# Certificate 1 Session 1 — Script

**Disclaimer**: Hi Bedour! Here I briefly outlined what I was planning to convey with every slide. Please feel free to add to it and to explain things in your own way. This file is meant as a an add-on to the slides, to make sure everyone is on the same page =)

[slide 1]
In the first session, in course of the Certificate 0, we got acquainted with the platform and tried coding simple rules for the game of Noughts and Crosses. Today we are going to move to a more complicated game of Battleship and to study how to make moves in a more *systematic* manner.

[slide 2]
Let us first consider an example. Here is a simple maze, and the task is to find a path through the maze, from the stating point marked as *in,* to the exit marked as *out.*

[slide 3]
We can represent the maze in a structured, machine-readable way as a Python *dictionary*. For every cell we specify a set of all other cells immediately reachable from that cell. E.g., from cell *5* it is possible to go to cells *2*, *4* and *8*. Then we put this information into a dictionary: *5* as a key, and *{2,4,8}* as a value. The curly brackets are used in Python in the definition of sets and dictionaries. And we use sets as values in our maze dictionary and not for example lists, because we do not want to order the elements.

[slide 4]
Now, given this maze, we can outline all possible paths through the maze in a form of a *search space tree.* Every node in the tree is a particular state of the maze (and the pointer inside it), and an edge between the states means that one state can be reached from the other one in one step of the pointer. Edges are obviously determined by the structure of the maze.

The Key point is that when we go through the maze, we traverse the space tree back and forth. And while we can do it randomly (as if we were making a random move in a game, remember the function *randomGuesser(gameState)* that the default bot for Noughts and Crosses uses), it is mode efficient to follow a *search strategy.* The search strategy defines how we move through the maze (or rather through the space tree). There are two fundamentally different search strategies: depth-first search and breadth-first search.

[slide 5]
*Depth-first search,* or DFS [di-ef-es], follows the strategy to stick to a path until a dead-end or a terminating state is reached and then to *backtrack* to the last seen “cross-road”.

[slide 6]
Let us traverse our space tree in a depth-first search manner. We start with the root of the tree, and until there are no alternatives to take, we simply follow the edges in the tree (or should I say its trunk =). Note that we are memorising all previously seen states, marking them as ‘visited’.

[slide 7]
Our only option to move to cell number *4, …*

[slide 8]
… and then to cell number *5,* until we reach a “cross-road” where we need to make a choice which path to follow. Unless we have some intuition/knowledge which path is more preferable,

and in our example we do not (*uninformed search)*, we choose one of the child states *non-deterministically*, or randomly.

[slide 9]
We choose to move to the cell number *2, …*

[slide 10]
… then follow to cell number *3*, …
[slide 11]
… and on to slide number *6*. A this point we reach the leaf of the space tree —or the dead-end in the maze — and we have to backtrack to the previous state, or to a previously visited state that provides alternative ways to move forward.

[slide 12]
Such state is cell number *5*, from which we now take an alternative lath and move to cell number 8. It is another "cross-road", …

[slide 13]
… and we randomly choose to move to cell number *7*. Remember that we might have as well moved to cell number *9*. The current state is a dead-end, so we backtrack to cell number *8*,…

[slide 14]
… and move to cell number *9*.

[slide 15]
We then follow the branch of the space tree and reach the exit from the maze. Well done!

[slide 16]
Let us now formalise the general strategy behind depth-first search into an *algorithm*. But before we do so, we need to make two important comments. Firstly, the general DFS works not only on trees, as in our example, but on any types of *graphs*. Secondly, as we need to memorise the states we have already visited, we need a special *data structure* in which we will keep track of the nodes already visited. A data structure we are going to use is called a *stack*. Think of a stack of books. Every time you place a new book in the stack, it becomes the top book, and the first to pick up. If you put one more book on top of the stack, the new book becomes the top one, and previous book moves one step away from the top of the stack. In order to pick up that book, you first need to take away the one on top. In computer science, this principle is called "*last in — first out"*. Very intuitive!

[slide 17]
Now we are ready to formalise the DFS algorithm. So, given a graph *G,* a starting node *n* in that graph and an empty stack, first put the starting node into the stack. Then take it as the currently top node of the stack, mark it as visited and put all nodes connected to it — or *reachable from* it — and not yet marked as visited into the stack instead of the top node. Continue the process of removing the top node of the stack and replacing it with its reachable, unvisited neighbours until the stack is empty. That's all it takes to perform the DFS search!

[slide 18]
Here is a formal representation of a stack. It is an ordered set of elements (a list or an array, depending on the actual implementation) with a *pointer* that indicates the top element of the stack. Formally, a pointer is a variable that keeps record of the cell that is currently the top of the stack. In computer science parlance, putting an element in the stack is called *to push*, and taking it from the stack is called *to pop*. And as we said, the operating principle of the stack is "last in — first out", or LIFO. You can read about the stack data structure and its various implementations in Wikipedia: https://en.wikipedia.org/wiki/Stack_(abstract_data_type).

[slide 19] *# it may be useful to go through the code line by line using the blackboard*
Now that we know how to formalise DFS, let's implement it in Python! First we define a function that will perform the depth-first search: "`def dfs():`". The function will take as input two *parameters*, a graph and a starting node: "`def dfs(graph, start)`".
We defined the input of the procedure, now let's *instantiate* two auxiliary data structures: an empty set of visited nodes "`visited = set()`" and an empty stack, which we implement as a list "`stack = []`". Now we can put the starting node into the stack, as our DFS algorithm tells us: "`stack = [start]`". For brevity we can write all the above steps in one line (see line 2): "`visited, stack = set(), [start]`". Now we are going to create a *loop*. We would like to implement the condition from the algorithm "while the stack is not empty", so we write "`while stack:`". As you know from the introductory class to Python, the while-loops follow the structure "while some boolean condition holds, do something". In Python, one can use a list as a boolean expression, so that the condition is true as long as the list is not empty. A very convenient syntax, especially for our case.
Now let's write the body of the while-loop. We want to take the top element of the stack — that is line 4, "`vertex = stack.pop()`". Here we use a predefined operation on lists in Python `pop()` that simply returns the last element from the list, removing it from the list itself. Then we want to check whether the element has not been yet visited — so we write an if-statement "`if vertex not in visited:`" (line *5*). Do not forget the colon at the end of while- and if-statements! Then we mark our node as visited, i.e., we add it to the set of visited nodes: "`visited.add(vertex)`". `add` is another predefined operation over lists in Python. Lastly, we would like to add to the stack all the nodes in the graph that are reachable from the current node and are not yet visited. Try to write a line of code that implements this step. To add several elements to the list in one go, you can use a predefined operation `extend()`.
Optionally, you can vertices in the order the algorithm processes them so that you can see how your algorithm actually goes through the graph (line *8*). You can *debug* the algorithm!

[slide 20]
Here is the full implementation of the DFS search. In line *7*, we add to the stack all elements reachable from the current vertex in the `graph` dictionary minus (set minus!) all the nodes that are in the `visited` set. We could of course check the membership of each reachable node by an if-statement, but Python provides as with a far more elegant syntax.
You can find the full implementation of the DFS algorithm in the file *maze-searches.py*. The file also contains a machine-readable description of the maze we used as our running example, and you can try running the DFS search over the maze by simply calling the DFS function: `dfs(maze, 'in')`.

# here it might be a good time to take questions about DFS, if any

[slide 21]
Now let us move to a different search strategy. As we have mentioned, there exist two fundamentally different search strategies: depth-first search and breadth-first search. *Breadth-first search*, or BFS [bi-ef-es], takes one step into all possible directions available from a given node instead of sticking to one path. Let us again look at the example.

[slide 22]
We have the same maze and the same seach tree. As before, we start at the entrance of the maze and step into the cell number *1*.

[slide 23]
As before, we memorise the cells that we already visited. Our only option to move to cell number *4*, …

[slide 24]
… and then to cell number *5,* until we reach a familiar "cross-road". As before, we do not have preferences over which path to take, so we choose one of the reachable states randomly.

[slide 25]
Suppose we choose to mode to cell number *2* first. This is not an exit state, so in the DFS case, we would follow the path from cell number *2* until an exit or a dead-end is reached. In the BFS case, we follow a different strategy. We *backtrack* to the previously visited cell (number *5*) …

[slide 26]
… and move to another of its reachable cells, i.e., cell number *8*. Again, cell number *8* is neither an exit, nor a dead-end, so we backtrack to cell number *5*. There are no more ways out of cell *5* that we have not considered, so we move …

[slide 27]
… to the first cell reachable from *5* that we have considered, that is cell number *2*, and now we check all of its reachable cells. The only possibility is to move to cell number *3*. Since there are no more cells reachable from *3*, and *3* is not an exit, we move to the other cells that are on the same *depth* in out search tree as the cell number *3*. These are the cells reachable from cell number *8*.

[slide 28]
We check cell number *7* first. It is not an exit, so we move to…

[slide 29]
… cell number *9.*

Note how the DFS search goes through the search tree *layer by layer* instead of committing to one path. This is a very handy quality of a search algorithm, because in real life we may often work with graphs that have infinite paths. DFS can be unlucky enough to get stuck in such a path, while with DFS it will never happen. Moreover, if we are trying to find the *shortest path* between the two nodes, e.g., between two cities on a map or between two people in a social network, BFS is guaranteed to find it.

Cell number *9* is not an exit, and there are no more cells on this depth of the search tree that we have not considered, so we move to the next layer,…

[slide 30]
… to cell number *6*. It is a dead-end, so we move to the last remaining state in the search tree, …

[slide 31]
… which is the exit from the maze. Hurrah!

[slide 32]
OK, now that we got the intuition of the BFS search and going through the tree layer by layer, let's talk about the algorithm behind it. In BFS, as well as in DFS, we need to memorise the states we have already visited. So again we need a special data structure in which we will keep track of the nodes already visited. However, this time we keep and access memorised nodes in a completely different fashion. Hence, we need a different data structure, namely a *queue*. When a person enters the queue, she becomes the last element of the queue, and it will bw her turn only after all the people before her exit the queue. In computer science, this principle is called "***first** in — first out"*. Not to be confused with LIFO!

[slide 33]
Now we are ready to formalise the DFS algorithm. Given a graph *G,* a starting node *n* in that graph and an empty queue, first put the starting node into the queue. Then take it as the currently first node of the queue, remove it from the queue, mark it as visited and add all nodes connected to it — or *reachable from* it — and not yet marked as visited to the end of the queue.

Continue the process of removing the first node of the queue and adding all of its reachable, unvisited neighbours to the queue until the queue is empty.
Notice how similar the two search algorithms look! The difference in their performance comes from the way we store and access intermediate results into a stack — or a queue.

[slide 34]
Here is a formal representation of a queue. Similar to a stack, a queue is an ordered set of elements (a list or an array, depending on the actual implementation) with a *pointer* that indicates the top element of the queue. Formally, a pointer is a variable that keeps record of the cell that is currently the top of the queue. In computer science parlance, putting an element in the queue is called *to enqueue*, and taking it from the queue is called *to dequeue*. And as we said, the operating principle of the stack is "first in — first out", or FIFO. You can read about the queue data structure and its various implementations in Wikipedia: https://en.wikipedia.org/wiki/Queue_(abstract_data_type).

[slide 35] *# it may be useful to go through the code line by line using the blackboard*
Now that we know how to formalise BFS, let's implement it in Python! First we define a function that will perform the breadth-first search: "`def bfs():`". The function will take as input two *parameters*, a graph and a starting node: "`def bfs(graph, start)`".
We defined the input of the procedure, now let's *instantiate* two auxiliary data structures: an empty set of visited nodes "`visited = set()`" and an empty queue, which we implement as a list, same as the stack: "`queue = []`". Now we can put the starting node into the queue, as our BFS algorithm tells us: "`queue = [start]`". For brevity we can write all the above steps in one line (see line 2): "`visited, queue = set(), [start]`".
Now we are going to create a *loop*. We would like to implement the condition from the algorithm "while the queue is not empty", so we write "`while queue:`". As we have already discussed, in Python one can use a list as a boolean expression, so that the condition is true as long as the list is not empty.
Now let's write the body of the while-loop. We want to take the first element of the queue — that is line 4, "`vertex = queue.pop(0)`". Here we use a predefined operation on lists in Python `pop()` that simply returns the last element from the list, removing it from the list itself. However, if you *parameterise* the `pop()` operation with an index number, it will return and remove from the list the element at the specified position, in our case the first element. Keep in mind that the indices of list elements start with *0*, not with *1*, e.g, the third element of a list will have the index *2*.
Now we want to check whether the element has not been yet visited — so we write an if-statement "`if vertex not in visited:`" (line *5*). Do not forget the colon at the end of while- and if-statements. Then we mark our node as visited, i.e., we add it to the set of visited nodes: "`visited.add(vertex)`".
Lastly, we would like to append to the queue all the nodes in the graph that are reachable from the current node and are not yet visited. Try to write a line of code that implements this step. Remember, to add several elements to the list in one go, you can use a predefined operation `extend()`.
Optionally, you can vertices in the order the algorithm processes them so that you can see how your algorithm actually goes through the graph (line *8*). You can *debug* the algorithm!

[slide 36]
Here is the full implementation of the BFS search. In line *7*, we add to the queue all elements reachable from the current vertex in the `graph` dictionary minus (set minus!) all the nodes that are in the `visited` set: `queue.extend(graph[vertex] - visited)`. Optionally, one can *sort* the reachable nodes with respect to their name/number. For that Python provides a predefined operation `sorted()`.
You can find the full implementation of the BFS algorithm in the file *maze-searches.py*, and you can try running the BFS search over the maze by simply calling the BFS function: `bfs(maze, 'in')`.

# here it might be a good time to take questions about BFS, if any

[slide 37]
Now let us move to our gaming platform! If you go to https://www.aigaming.com/, you can choose from a range of games to play. Last session we have looked into the game of Noughts and Crosses; today we move to the Battleships game and we are going to apply search strategies for finding the opponent's ships!

[slide 38]
Go to the online editor and select game type "battleships": https://www.aigaming.com/OnlineCodeEditor. You can read about the game in the About section on your right. We are going to play the classical 8x8 battleships game, with ships of size *5-4-3-3-2* and no land, that is, the game style number *104*.

*# if it makes sense, go through the game basics and recap the previous session info*

[slide 39]
The main part of the code is the `calculateMove(gameState)` function (line *29* in the editor). The function returns your move which is then processed by the server and is taken into account by the opponent. The opponent is then making her move, and the game state is updated both with your move and the opponent's move.
The dictionary `gameState` contains all the information about the game, e.g., what is the state of your board, the state of your opponent's board (that is visible to you), who's move it is and many more. You can see the content and structure of `gameState` by printing it: `print('Game state: ' + str(gameState))`. Note that you first need to convert a dictionary into a string, and there is a Python function `str()` for that.
You can think of `gameState` as a snapshot of the game at a given time. It reflects what's on boards and who is playing. However, it does not contain the 'historic' information, like what was your last move. For that, we have another dictionary `persistentData`. For now, the default code only stores `gameState` which is the number of moves that you have made. It is up to you what to store in `persistentData` and you can define your own keys and values.

Let us quickly go through the code of the `calculateMove()` function. It takes as input the current state of the game and outputs the next move (line *39*): `return move`. Lines *30* and *31* check whether a key `handCount` is present in the `persistentData` dictionary, and of not, set it to zero. Line *32* checks whether it is the first move when the users should deploy their ships. The ships can be deployed manually (line *33*) or randomly (line *34*). In case the ships are already deployed (line *35*), the main part of the game starts and it is time to choose the cell to shoot (line *37*).

[slide 40]
The main principle of the Battleships game is quite simple: first you randomly search for ships on the board — this is called *hunt mode*. Whenever a ship is hit, you switch to *target mode* and *search* around the neighbouring cells to sink the ship. Let is try to apply the two search strategies we just discussed to sinking ships!

[slide 41]
Consider the following 5x5 board of an opponent. In that board, we have already opened three empty cells without ships (3 misses), and we finally hit a part of a ship in row *4*. Now we enter the target mode and start searching for the remaining parts of the ship.

[slide 42]
According to the BFS strategy, we first consider the cells adjacent to the cell with the red dot. And even is we are lucky, and the cell above which we consider first also contains a ship…

[slide 43]
… we will not follow the promising path and instead consider the other adjacent cells of the first cell.

[slide 44]
In the DFS mode, we will first consider one of the adjacent cells, and then the adjacent cell of that cell and so on, as if following a path.

[slide 45]
The problem is that even if there is no ship in the adjacent cell…

[slide 46]
…we will still follow that direction.

[slide 47]
The optimal solution is to adapt DFS to our needs and to introduce *pruning*. We first consider only one path, but when we see there is no ship cell in the direction, we consider it a dead-end, *prune* this path…

[slide 48]
…and move to the next path. If the cell in that direction contains a ship…

[slide 49]
…we continue in that direction until a dead-end. Thus, we choose to use DFS search for the target mode, and in addition to a stack, we will use a pruning condition.

[slide 50]
Now we are ready to implement search in the context of the Battleships game. We are going to mostly work inside the `calculateMove` function, and more specifically, in the `else`-statement that brings us to the moment in the game when the ships are already deployed and the shooting starts.

[slide 51]
We are ready to code!

[slide 52]
Let us discuss what has to be done in order to improve the target mode (and in fact, create it, since in the default implementation the bot is always in the hunt mode aiming randomly). On a high level, we need (*1*) to be able to memorise our previous moves, (*2*) to access them and to check whether these moves were hits or misses. Moreover, depending on the situation, we need to (*3*) be able to alternate between hunt and target modes (and to know when to do so). Finally, we need to (*4*) perform the DFS search with pruning.
This is the high-level specification of what we need to do. Obviously, there are multiple ways how one can actually implement it, especially given the fact that integrating DFS into the bot framework is *not straightforward.*
Disclaimer: below is one of the possible implementations, but it is *not the only*, and it can be *further optimised*.

[slide 53]
A move is represented in the game as a dictionary of the form `{'Row': 'Letter', 'Column': number}`, which you can check by simply printing the `move` variable: `print(move)`. Hence, we will save moves in the `persistentData` structure as a dictionary.

[slide 54]
The first update to the code that we do is in lines *30-31*: instead of storing a key `handCount` in `persistentData`, we now store `previousMove` (by simply renaming the key).

Secondly, we change the way we update `persistentData`: (a) we flip lines *36* and *37* so that we compute the next move first, and (b) we update `persistentData` with the move. You can see what is going on with your data structures by simply printing them (lines *38-39)*.
Lastly, we need to update line *40* that prints current state of affairs after every move of ours. The `persistentData` dictionary does not contain `handCount` any more, so we need to remove it. You can form any print statement you like, e.g., printing the current move, `gameState`, `persistentData` etc. I will just leave the information about the move for now.

[slide 55]
So we implemented the step of saving a move. Now we need to access what we saved in the previous move (line *36*): `previousMove = persistentData['previousMove']`, — and in case there was indeed a previous move (line *38*) — `if len(previousMove) > 0:` — to check whether it was a hit or a miss. Checking can be done in the body of the main function `calculateMove`, or it can be put into a separate function, in order to declutter and modularise the code. It is up to you, but we will write a separate function: `checkHitOrMiss(previousMove, gameState)`. The function takes as input two dictionaries, the previous move and the game state, and it outputs a boolean value: true if the move was a hit, and false otherwise. We save this value into a variable `isHit`: `isHit = checkHitOrMiss(previousMove, gameState)`. Again, you can check every step of the way with print statements.

[slide 56]
Let us go through the code of the `checkHitOrMiss` function. First we get the opponent's board from the game state (line *50*): `board = gameState['OppBoard']`. Then from that board we access the cell to which the previous move was made.
One technicality you will need in order to check the previous move's outcome is to convert a move dictionary into a pair of indices that we can query the opponent's board with. For example, from a move "row *F*, column *2*" we need to get indices "row *5*, column *1*". Why is that? The rows of our board are labeled with letters *A* to *H*, and columns are labeled with numbers from *1* to *8*. However, the board is represented as a *list of lists*, and elements of any list in Python are indexed *from 0, not from 1*.
So, to convert the column number to the column index, we simply need to subtract *1* from the column number (in order to shift indices from *1-2-3-…* to *0-1-2-…*). In line *54*, we access the column number from the move dictionary and subtract one: `column = move['Column']-1`.
Converting the row letter is a bit more complicated. To save time now, you can simply use the following command: `row = ord(move['Row']) - 65` (line *52*).
[Detailed explanation: In Python, one can convert a letter into its corresponding Unicode code with a built-in function `ord()`. You can see the codes here: https://unicode-table.com/en/. The code for capital letter *A* is *65*. Hence, in order to "shift" these codes to *0*, we need to substract *65*.]
Now we have the indices for row and column, and we can access the cell on the opponent's board (line *57*): `moveValue = board[row][column]`. We first access the relevant list that corresponds to the row, and then access the entry in the list that corresponds to the column.
The one remaining step is to check whether the value in the cell is *H* (hit) or *M* (miss) and to return *true* or *false*, respectively. We do this via an `if-else` statement (lines *58-61*).

[slide 57]
Now let's put everything together. The logic of our program is as follows:
- if persistent data does not already contain the information about the previous move, add the corresponding key to the `persistentData` dictionary (lines *30-31*);
- we will also keep in the persistent data the information whether or not we are in the target mode, in the form of a boolean value for the key `targetMode`; initially we set the value to be `False`, since we start the game in the hunt mode (lines *32-33*);
- if it is the first round of the game, deploy your ships, manually or randomly (lines *35-37*);

- otherwise (line *38*) access the previous move information from `persistentData` (line *39*);
- if there was indeed some previous move (line *40*), check whether it was a hit or a miss (line *41*);
- if it was a hit and we are not yet in the target mode (line *42*), set the value for the target mode to `True` (line *43*);
- now we check whether we are in the target mode (line *44*);
- if yes, we will perform our DFS search with pruning (line *46*);
- otherwise (line *47*) make a random move (line *48*);
- if there was no previous move (line *49*), also move randomly (line *50*);
- after the move is generated (randomly or via search), save it to persistent data (line *51*);
- finally, return the move (line *53*).

[slide 58]
It remains to define the `searchNeighbours` function, which is probably the most complicated part of the implementation. Let's go through the code step by step:
- since we need to keep in memory two additional structure — the `visited` set and the `stack` — we will keep them in `persistentData`;
- if `stack` and `visited` are not yet created, initialise them in `persistentData` and create variables `stack` and `visited` inside the function (lines *58-64*);
- then add `previousMove` that we pass to the `searchNeighbours` as the input parameter to `visited` (line *66*);
- if the previous move was a hit (line *68*), get the neighbours of that cell (line *71*); for that we use a helper function `selectUntargetedAdjacentCell`. It takes as input row index (line *69*) and column index (line *70*), and returns a list of pairs of indices, e.g., *(0,5)*;
- **NB!** we perform the above operations only if the previous cell was a hit, meaning we do not get the neighbours for missed shots — this is exactly our pruning condition!
- since in the rest of the code we work we row letters and column numbers, we need to convert indices to the original dictionary format of moves (lines *72-75*), and for this we use another helper function `translateMove` (line *74*);
- when we bring neighbour cells to the required format, we can update the stack as we did in the basic implementation of DFS (line *76*);
- then if the stack is not empty (line *78*), get the move from the top of the stack (line *79*); there is some syntactic sugar involved, namely the built-in `eval()` function that takes a string representation of a dictionary and converts it into a real dictionary;
- we then return this move (line *80*);
- if the stack is empty, and the search is over, we "reboot" the values for `visited` and `stack` in `persistentData` (lines *82-83*), and we "switch off" the target mode (line *84*);
- we then make a random move (line *85*) and return it (line *86*).

Happy playing!