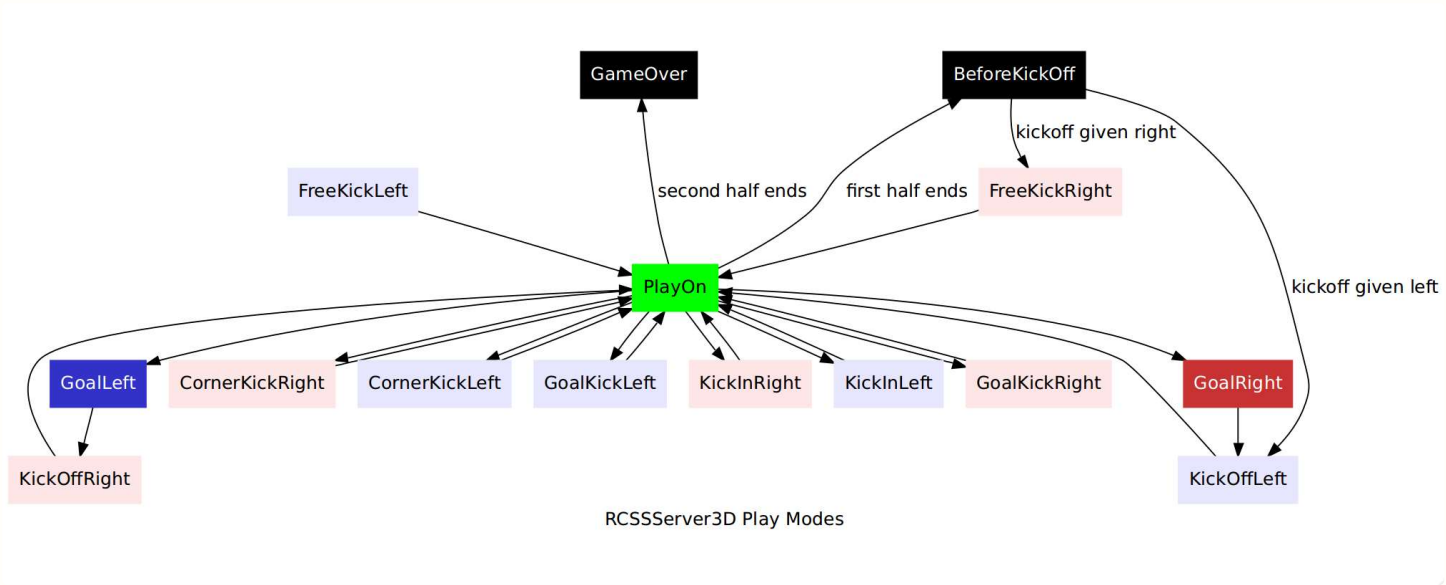
Note: The passing behaviour described above constitutes the behaviour of Baseline 1

7.4.3 getup

If your player falls over for any reason they will begin to execute the `getup` behaviour. Therefore you do not have to worry about it.

7.5 Game Modes

During the course of a match the players will have to adapt to number of different game modes. These include ones related to Corner Kicks, Goal Kicks, Kick Ins ect. For the most part the game will run in what is called Play On. However, as seen below there can many transitions to and from different game modes.



RCSSServer3D Play Modes

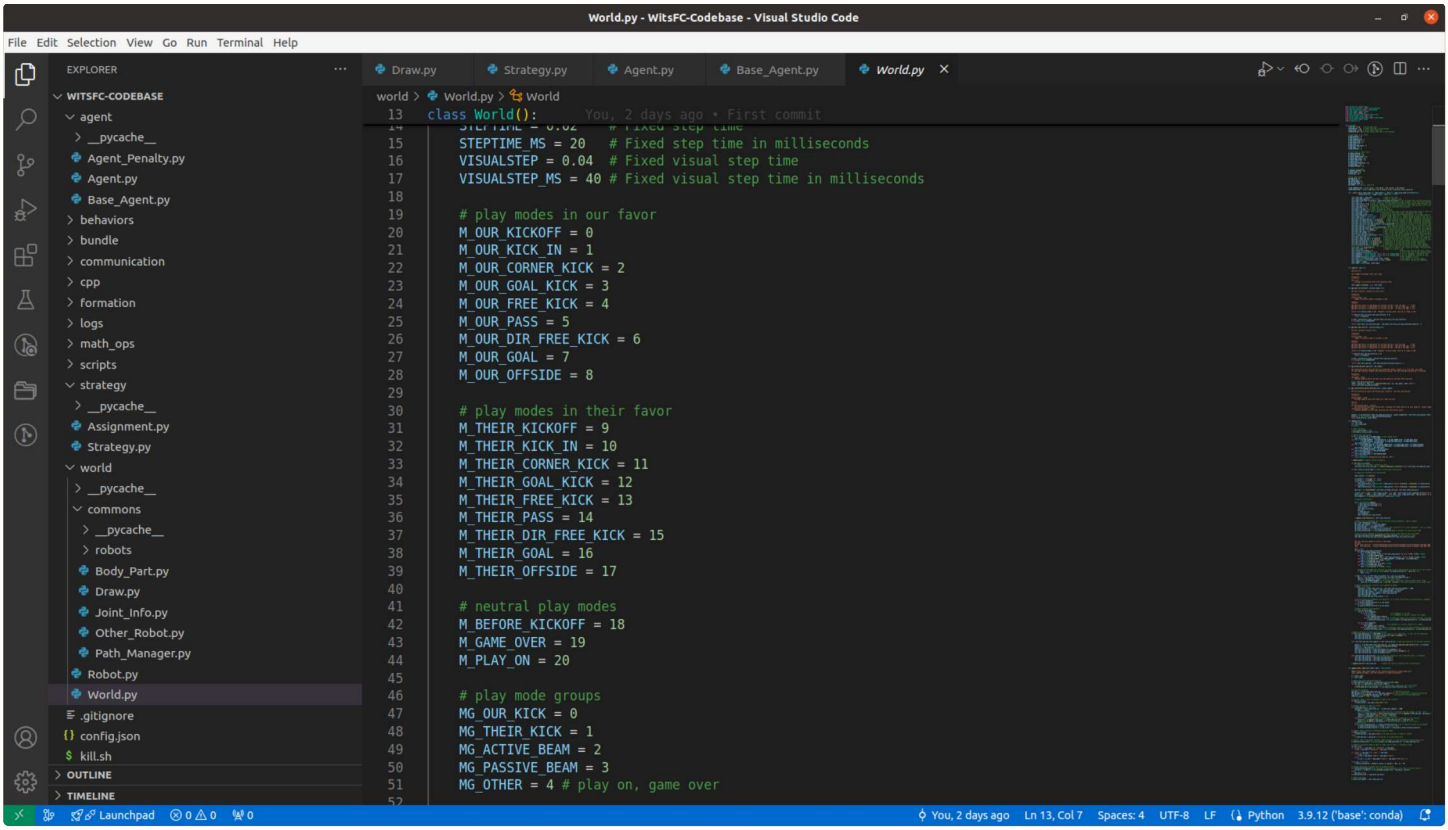
The current play mode value can be referenced from inside the `Strategy` class and is stored inside the `play_mode` variable as seen below.

```
class Strategy():
    def __init__(self, world):
        self.play_mode = world.play_mode
        self.robot_model = world.robot
```

This variable can then be compared against a set of all game mode constants which is stored inside the `World` class. Below we can see that we are trying to check if the current `play_mode` is `M_GAME_OVER`. This example is found in the `think_and_send` function inside `Agent.py`.

```
if strategyData.play_mode == self.world.M_GAME_OVER:
    pass
```

The figure below shows the names given to all the different play modes stored as constants within the `World` class.



The table below is a comprehensive list of the possible game modes a team will encounter.

Play mode	Description	Conditions
BeforeKickOff	Before the match	The ball is at (0,0), the midfield and may not be moved. Players may use their beam effectors. Game time does not progress. This state is only left when a user instructs the simulator to start.
KickOff_Left	Kick off for the left team	The left team have a period in which to make their first kick. During this time the right team are lot allowed to cross the centre line, or inside the centre circle. The left team may not cross the centre line, unless they are within the goal circle.
KickOff_Right	Kick off for the right team	The right team have a period in which to make their first kick. During this time the left team are lot allowed to cross the centre line, or inside the centre circle. The right team may not cross the centre line, unless they are within the goal circle.
PlayOn	Regular gameplay	
KickIn_Left	Kick in left team	The right team have kicked the ball off the side of the field. The ball is positioned on the sideline at the position it left the field. The right team are not allowed within a fixed radius of the ball, and the left team have a period of time in which to kick the ball back into play.
KickIn_Right	Kick in right team	The left team have kicked the ball off the side of the field. The ball is positioned on the sideline at the position it left the field. The left team are not allowed within a fixed radius of the ball, and the right team have a period of time in which to kick the ball back into play.
CORNER_KICK_LEFT	Corner kick left team	
CORNER_KICK_RIGHT	Corner kick right team	
GOAL_KICK_LEFT	Goal kick for left team	
GOAL_KICK_RIGHT	Goal kick for right team	
OFFSIDE_LEFT	Offside for left team	Currently unused.
OFFSIDE_RIGHT	Offside for right team	Currently unused.

Play mode	Description	Conditions
GameOver	After the match	Play has finished. Agents may still move about, but no actions will have an effect upon the result of the match.
Goal_Left	Goal scored by the left team	This state exists for a few moments, before transitioning to KickOff_Right.
Goal_Right	Goal scored by the right team	This state exists for a few moments, before transitioning to KickOff_Left.
FREE_KICK_LEFT	Free kick for left team	The right team are not allowed within a fixed radius of the ball, and the left team have free access. PlayOn commences when the left team touches the ball, or if they fail to do so after a fixed period.
FREE_KICK_RIGHT	Free kick for right team	The left team are not allowed within a fixed radius of the ball, and the right team have free access. PlayOn commences when the right team touches the ball, or if they fail to do so after a fixed period.
NONE	No or unknown play mode	Agents should never receive this play mode.

7.6 Field Layout

The following Figures depict the field as well as give you a better visualistation of the coordinate system in place.

In the figure above you can see the visualisation of the field before any agents have been added. On the field you can also see 30 visual stripes of alternating green shades. Now these stripes aren't only to simulate that grass effect of a real field but also as a visual key to see different positions on the field. Since our field width denoted by FIELD_X = 30 is equal to 30 each stripe represents a unique integer value from -HALF_FIELD_X to HALF_FIELD_X or -15 to 15 with the center being at FIELD_CENTER_X or 0. Additionally we have drawn lines on the field between two pairs of points namely (-5,10),(-5,-10) and (5,10),(5,-10).

An Important thing to note is that this visualisation would be flipped for the team loaded in on the right hand side. This means that the position of your own goal is always at (-15,0) and the opponents goal is always at (15,0). The figure below depicts the visualisation of those same coordinates from the perspective of the team on the right hand side. The left hand side team does not draw anything to the visualiser. The same code as above was utlised.

7.7 Initial Position of Players

When you load a team you will find that the individual players are all placed at certain locations. This can be considered there `initial formation`. These initial positions can be modified by modifying the `init_pos` variable in the `Agent` class inside `Agent.py`. Currently you will find the following initial formation:

```
self.init_pos = ([-13,0],[-7,-2],[0,3],[7,1],[12,0])[unum-1] # initial formation
```

This assigns a 2d position for each of our 5 players for example `Player 1` will be assigned an initially position of `[-13,0]`.

7.8 Rule and Fouls

There are multiple “cardable” offenses and fouls which a player can recieve. In such cases the penalised player is teleported to the side of the field where they will then be able to instantly come back on.

7.8.1 Kick ins

There are a number of game modes which result in a kick in needing to be performed. The list of these can be seen below:

```
# play modes in our favor
M_OUR_KICKOFF = 0
M_OUR_KICK_IN = 1
M_OUR_CORNER_KICK = 2
M_OUR_GOAL_KICK = 3
M_OUR_FREE_KICK = 4
M_OUR_PASS = 5
M_OUR_DIR_FREE_KICK = 6
M_OUR_GOAL = 7
M_OUR_OFFSIDE = 8

# play modes in their favor
M_THEIR_KICKOFF = 9
M_THEIR_KICK_IN = 10
M_THEIR_CORNER_KICK = 11
M_THEIR_GOAL_KICK = 12
M_THEIR_FREE_KICK = 13
M_THEIR_PASS = 14
M_THEIR_DIR_FREE_KICK = 15
M_THEIR_GOAL = 16
M_THEIR_OFFSIDE = 17
```

In such situations a circle will form around the ball which only allows the team performing the kick in, to go inside it. This is to prevent the opposition from disrupting the play.

Note you can not score directly from any Kick In mode

7.8.2 Double Touch

During set plays (Kick Ins, Goal Kicks, Kick Off ect.) the same player can not touch the ball twice, in such cases the team who did not commit the offence will recieve a KICK IN of their own.

7.8.3 Charging and Touching

This is a cardable offensive which results in the offending player being beamed off the field. It occurs when an player bumps into the back side of another player This can sometimes be quite fickle and we have seen that other forms of contact such as side on and even accidental contact can lead to a charging offensive.

7.8.4 offside

There is no rule governing offsides, however, during and before kick off you cannot move into the opponents half. Failing to do so will result in the offending player being teleported(beamed) back onto their own half.

7.9 Drawing Visualisation Aids

The below video which corresponds to an example behaviour which firstly naivelly performs role assignment and then pass selection demonstrates the drawing capabilities of the codebase.

Here we can see examples of text and line drawing. text drawing is handled by the `annotation` function which can be found in the `Draw.py` file. It is implement as follows:

```
def annotation(self, pos, text, color:bytes, id:str, flush=True):
    '''
    Draw annotation

    Examples
    ----------
    Annotation in 3D: annotation((1,1,1), "SOMEtext!", Draw.Color.red, "my_annotation")
    Annotation in 2D (z=0): annotation((1,1), "SOMEtext!", Draw.Color.red, "my_annotation")
    '''
    if not self.enabled: return
    if type(text) != bytes: text = str(text).encode()
    assert type(color)==bytes, "The RGB color must be a bytes object, e.g. red: b'\xFF\x00\x00'"
    z = pos[2] if len(pos)==3 else 0

    if self._is_team_right:
        pos = (-pos[0],-pos[1],pos[2]) if len(pos)==3 else (-pos[0],-pos[1])

    msg = b'\x02\x00' + (
    f'{f"{pos[0]  :.4f}":.6s}'
    f'{f"{pos[1]  :.4f}":.6s}'
    f'{f"{z       :.4f}":.6s}').encode() + color + text + b'\x00'

    Draw._send(msg, self._prefix + id.encode(), flush)
```

Here you can see it takes a position where it should be rendered as well as the actual text to be displayed, the colour it should be written in and a unique identifier for the drawing. This identifier can be used to edit or clear the drawing later.

The second example drawing is that of a line, which once again is found in the `Draw.py` file and is implemented as follows :

```
def line(self, p1, p2, thickness, color:bytes, id:str, flush=True):
    '''
    Draw line

    Examples
    ----------
    Line in 3D: line((0,0,0), (0,0,2), 3, Draw.Color.red, "my_line")
    Line in 2D (z=0): line((0,0), (0,1),   3, Draw.Color.red, "my_line")
    '''
    if not self.enabled: return
    assert type(color)==bytes, "The RGB color must be a bytes object, e.g. red: b'\xFF\x00\x00'"
    assert not np.isnan(p1).any(), "Argument 'p1' contains 'nan' values"
    assert not np.isnan(p2).any(), "Argument 'p2' contains 'nan' values"

    z1 = p1[2] if len(p1)==3 else 0
    z2 = p2[2] if len(p2)==3 else 0

    if self._is_team_right:
        p1 = (-p1[0],-p1[1],p1[2]) if len(p1)==3 else (-p1[0],-p1[1])
        p2 = (-p2[0],-p2[1],p2[2]) if len(p2)==3 else (-p2[0],-p2[1])

    msg = b'\x01\x01' + (
    f'{f"{p1[0]  :.4f}":.6s}'
    f'{f"{p1[1]  :.4f}":.6s}'
    f'{f"{z1     :.4f}":.6s}'
    f'{f"{p2[0]  :.4f}":.6s}'
    f'{f"{p2[1]  :.4f}":.6s}'
    f'{f"{z2     :.4f}":.6s}'
    f'{f"{thickness :.4f}":.6s}').encode() + color

    Draw._send(msg, self._prefix + id.encode(), flush)
```

Both these functions can be seen in action in the Pass Selector section of `select_skill`, here we use the `drawer` object to access the drawing functions. Note the usage of `clear_player` and `clear` for the purpose of clearing all the drawings for a particular player as well as clearing a particular drawing based on an id (`pass line`).

```
#-----------------------------------------------------
#Pass Selector
if strategyData.active_player_unum == strategyData.robot_model.unum: # I am the active player
    drawer.annotation((0,10.5), "Pass Selector Phase" , drawer.Color.yellow, "status")
else:
    drawer.clear_player()



if strategyData.active_player_unum == strategyData.robot_model.unum: # I am the active player
    target = pass_reciever_selector(strategyData.player_unum, strategyData.teammate_positions, (15,0))
    drawer.line(strategyData.mypos, target, 2,drawer.Color.red,"pass line")
    return self.kickTarget(strategyData,strategyData.mypos,target)
else:
    drawer.clear("pass line")
    return self.move(strategyData.my_desired_position, orientation=strategyData.ball_dir)
```

« 6 Debugging

9 Submission 1 - Role Assignment »

"**COMS3005A RoboCup Assignment**" was written by Branden Ingram.

This book was built by the bookdown R package.