SECME Project Starter Guide

Getting Started with Scratch

1. Make an account at www.scratch.mit.edu.
2. Decide whether you'd like to make a story or a game.
3. Brainstorm some ideas and write them down on a sheet of paper.
4. Plan out how you intend to implement your ideas.
    a. Break down the actions, and use pseudocode to describe events and if/then conditions before starting to build the program.
5. Build your idea on Scratch
6. Have lots of people pay your game or story, get feedback, improve!
7. **Share your project! (Make sure the project is "Shared")**

Here's an example of a winning story project: https://scratch.mit.edu/projects/282844316/
Here's an example of a winning game project: https://scratch.mit.edu/projects/281013206/

Notice the following:

- Very well written **Instructions** for the end user
- Appropriate **Notes and Credits** given
- A complete story with subject, setting, problem, resolution / call to action
- Original art and sound
- Positive message, interactive storyline

Getting Started with Python

1. Download Python 3.7.4

    a. Download Link: https://www.python.org/downloads/release/python-374/
    b. Video guide: https://www.youtube.com/watch?v=rVb1TqqbPj0

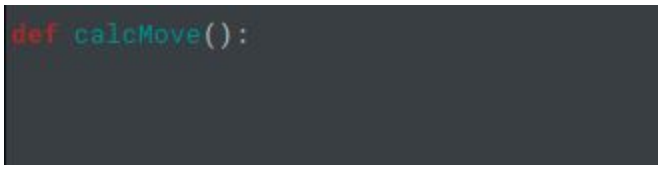
2. Go over the basics of python

    a. https://www.w3schools.com/python/

3. Write down how you want to implement your ideas on a sheet of paper.

4. Review the framework for using the functions in our framework. The most important functions are **getGameState(), getTeam(), calcMove(),** and **submitMove()**. Here's an example of how you should use these functions to get the information your computer player needs to see the game board, make it's decision on where to put its next move and submitting that move:

    First, you'll need to define the function called **calcMove()**. Basically, whenever it is your "player's" turn, our framework will call this function to allow your player to calculate and determine its next move. All of your steps for determining the next move should be contained inside this function. Your code should now look something like this:

    Next, you'll need some way for your computer player to see the game board, so they know what to do next! That's where **getGameState()** comes in! This function is part of our framework, and when its called, will return a 2-D list of the characters on the game board. What is 2-D list, you ask? In Python, a **list** can be seen as well, a list of things. In this case, it's a list of characters on the board. On our board, every tile is either an 'X' character, an 'O' character or a space character ' '. The syntax for a 1-D array is listName[index] where "listName" is the name of the list variable, and "index" is the position in the list. **IMPORTANT NOTE: the first element of a list is ALWAYS at index [0], so the first element of the example listName array is listName[0].**

    But what is a **2-D list**? Basically, a list of lists. But to put it more simply, it can be seen as a table. A 2-D list has 2 indices and they represent the row and column of our tic tac toe board. The board is laid out in the following way:

    [0] [0] [0] [1] [0] [2]

[1] [0] [1] [1] [1] [2]

[2] [0] [2] [1] [2] [2]

So, if you wanted to get the middle tile, you would use the index [1][1]

But first, you'll need to create your own list to hold the list returned by getGameState(). In Python, it would look something like this:

myList = getGameState()

This will get the game board list and store it in a variable called **myList** (you can name this variable whatever you'd like it doesn't have to be myList, choose a name that makes sense to you)**.**

Now you can get the value of any space on the board using the above chart!
The bottom right space would be myList[2][2], the top right space would be myList[0][2] etc.

So now, in your calcMove() function, you can create a variable to store the game board:

```
def calcMove():
        myList = getGameState()
```

Great! So now we have a way of looking at the board. You can also get your team (either 'X' or 'O') with the getTeam() function. It is important that you use this if you want your computer player to know what team it is on, as it can change from game to game. You can use it like this:

myTeam = getTeam()

As before, you can use whatever variable name you want instead of myTeam

Now your code should look something like this:

```
def calcMove():

        myList = getGameState()

        myTeam = getTeam()
```

Here comes the fun part! Now you need to figure out how to make the best move based on the current state of the board. This is where your computer player needs to actually "think". YOU need to create the algorithm to figuring out where you should place your 'X' or 'O'. **Remember, this is a difficult challenge.** In a bit we'll give you some tips on some tools you can use to help you with this.

Once your computer player has decided its move you **must** use the **submitMove()** function to send your move to the game. You would want to create two variables to hold onto these moves at the top of **calcMove()**, one for the row and one for the column that your move should be placed at. Your player should change the values of the row and column variables to whatever they need (they should be integers), and when your computer player figures out the perfect move, you can submit your move using:

submitMove(row, column)

Where **row** and **column** are integer values that correspond to tiles on the tic tac toe board (see the board chart above)

So, in the end your code should look like this:

```
def calcMove():

        row = 0
        column = 0

        myList = getGameState()

        myTeam = getTeam()

        ####### HERE'S WHERE YOUR ALGORITHM SHOULD GO #########

        submitMove(row, column)
```

Of course you need to replace the comment with your on algorithm, otherwise your player would always submit 0,0 and you can't win that way.

Now since the nature of this competition is to create your own algorithm or set of steps to finding the next best move, we can only tell you so much, but here are some tips to creating your own algorithm:

Iterating through a 2D array:

You may or may not be familiar with python for loops, if not we suggest you look here:

https://www.w3schools.com/python/python_for_loops.asp

Once you understand how for loops work, you can use them to go through each element in an array one by one.

Here's and example with a 1-D list:

exampleList = ['A', 'B', 'C', 'D']

To go through all of the elements in the list and print them you could use this:

```
for index in range(exampleList.length)
        print(exampleList[index])
```

Remember "index" here is a variable so you can name it what you want

Now we can apply this same process for a 2-D array, looping through each row while looping through each item in that row.

Here's an example with the list we get from **getGameState():**

myList = getGameState()

We'll call our list variable **myList**

Let's say we wanted to go through all of the squares in the board and print them

Since we can look at a 2D list as like a table, we can go down through each row, and for each row go through each column in that row. Remember a 2D list is lists inside a list, so we need a loop inside a loop!

To go through the elements of this list and print them to the console you would do this:

```
for i in range(myList.length):
        for j in range(myList[i].length)
                print(myList[i][j]
```

Remember “i” and “j” are variables, so you can name them whatever you like, just don’t confuse them with other variables

Notice the **myList[i].length** in the second loop. Since each row is an array itself, we can use this to get the length of the current row

Using If Statements:

Let’s say you want our player to place their move on the center tile [1][1]

First we should check if that tile is empty or not (if it’s not empty, you shouldn’t place your move there, or our application will just skip your turn)

We could use an **if** statement to determine whether we should set our move there:

```
if myList[1][1] == ' ':
        myRow = 1
        myColumn = 1
```

Assuming that we are using the same variable names from when we set up our computer player, we test to see if the center tile is empty (represented by a space char ' '), if it is, set our move there. Notice that we use **==** instead of **=**. A single equal sign **=** means assignment in java, assigning a value to a variable. Double equal **==** means to check if the two values are equal and return true or false.

Chaining if Statements:

Maybe we want it to do something else if it the middle spot isn’t empty, in this case we could use an **elif** (else if):

```
if myList[1][1] == ' ':
        myRow = 1
        myColumn = 1
```

```
elif myList[1][2] == ' ':
        myRow = 1
        myColumn = 2
```

Here, we check if the middle space is empty, if it is , place our move there, otherwise, check if [1][2] is empty (the rightmost tile in the center row), if it is, place it there, otherwise, do nothing.

You can chain multiple else if's to a single if, and it will go through each one until it finds the one that evaluates true.

Nested If:

Maybe you want to test one condition, and if it's true, test another condition:

Let's say if the middle tile is marked with our team character (depending on getTeam())
And the tile one to the right is empty and we want to place our tile there:

```
If myList[1][1] == myTeam:
        If myList[1][1+1] == ' ':
                myRow = 1
                myColumn = 1 + 1
```

This can also be done more elegantly like this using the **and** operator:

```
If myList[1][1] == myTeam and myList[1][1+1] == ' ':
        myRow = 1
        myColumn = 1 + 1
```

Before winter break, we will begin hosting a site where you can test your code in our framework. More information on that to come.

Getting Started with Java

1. Download and install the Java developers kit(JDK)

    a. Written guide for JDK: https://docs.oracle.com/en/java/javase/12/install/installation-jdk-microsoft-windows-platforms.html
    b. Video guide for JDK: https://www.youtube.com/watch?v=uadGsNA6h5Q
    c. Download Link: https://www.oracle.com/technetwork/java/javase/downloads/jdk12-downloads-5295953.html

2. Download and install the Java Runtime Environment(JRE)

    a. Written guide for JRE: https://java.com/en/download/help/windows_manual_download.xml
    b. Video guide for JRE: https://www.youtube.com/watch?v=Zx3ceLdFm64
    c. Download Link: https://java.com/en/download/manual.jsp

3. Download one of the following IDEs (**Recommended.** You can do your project without an IDE, but most people find it easier to use one)

    a. Eclipse: eclipse.org/downloads/
    b. Visual Studio Code: https://code.visualstudio.com/
    c. NetBeans: https://netbeans.apache.org/download/index.html

4. Go over the basics of java

    a. https://www.w3schools.com/java/

5. Review the framework for using the functions in our application. The most important functions are **getGameState(), getTeam(), calcMove(),** and **submitMove()**. Here's an example of how you should use these functions to get the information your computer player needs to see the game board, make its decision on where to put its next move and submitting that move:

    First, your java code should be a class named **Lastname_Firstname**, where you substitute your own name. Your .java source code file should also be named **exactly the same as your class name** (Java requires that all .java files have a class in them that has the same name of the file) make sure you use the **public** keyword before your class. Also make sure that your class **extends** the class **Player**. This is what is known as *inheritance* in java, it says that your class is a *type of* player, and makes sure that it can use all of the functions in our Player class (your IDE will give you an error that class "Player" is not found, this is ok for now since your code needs to be integrated with our

code to run) For example, if your name were John Smith, the top of your file should look something like this:

```
public class Smith_John extends Player {


}
```

Next, you'll need to create the *function* (or *method* as they are called in java) called **calcMove()**. Basically, whenever it is your "player's" turn, our framework will call this function to allow your player to calculate and determine its next move. All of your steps for determining the next move should be contained inside this function. The function should be placed *inside* your class (between the curly braces) and should be declared as **public** and **Void**. Your code should now look something like this:

```
public class Smith_John extends Player {

    public void calcMove(){



    }

}
```

Next, you'll need some way for your computer player to see the game board, so they know what to do next! That's where **getGameState()** comes in! This function is part of the **Player class** from our framework, and when its called, will return a 2-D array of chars. What is that, you ask? In Java, an **array** can be seen as a list of similar things, like a list of numbers, or a list of objects of the same type. In this case, it's a list of **char** values. In Java, all variables have a **type,** like **int** for integer numbers, **float** for numbers with a decimal point, or in this case **char** for a single character, like a letter, number, symbol, punctuation mark etc. On our board, every tile is either an 'X' character, an 'O' character or a space character ' '. The syntax for a 1-D array is arrayName[index] where "arrayName" is the name of the array variable, and "index" is the position in the array. **IMPORTANT NOTE: the first element of an array is ALWAYS at index [0], so the first element of the example arrayName array is arrayName[0].**

But what is a **2-D array**? Basically, an array of arrays. But to put it more simply, it can be seen as a table. A 2-D array has 2 indices and they represent the row and column of our tic tac toe board. The board is laid out in the following way:

[0] [0]	[0] [1]	[0] [2]
[1] [0]	[1] [1]	[1] [2]
[2] [0]	[2] [1]	[2] [2]

So, if you wanted to get the middle tile, you would use the index [1][1]

But first, you'll need to create your own array to hold the array returned by getGameState(). In Java, it would look something like this:

char[][] myArray = getGameState();

This will get the game board array and store it in a variable called **myArray** (you can name this variable whatever you'd like it doesn't have to be myArray, choose a name that makes sense to you)**.**

Now you can get the value of any space on the board using the above chart!
The bottom right space would be myArray[2][2], the top right space would be myArray[0][2] etc.

So now, in your calcMove() function, you can create a variable to store the game board:

```
public class Smith_John extends Player {

    public void calcMove(){

        char[][] myArray = getGameState();

    }

}
```

Great! So now we have a way of looking at the board. You can also get your team (either 'X' or 'O') with the getTeam() function. It is important that you use this if you want your computer player to know what team it is on, as it can change from game to game. You can use it like this:

char myTeam = getTeam();

As before, you can use whatever variable name you want instead of myTeam

Now your code should look something like this:

```
public class Smith_John extends Player {

    public void calcMove(){

        char[][] myArray = getGameState();

        char myTeam = getTeam();

    }

}
```

Here comes the fun part! Now you need to figure out how to make the best move based on the current state of the board. This is where your computer player needs to actually "think". YOU need to create the algorithm to figuring out where it should place its 'X' or 'O'. **Remember, this is a difficult challenge.** In a bit we'll give you some tips on some tools you can use to help you with this.

Once your computer player has decided its move you **must** use the **submitMove()** function to send your move to the game. You would want to create two variables to hold onto these moves at the top of **calcMove()**, one for the row and one for the column that your move should be placed at. Your player should change the values of the row and column variables to whatever they need, and when your computer player figures out the perfect move, you can submit your move using:

submitMove(row, column);

Where **row** and **column** (can be named whatever you choose) are **int** values that correspond to tiles on the tic tac toe board (see the board chart above)

NOTE: DO NOT attempt to directly modify the board variable you created from getGameState(). Only use it to assess the game (think "read only"). The board variable you receive from getGameState() is not a reference to the game board but a copy, so you can't just set the board so you win immediately (nice try though) You should always use **submitMove()**.

So, in the end your code should look like this:

```
public class Smith_John extends Player {

    public void calcMove(){

        int row = 0 , column = 0;

        char[][] myArray = getGameState();

        char myTeam = getTeam();

        ////////////HERE'S WHERE YOUR ALGORITHM GOES/////////////

        submitMove(row, column);

    }

}
```

Of course you need to replace the comment with your own algorithm, otherwise your player would always submit 0,0 and you can't win that way.

Now since the nature of this competition is to create your own algorithm or set of steps to finding the next best move, we can only tell you so much, but here are some tips for creating your own algorithm:

Iterating through a 2D array:

You may or may not be familiar with java for loops, if not we suggest you look here:

https://www.w3schools.com/java/java_for_loop.asp

Once you understand how for loops work, you can use them to go through each element in an array one by one.

Here's an example with an array called **exampleArray:**

char[] exampleArray = new char[3];

This statement will create a new char array that is 3 elements long (**REMEMBER**: arrays always **start counting at 0**, so the elements can be accessed by exampleArray[0], exampleArray[1] and exampleArray[2]. **exampleArray[3]** will give you an out of bounds error!

To go through the elements of this array and set each one to the char 'X' you would do this:

```
for(int index = 0; index < exampleArray.length; ++index){

        exampleArray[index] = 'X';
}
```

Notice how the index starts at 0, goes up by one each time around the loop, and then stops when it reaches the length of the array (remember we use < instead of <= because the length (3) would be out of bounds as an index)

Now we can apply this same process for a 2-D array, looping through each row while looping through each item in that row.

Consider the 2D array exampleArray2:

```
Char[][] exampleArray2 = char[3][3];
```

Since we can look at a 2D array as like a table, we can go down through each row, and for each row go through each column in that row. Remember a 2D array is arrays inside an array, so we need a loop inside a loop!

```
for(int row = 0; exampleArray2.length; ++row){
        for(int column = 0; exampleArray2[row].length; ++row){
                exampleArray[row][column] = 'X';
        }
}
```

Notice the **exampleArray2[row].length** in the second loop. Since each row is an array itself, we can use this to get the length of the current row

We can use the same structure to get values from the array as well:

```
for(int row = 0; exampleArray2.length; ++row){
        for(int column = 0; exampleArray2[row].length; ++row){
                System.out.println(exampleArray[row][column]);
        }
}
```

Here we go through the whole array and print each of the values to the console, since we set each value to 'X' in the last part, we should get 9 X's, each on their own row.

Using If Statements:

Let's say you want our player to place their move on the center tile [1][1]

First we should check if that tile is empty or not (if it's not empty, you shouldn't place your move there, or our application will just skip your turn)

We could use an **if** statement to determine whether we should set our move there:

```
if(myArray[1][1] == ' '){
        myRow = 1;
        myColumn = 1;
}
```

Assuming that we are using the same variable names from when we set up our computer player, we test to see if the center tile is empty (represented by a space char ' '), if it is, set our move there. Notice that we use **==** instead of **=**. A single equal sign **=** means assignment in java, assigning a value to a variable. Double equal **==** means to check if the two values are equal and return true or false.

Chaining if Statements:

Maybe we want it to do something else if it the middle spot isn't empty, in this case we could use an **else if**:

```
if(myArray[1][1] == ' '){
        myRow = 1;
        myColumn = 1;
}
else if(myArray[1][2] == ' '){
        myRow = 1;
        myColumn = 2;

}
```

Here, we check if the middle space is empty, if it is , place our move there, otherwise, check if [1][2] is empty (the rightmost tile in the center row), if it is, place it there, otherwise, do nothing.

You can chain multiple else if's to a single if, and it will go through each one until it finds the one that evaluates true.

Nested If:

Maybe you want to test one condition, and if it's true, test another condition:

Let's say if the middle tile is marked with our team character (depending on getTeam())
And the tile one to the right is empty and we want to place our tile there:

```
if(myArray[1][1] == myTeam){
        if(myArray[1][1+1] == ' '){
                myRow = 1;
                myColumn = 1 + 1;
        }
}
```

This can also be done more elegantly like this using the and operator, **&&**:

```
if(myArray[1][1] == myTeam && myArray[1][1+1] == ' '){
        myRow = 1;
        myColumn = 1 + 1;
}
```

Before winter break, we will begin hosting a site where you can test your code in our framework. More information on that to come.