Battleships

Difficulty: Intermediate

This exercise is meant to be challenging. It relies on understanding concepts such as variables, loops, printing output and reading input. It introduces lists and nested lists to create grids.

Not every step is explicitly described, you’ll have to work things out as you go!

# Introduction

Battleships is a classic pen and paper game usually played by 2 players. This exercise builds a simple, computer based, single-player version where the player has to guess where Battleships are hidden in a grid.

The game works as follows:

1. A square grid, of any size, is set up to contain “hidden” Battleships. The computer knows where they are, but they are not shown to the player.
2. The player is permitted a set number of attempts to find all the Battleships.
3. In a loop:
   1. The player picks a grid reference (a set of co-ordinates) to fire at.
   2. The player is then told whether they have missed or hit a Battleship.
4. The game ends when either:
   1. The player has run out of attempts and loses.
   2. The player has hit all of the Battleships and wins.

In order to create a simple Battleships game, we need to understand Python lists first. These will help us build a grid to play on.

## Lists in Python

Lists are ordered collections of values. Instead of assigning one value to a variable you can store a lots of values together in a variable. To create an list use the square brackets [ and ] to mark the beginning and end of the list, and then use a comma (,) between each list element.

Try this out on in Python, creating your own lists, printing out the whole list, individual elements in the list and adding your own.

>>> fruits = ["Apples", "Bananas", "Pears"]

>>> print (fruits)

['Apples', 'Bananas', 'Pears']

>>> print (fruits[0])

Apples

>>> print (fruits[1])

Bananas

>>> print (fruits[2])

Pears

>>> print (fruits[3])

Traceback (most recent call last):

File "<pyshell#13>", line 1, in <module>

print (x[3])

IndexError: list index out of range

>>> fruits[1] = "Cabbage"

>>> fruits

['Apples', 'Cabbage', 'Pears']

The list above has three elements. Each element of a list can be anything that could normally put in to a variable. In the example above, we’re using strings (words), but you could as easily use numbers. Notice two really important things here:

1. The first element of the list isn’t x[1], it’s x[0].
2. If you try to access an element that doesn’t exist (e.g. the fourth element of a three element list) then you will get a “list index out of range” error.
3. You can update a single element of a list.

When working with lists you will often want to do something to *all* the elements in the list. In programming, you do this by iterating over every element and run that code against each element automatically.

>>> fruits = ["Apples", "Bananas", "Pears"]

>>> for fruit in fruits:

print("I'm a fruit:", fruit)

I'm a fruit: Apples

I'm a fruit: Bananas

I'm a fruit: Pears

Note that you’ll have to type a blank line (just press return) to get the code to execute. This is because Python needs to know if there’s anything else you need to do inside the loop after printing out the message. You tell Python this by inserting a blank line.

The for keyword tells Python that we want to execute all the indented code underneath on each element of the list. This is what Python does under the covers:

1. Set an internal counter to 0 (zero).
2. Copy the value at fruits[*counter*] in to a new variable: fruit.
3. Execute the indented code block: print(fruit).
4. Are there any more fruit? If so increment the counter and go to 2. If not, finish!

## Two Dimensional Lists

If each element within a list is a list then this allows us to create a two dimensional list, or a grid, of elements.

For example:

>>> grid = [[ "a", "b", "c"], ["d","e","f"], ["g", "h", "i"]]

>>> print(grid)

[['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h', 'i']]

>>> print(grid[0])

['a', 'b', 'c']

>>> print(grid[1])

['d', 'e', 'f']

>>> print(grid[2])

['g', 'h', 'i']

|  |  |  |
| --- | --- | --- |
| a | b | c |
| d | e | f |
| g | h | I |

Each element in the grid has co-ordinates, starting at the top left corner (0,0). The co-ordinates for a 3 by 3 grid would look like this:

|  |  |  |
| --- | --- | --- |
| [0][0] | [0][1] | [0][2] |
| [1][0] | [1][1] | [1][2] |
| [2][0] | [2][1] | [2][2] |

You can try this out for yourself by putting the co-ordinates of the element you want in square brackets, like this:

>>> grid[0][0]

'a'

>>> grid[1][2]

'f'

>>> grid[2][0]

'g'

Try setting individual grid elements to a new value and printing out the grid again so you can see what you’ve done.

Now we understand how to create a grid, put values inside it read them out and change them, we can start building the game.

# Task 1: Create the game grid

A paper-based game of Battleships requires two grids:

1. A grid that the player never sees, that records whether an element contains a battleship or empty sea.
2. A grid that the player records their guesses on and, having made a guess, whether the guess hit a battleship or empty sea.

Because we can control what the player sees, we can model this in Python using a single grid.

Consider the different “states” that a single grid element can be in:

1. The player **hasn’t** guessed the element and it **doesn’t** contain a Battleship.
2. The player **hasn’t** guessed the element and it **does** contain a Battleship.
3. The player **has** guessed the element and it **doesn’t** contain a Battleship.
4. The player **has** guessed the element and it **does** contain a Battleship.

We can use different numbers to record the state of the grid:

|  |  |  |
| --- | --- | --- |
| State | Has Guessed? | Contains Battleship? |
| 0 | N | N |
| 1 | N | Y |
| 2 | Y | N |
| 3 | Y | Y |

To keep things really simple, let’s just play on a 4 by 4 grid. You can change this later, but this is good for development and testing because there’s only 16 elements to think about.

Task list:

1. Create a grid, 4 by 4.
2. Set a value for each element. No guesses have been made yet, so set each element to either the number 0 (for no Battleship), or the number 1 (for a Battleship).
3. Use a for loop to print out the grid, one row on each line.

# Task 2: Implement the Main Game Loop

The main game loop will:

1. Read some co-ordinates from the user.
2. Change the state of the board and tell the user whether they hit a battleship
3. Print the board.
4. Work out if the player has won (guessed the location of all the Battleships) or lost and terminate if so.

The next sub-tasks break this down in simple steps.

## Task 2.1: Read some co-ordinates from the user.

You’ll need to get a row and column from the user. You can use the input function to do this.

See if you can capture both the x and y co-ordinate in one single call to input and parse them out. Look up the Python split on a string function to do this.

## Task 2.2: Change the state of the board and tell the user whether they hit a Battleship

1. IF guessed this co-ordinate before, print a message saying that the user has already guessed that co-ordinate, ELSE
2. IF not guessed before:
   1. Tell the user whether they hit a Battleship or not.
   2. Update the grid to mark it as guessed, making sure not to lose whether a Battleship was there or not.
   3. Decrement (subtract 1) from the number of guesses remaining.

# Task 3: Print the board

Print out the board by taking the for loop that printed the board and put it in to a function, then call it from here.

# Task 4: Work out if the player has won or lost

Scan the board to see if there’s any remaining Battleships that the player hasn’t hit. If not, then finish the game.

To scan the board, iterate over every element in the grid, looking for a cell that contains a Battleship but hasn’t been hit.

If the number of remaining guesses is now 0, then finish the game and tell the user that they have lost.