thebillington

development blog

Written 6 hours ago by Billy Rebecchi

Writing a Tic Tac Toe game in python

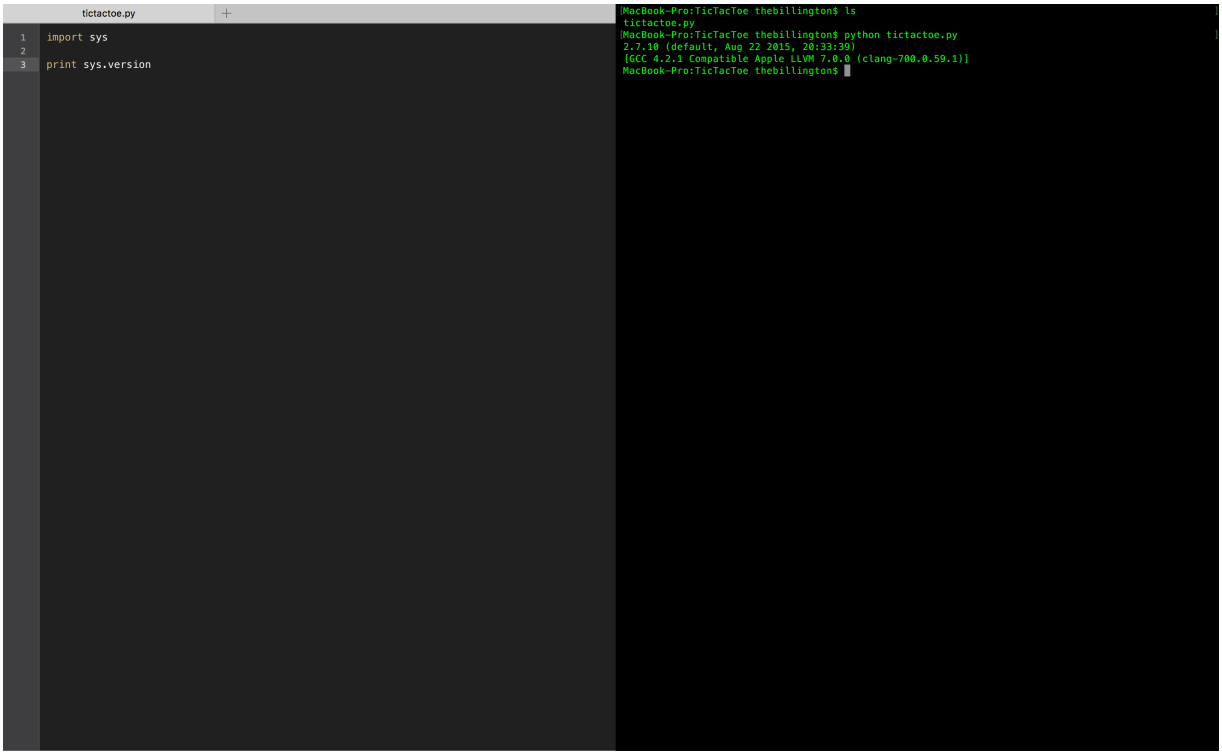
Python is an extremely versatile language with relatively simple syntax that is very easy for a beginner to pick up. Today we are going to get started using python to write basic logic for a tic tac toe game.

Getting Started

First we are going to need to setup our development environment. On Mac OSx you should have Python installed by default. You can check this by creating a simple python script in a text editor and running it on the command line. My favourite text editor for Mac is available for free from the Text Mate[1] website. On Windows you can download an Integrated Development Environment (IDE) called PyCharm[2]. I'm going to be using Python 2.7.7 for our code today.

Once you have your basic IDE started up create a new directory in your Documents called `Python` and add a new directory to this folder called `TicTacToe`. We are now going to open up our IDE/Text Editor and create an empty file. Save this file in your TicTacToe directory and name it `tictactoe.py`. I'm using El Capitan so I have multi window mode enabled, so I can open my text editor alongside a `Terminal` window. You can find `Terminal` by navigating to `Applications > Utilities > Terminal`.

With my text editor and Terminal window open my screen now looks like this:



You can see on the left that I have a file open which contains my python code. I can then run my python code by calling:

```
python tictactoe.py
```

I have written some basic code in to check my version of python. Try pasting this into the python file, saving and running it:

```
import sys
print sys.version
```

Printing our board

The next thing we are going to do is write a function that when called, will print our game board out. Our game board is going to be printed with two conditions:

1) For any square, if one of our players has already chosen that square then print an

"X" or an "O".
2) For any square, if neither player has chosen that square, then print a number "i" denoting that squares value.

So the first thing we are going to do is create an empty list with 9 values, and populate it with string values denoting the integers from 1-9:

```
choices = []


for x in range (0, 9) :
        choices.append(str(x + 1))
```

A list is created using the `= []` notation as above. What this does is creates a place in memory where we can store more than one value. Once we have a list, we can add an element to it by calling the `.append()` function on it. The above code creates an empty list, which we can access the values of by calling `choices[x]` where x is any integer that is less than the length of the list minus one.

So if we compiled the above code, and then added a function to print the 0th element, we would print out the number 1:

```
print choices[0]
```

We are now going to print out our values in the style of a game board. So let's start by defining a function called `printBoard`:

```
def printBoard() :
```

Next we are going to write our `printBoard()` function. We start by printing out a line with dashes to represent the top of our game board. We are then going to print each of the elements in the list that we created before:

```
def printBoard() :
        print '\n -----'
        print '|' + choices[0] + '|' + choices[1] + '|' + choices[2] + '|'
```

```
        print ' -----'
        print '|' + choices[3] + '|' + choices[4] + '|' + choices[5] + '|'
        print ' -----'
        print '|' + choices[6] + '|' + choices[7] + '|' + choices[8] + '|'
        print ' -----\n'
```

Save and run your code now and nothing happens. This is because we have defined our print board function but not made a call to it in our execution code. Try adding in a call to `printBoard()` to your code. Your full class should now look like this:

```
choices = []

for x in range (0, 9) :
        choices.append(str(x + 1))

def printBoard() :
        print '\n -----'
        print '|' + choices[0] + '|' + choices[1] + '|' + choices[2] + '|'
        print ' -----'
        print '|' + choices[3] + '|' + choices[4] + '|' + choices[5] + '|'
        print ' -----'
        print '|' + choices[6] + '|' + choices[7] + '|' + choices[8] + '|'
        print ' -----\n'

printBoard()
```

And your output should look like:

```
  -----
|1|2|3|
  -----
|4|5|6|
  -----
|7|8|9|
  -----
```

So now we have a game board that is created using elements in a list, that we have full control over to change at any time.

Implementing player turns

Next we are going to add functionality in to allow a player to take a turn. Since we only have two players, we only need to keep track of if it is one players go. We can do this using something called a boolean which has two possible values, either True or False.

We are going to give our boolean a name that describes what it does, so add the following code to your class:

```
playerOneTurn = True
```

Now that we have a boolean to keep track of who's turn it is, we need to add another boolean. This one has a completely different job to the above boolean, as it is going to keep track of whether a player has 'won' the game. For as long as this boolean is false, we are going to run our Game Loop.

A game loop is a construct that iterates over game variables to check player position, check enemy position, deal damage and almost everything that you could think to do in a computer game. Our game is very simple though, so we only need our game loop to do a few simple things:

1) Our game loop needs to keep track of who's turn it is.
2) Our game loop needs to know if a player has won.
3) Our game loop needs to provide functionality to allow our players to place their counters.

So let's create our game loop. First we are going to create a boolean value and call it winner, then initialise it to false. We will then write a loop that checks the value of this boolean. The loop is going to check if there has been a winner, and if not allow our next player to take a turn:

```
winner = False
```

```
while not winner :
        print "HERE"
```

The above loop will just print "HERE" over and over until the end of time. To make our game loop useful we are going to add some logic. Our first bit of logic is going to use our current value of the `playerOneTurn` boolean to check who's go it is:

```
if playerOneTurn :
        print "Player 1:"
else :
        print "Player 2:"
```

We are now going to get the box that the player would like to place their token in. We are going to do this by returning their selection, and add it to the (n - 1)th element in our choices list. We can achieve that with the code below:

```
choice = int(input(">> "))

if playerOneTurn :
        choices[choice - 1] = 'X'
else :
        choices[choice - 1] = 'O'
```

The above code stores out input() as an int variable 'choice'. See where we have called the in() operator on our input() function. This is a cast, so the compiler will attempt to take whatever is entered by the user in the terminal, and cast it to an int. Valid inputs are any real numbers e.g {1, 3, 9} would all be accepted. If we enter anything other than an int e.g {h, &, 3.2} then we will receive a runtime error.

Now we check who's turn it is, and if it is player one's turn then we change the element in our choices list to "X", otherwise we change it to "O". We do this by changing the choices[choice - 1] element in our list. The reason we call (choice - 1) instead of just choice is that we have changed the values of our inputs. Remember the code at the start, where we iterated from 0 to 8 and added the (x + 1) values into our empty list. The reason we added the extra 1 is that python is something called a 0 indexed language. Unlike a lot of real life maths where 0 isn't classed as a number, in

a lot of programming languages it is.

This means that if we want to access the first variable in a list, instead of calling list[1] we call list[0]. By calling list[1] we would return the second element in the array. This means when our user chooses the 1st box in our game, we must change the value x they have chosen to x - 1, as this would give us the 0th element in our array.

If we compile and run our entire class as below, then we would be able to add an X to every element in our list. However we haven't yet made our game loop switch players. After every turn of our game, we are going to switch the playerOneTurn to the opposite value by calling the not operator on it, like below:

```
playerOneTurn = not playerOneTurn
```

And now we have all the basics for a Tic Tac Toe game implemented in Python. If we run our full code, then nobody can win. This is because we haven't written conditional logic yet to state what classes as a win. However we have a full board that can be populated with X's and O's.

```
choices = []

for x in range (0, 9) :
        choices.append(str(x + 1))

playerOneTurn = True
winner = False

def printBoard() :
        print '\n -----'
        print '|' + choices[0] + '|' + choices[1] + '|' + choices[2] + '|'
        print ' -----'
        print '|' + choices[3] + '|' + choices[4] + '|' + choices[5] + '|'
        print ' -----'
        print '|' + choices[6] + '|' + choices[7] + '|' + choices[8] + '|'
```

```
        print ' -----\n'

while not winner :
        printBoard()

if playerOneTurn :
                print "Player 1:"
        else :
                print "Player 2:"

choice = int(input(">> "))

if playerOneTurn :
                choices[choice - 1] = 'X'
        else :
                choices[choice - 1] = 'O'

playerOneTurn = not playerOneTurn
```

Checking our winner

The final thing we need to implement in our Python class is a section of our game loop that is going to check if either player has won. Since our players have tokens to represent their go, it is going to be much easier to write our logic, as we don't need to check for two squares containing two O's or containing two X's. We just have to check that the two squares have an equal value.

Let's look at our game board:

```
  -----
|1|2|3|
  -----
|4|5|6|
  -----
|7|8|9|
```

```
-----
```

Our condition for winning Tic Tac Toe is that our player has three counters in a row. But how can we tell if this is true? First lets work out how many combinations we have for a player to have won:

```
1 = 2 = 3
4 = 5 = 6
7 = 8 = 9
1 = 4 = 7
2 = 5 = 8
3 = 6 = 9
1 = 5 = 9
3 = 5 = 7
```

If any of the above conditions are met then it means our game has been won. If our first square is equal to "X", and our second and third squares are both equal to "X" as well then we know that player one must have won.

However we don't want to write eight separate if statements in our code, as this is messy and a waste of resources. Instead let's work out some relationships that represent a winning player.

For all of our horizontal wins, we can see that the nth element is equal to the (n + 1) and (n + 2) elements e.g 1 = 1+1 = 1+2, 4 = 4+1 = 4+2. However we want to ensure that when we write our logic we don't introduce any bugs such as the game thinking that 2 = 2+1 = 2+2 counts as a win. If our second, third and fourth squares are all equal then our condition holds, n = (n+1) = (n+2) but we know that we can't have a winner as our second, third and fourth squares are on different lines.

This means that we need to ensure that we check our condition three times, once with x = 1, once with x = 4 and once with x = 7. Or in logical terms, we need a loop which iterates from 0 to 3 to 6:

```
for x in range (0, 3) :
```

The above loop will iterate from 0 to 1 to 2. This means we need to work out a simple formula that turn our x values into 0, 3 and 6. We can do this by calling x and multiplying it by 3:

```
y = x * 3
```

Now for each of these numbers, let's do a check whether their positions in the list are equal to each other:

```
if (choices[y] == choices[(y + 1)] and choices[y] == choices[(y + 2)]) :
```

Our above line of code first checks whether box[0] = box[1] and box[0] = box[2] contain equal values, ie our letters "X" or "O". It will then run the same check, but for box[3] = box[4] and box[3] = box[5]. We are completing all of our steps to check if the player has won horizontally, but in much fewer lines of code.

Once you have added the above functionality run your full class, and enter the player 1 tokens into the 1st, 2nd and 3rd boxes. Still no winner? This is because we have only run our check, we haven't done anything with it. Inside your if statement we are going to add the following code:

```
winner = True
printBoard()
```

No whenever either player places 3 tokens horizontally, the game will see this as a win, print the game board out and then exit our game loop, as winner has been set to true.

Your full class should now look like this:

```
choices = []

for x in range (0, 9) :
        choices.append(str(x + 1))

playerOneTurn = True
```

```
winner = False

def printBoard() :
        print '\n -----'
        print '|' + choices[0] + '|' + choices[1] + '|' + choices[2] + '|'
        print ' -----'
        print '|' + choices[3] + '|' + choices[4] + '|' + choices[5] + '|'
        print ' -----'
        print '|' + choices[6] + '|' + choices[7] + '|' + choices[8] + '|'
        print ' -----\n'

while not winner :
        printBoard()

if playerOneTurn :
                print "Player 1:"
        else :
                print "Player 2:"

choice = int(input(">> "))

if playerOneTurn :
                choices[choice - 1] = 'X'
        else :
                choices[choice - 1] = 'O'

playerOneTurn = not playerOneTurn

for x in range (0, 3) :
                y = x * 3
                if (choices[y] == choices[(y + 1)] and choices[y] == choices[(y + 2)])
:
                        winner = True
                        printBoard()
```

Now let's consider what happens when we check our vertical wins. If 1 = 4 =7 OR 2 = 5 = 8 OR 3 = 6 = 9 then we know that our player has won. We have already written our loop to do 3 checks, so we can use our value of x to run another check to see if the player has a vertical line:

```
if (choices[x] == choices[(x + 3)] and choices[y] == choices[(x + 6)]) :
        winner = True
        printBoard()
```

Let's think what happens each time we iterate through our for loop with the above statement. Firstly, x = 0 so we check if the 0th element of our list is equal to the 3rd element, as x + 3 = 3. Next it will check if our 0th element is equal to our 6th element as x + 6 = 6. The next iteration of the for loop will check if 1 = 4 = 7, then 2 = 5= 8. And that's it, we've now written another check in for a vertical win.

Lastly, we need to check if we have diagonal wins. Since there are only two checks here we are just going to hardcode them in. Add the following code outside of your for loop:

```
if((choices[0] == choices[4] and choices[0] == choices[8]) or (choices[2] ==
choices[4] and choices[4] == choices[6])) :
        winner = True
        printBoard()
```

This last check is looking if 1 = 5 = 9 or 3 = 5 = 7 which are the two conditions for our player to have a vertical win.

Now we are going to add one more piece of code outside of our game loop. This code is only reached once we have found a winner, so we are going to check which player took the last turn and print out who won:

```
print "Player " + str(int(playerOneTurn + 1)) + " wins!\n"
```

Our code is checking the value of `playerOneTurn`, and converting it to an int, then adding 1 to the value. If a boolean has a True value then it returns as 1. If not then it returns as 0. Since whenever we fill the board, we switch the value of playerOneTurn,

if player one wins then playerOneTurn will equal false. It should be player 2s turn after player 1, but since we had a win condition the game loop stopped execution.

This means when playerOneTurn is true, we know that player two won. So we print out the boolean value as an int and add 1 to it. 1 + 1 = 2, so we print the statement "Player 2 wins!". If playerOneTurn is false we know that player one won on their last go. This means we print out playerOneTurn = false = 0, 0 + 1 = 1 so our code prints the statement "Player 1 wins!".

Your full class should now look like this:

```
choices = []

for x in range (0, 9) :
        choices.append(str(x + 1))

playerOneTurn = True
winner = False

def printBoard() :
        print '\n -----'
        print '|' + choices[0] + '|' + choices[1] + '|' + choices[2] + '|'
        print ' -----'
        print '|' + choices[3] + '|' + choices[4] + '|' + choices[5] + '|'
        print ' -----'
        print '|' + choices[6] + '|' + choices[7] + '|' + choices[8] + '|'
        print ' -----\n'

while not winner :
        printBoard()

if playerOneTurn :
                print "Player 1:"
        else :
                print "Player 2:"
```

```
choice = int(input(">> "))

if playerOneTurn :
                choices[choice - 1] = 'X'
        else :
                choices[choice - 1] = 'O'

playerOneTurn = not playerOneTurn

for x in range (0, 3) :
                y = x * 3
                if (choices[y] == choices[(y + 1)] and choices[y] == choices[(y + 2)])
:
                        winner = True
                        printBoard()

print "Player " + str(int(playerOneTurn + 1)) + " wins!\n"
```

And that's it, we can run our game and play against someone else. In a future tutorial I will look at expanding our TicTacToe class so it makes a more viable game. We will do this by adding a check to ensure that two players can't place their counter in the same square (at the moment, every turn a player could choose square 1, which will overwrite each time it is selected with that players counter). We can also add in error checking to ensure that the player enters an int, and not a string or another variable when it is their turn.

We shall also look at creating a basic CPU. There are many ways to do this, but since Tic Tac Toe is very simple, we should be able to write some relatively good CPU moves that are hard coded in.

If you have any questions about the code, feel free to leave a comment below!

1. https://macromates.com/
2. https://www.jetbrains.com/pycharm/