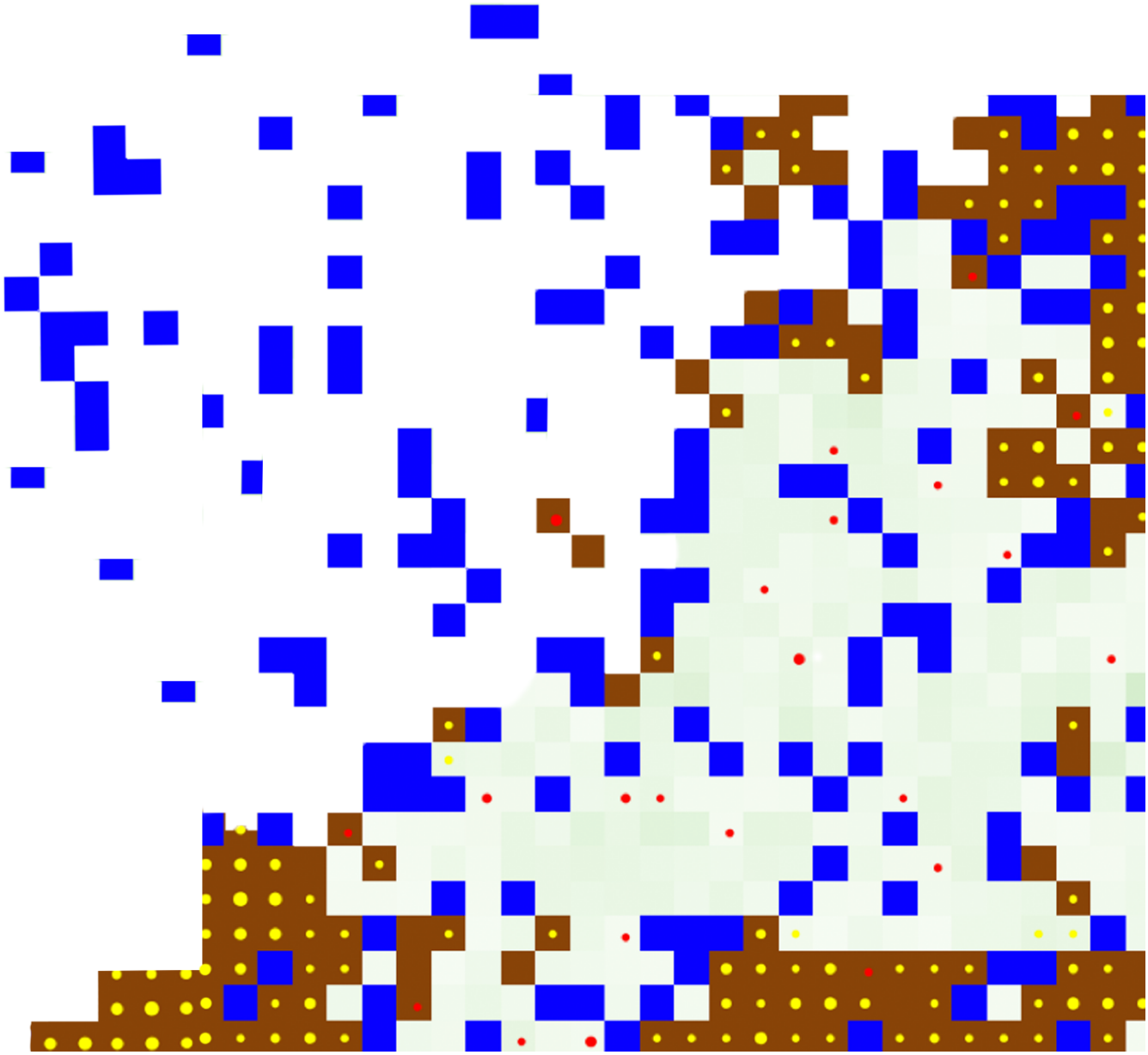
Planisuss

Matteo Salami *513974* - Pietro Saveri *524921*

509477 - Computer programming, Algorithms and Data str., Mod. 1
Project 2022/2023



Every morning in Planisuss, a Erbast wakes up. It knows it must outrun the fastest Carviz or it will be killed. Every morning in Planisuss, a Carviz wakes up.

It knows it must run faster than the slowest Erbast, or it will starve.

It doesn't matter whether you're the Carviz or a Erbast —when the sun comes up, you'd better be running.

Unless you are a Vegetob.

Planisuss proverb

Table of contents

Goal of the project

The **main goal** of the project was to challenge the students, and that would be ourselves, to use the python programming language to create a simulation of a fictitious planet named '***Planisuss***' constituted by a single continent that hosts three living species, the **Vegetob** species that would be the equivalent of the vegetation on our planet, the **Erbast** species, a plant based group of individuals that circle around the planet eagerly searching for the next Vegetob to consume, an last is the **Carviz** species, a carnivore species that hunts tirelessly for its next prey.

To add to the challenge was the obligation to use a not yet known Python library, that would need to be learned from scratch, called *matplotlib* which intended use was for the graphic and interactive portion of the project.

The planet as a grid

The first problem to be solved was to actually create the grid that would represent our fictitious planet. the grid is composed, as specified, of 'NUMCELL' * 'NUMCELL' squares, where each square represents a cell that is actually a class with different attributes.
These cells are the fundamental structure of the little continent and everything, from spawn up to movement and hunting revolves around them.

The implementation of the grid is essentially a square matrix where each row is a list containing as many cell classes as indicated by 'NUMCELL' that are appended to the list with a simple iterative for loop.

Each cell object has its own material and its own coordinates that are given upon the creation of the object. For it to be a continent it needs to be surrounded by water, and that is why an if statement checks each and every cell, especially its coordinates, automatically choosing 'water' as the material when the coordinates match with those of the cells at the border of the map, otherwise the choice between 'water' or 'ground' is randomized.

Generating new life

The second task on the list was to make the plants and the creatures actually populate the newly created continent. The 'spawn' function is very straightforward, using two for loops it iterates through the whole grid and once it finds a cell object with the attribute material equal to 'ground' it goes on to the spawning portion where, again using the random library and more in particular the *random.choices([0, 1], weights=[0.4, 0.6])* method, it chooses between 0 and 1, the two numbers don't have the same probability of being chosen, because the latter is given by manually chosen weights.

Both existing creatures can spawn on the same cell along with the Vegetob species that has a much higher spawning probability. When spawning both Erbasts and Carviz are appended respectively to

the ‘erbast’ and the ‘carviz’ lists, both attributes of the cell class, the decision of implementing a list will be better discussed later when prides and herds will be introduced.

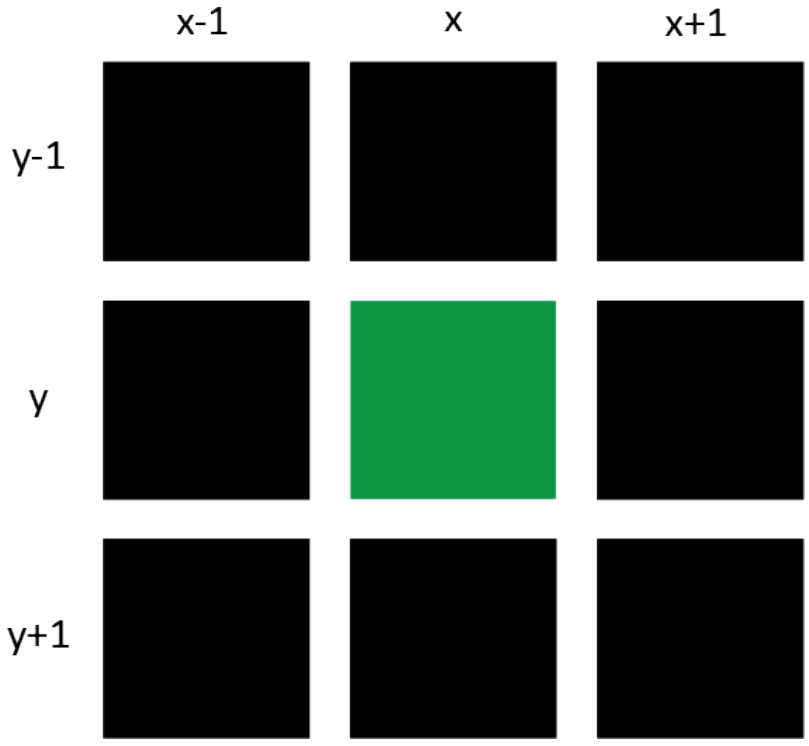
Some more detailed information about the creatures is necessary before moving on with the report. Both Erbasts and Carviz are classes each with their own attributes, almost all of which they have in common. Arguably the most important attribute for both classes is ‘ENERGY’, each creature is randomly given an initial energy, in the ‘spawn’ function, chosen between 0 and 76 using the numpy library and again more in particular the *numpy.random.randint(0, 76)* method which does not include the value 76 in the possible choices. Utilizing the same method also the value of the ‘SOCIALATTITUDE’ attribute is chosen, this time the value can be either 0, making for a non social creature, or 1 symbolizing the opposite, and we will get to the purpose of this attribute in a few lines.
Last but not least is the ‘AGE’ attribute which is basically self explanatory, its value is increased by one for every new day, and the ‘MAXLIFE’ attribute which is randomly selected upon initialization with the
method numpy.random.randint(1, 100). It will be ‘death_birth’ the function responsible, along with other things, for checking that no creature lives past what ‘MAXLIFE’ states.

The first steps

Now that the grid was all initialized and the creatures were also spawned, it was time to cope with the challenging task of gifting our planet with life, it was time to make the creatures move and start their journey for survival.

Neighbors

The movement of the living entities revolves entirely on what one can see and what are its needs.
It is 'neighbors' the function responsible for what one can see, and it is also at the foundation of all the other functions that will be described from now on. The goal of the function is simple: given a particular cell, let's call it current_cell, and its coordinates it will create and return a list of all the neighboring cells selecting amongst them only the ones with material == 'ground'.
The neighboring cells are the cells that directly surround the current_cell, so each cell has between 0, if it is completely surrounded by 'water', and 8, if it is completely surrounded by 'ground', neighbors.
The 'neighbors' function, given a particular x and y positions, that in our case are the and the col of the grid, creates a list of tuples (x1, y1), which are simply the coordinates of certain cells, using the following for loops *(for x1 in range(x-1, x+2) for y1 in range(y-1, y+2)* which will later be skimmed in order to return only the cells with material = 'ground' that are not the cell (x, y).



Moving solitary creatures

Once we had the neighbor function operational, we created two new functions 'check_neighbours_erbast' and 'check_neighbours_carviz' whose main purpose is to check for each and every solitary erbast or carviz, that hadn't already moved, amongst the neighboring cells which satisfies best the creature needs.
To check whether the cell hosts a solitary erbast, and the same is valid for carviz, it is sufficient to check the length, using *len()* a built-in python method, of the erbast list mentioned above, which is why we chose to implement the latter. If the length of the list equals to 0 it means that there are no erbasts on the current cell and so the program skips it in favor of the next one, while if there are 2 or more erbast objects in the list it means that there is a herd and it will be considered in another function called 'move herd'. As already stated the 'check_neighbours_erbast' and 'check_neighbours_carviz' only deal with solitary living entities so the length of the erbast or carviz list has to be 1 for it to be considered in its respective function.
To check if the creature hasn't moved already we created a 'moved' attribute for the Erbast and Carviz classes that was changed to true every time a movement happened and was changed back to False at the end of the current day by a loop iterating through the whole grid.
Once a solitary not yet moved creature is found two options are considered and confronted. For the erbasts for example, the value of the 'oncell_vegetob', i.e. the vegetob on the cell where the erbast is, is stored in a variable to be used later. Then comes another if statement looking into the energy of the erbast which must be greater than 5 for it to be able to move. If that is the case the program finds, among the neighbors, the cell with the highest vegetob value and also that does not host any element of the carviz species, and it does it utilizing an auxiliary function called 'get_max_vegetob_cell' which uses a for loop to iterate the neighbors and the *max()* method to get the one with the highest vegetob value.
In the auxiliary function there also is a while loop which is the one responsible for discarding the cells hosting at least a carviz.
Now if the 'max_vegetob_cell' has a vegetob value greater than 0 it is confronted with the 'oncell_vegetob' value, while also keeping in mind the energy it costs to move from one cell to the other.
The more remunerative in terms of energy is the one that will be chosen and the value of the vegetob of the chosen cell will be transformed to 0, symbolizing that it was eaten, and the erbast will be moved.

Placing some limits

Moreover a variable 'MAX_ENERGY_ERB' was initialized indicating the maximum value of the energy of any erbast at any point in time. Because of this constraint the program also checks if the energy gained from the more remunerative vegetob around would get the erbast to exceed the value of 'MAX_ENERGY_ERB' in which case gets the energy back into the limits. This prevents creatures from getting overly energetic as the days go by making for a more balanced battle between species. The same implementation holds also for the carviz species but in this case the variable is called 'MAX_ENERGY_CAR'.

Regarding the 'check_neighbours_carviz' function, is more of the same meaning the idea is to find the most remunerating erbast, in terms of energy, amongst the neighbors again using the 'neighbors' function and then storing the cells containing at least one erbast in a list called 'erbast_cells'.
From erbast cells, if there are any, is selected the 'max_en_cell' with the use of the *max()* method and an auxiliary function 'tot_group_en', at which we pass *decision= 'erbast'* or *decision= 'carviz'* depending on who we need the energy of , that returns the sum of all the energy of the erbasts, or erbast if it is only one, populating the cell.
Also in this case the energy of the erbasts, or erbast, on the cell is stored in a variable 'oncell_herd_energy', which is compared with the 'energy_herd' variable storing the sum of the erbasts, or erbast, energy hosted by the 'max_en_cell'.

The main difference between the 'check_neighbours_erbast' and 'check_neighbours_carviz' functions is the fact that when a carviz wants to eat a lonely erbast or a herd of erbasts, it must fight. The fight is simple and it is based on the energy of the two parties involved. The carviz, being the predator, has an advantage which we called 'PREDATOR_ADVANTAGE', so for the erbasts, or erbast, to resist the attack they must have a total sum of energy superior to the energy of the carviz summed with the value of the 'PREDATOR_ADVANTAGE' variable.
In case the erbasts, or erbast, resist the attack they will lose some of its energy due to the defensive efforts.

Later in the report a section will be added talking about how the balance between the strength of the two species can be handled and changed and the difference it brings to the outcome of the simulation.

What about boredom?

We needed to solve one more problem before moving on with the writing of the code; what if the carviz neither the neighboring cells to the cell the carviz is standing on, nor the actual cell it is standing on are hosting any erbasts? Then the carviz gets bored, and that is the reason why a 'bored' attribute was added to the Carviz class which is essentially a counter that gets increased by one every time the situation described above presents itself. When the counter reaches a certain value, that is the value of the 'BOREDOM' variable, the carviz checks if there are prides, or lonely carviz, around it to be joined and if the 'SOCIALATTITUDE' attribute value is equal to 1 it will join and the value of 'bored' will be set to 0.
If the carviz is a non social creature then its fate is unchangeable, it will remain bored until either death or a wandering erbast come and save him.

Movement of groups of creatures

The next challenge on the list was the one regarding the movement of groups of creatures, more in particular the movement of Herds, groups of 2 or more erbasts, and Prides, groups of 2 or more carviz.
To handle this next challenge we created two functions: the first one, 'move_herd' is used to identify and move an Herd while the second one, 'move_pride', is used to identify and move a pride.
The logic behind the two functions is essentially the same as for their version for single creatures, 'check_neighbours_erbast' and 'check_neighbours_carviz' to be clear, the major difference is that now for a cell to be considered it has to have at least two entities of the same species living on it that haven't yet moved.
More in particular, for 'move_herd', the initial step in the code's evaluation is to determine the energy allocation for each individual member of the herd based on their consumption of the vegetob present in the herd cell. This value was saved in a variable called 'oncell_vegetob', in order to be reused later.
When talking about a group of creatures, in this particular case a herd, there is an additional constraint to keep in consideration that involves the 'SOCIALATTITUDE' attribute of the creatures forming the group.

Moving only social creatures

Only social creatures, the ones with 'SOCIALATTITUDE' equal to 1, will move along as the group moves, and we called this part of the herd 'social_part_of_the_herd' which is simply a list containing all social creatures created using an auxiliary function called 'get_social'.
Then again the neighboring cells get involved, more in particular the neighboring cell with the highest vegetob value. If such a cell exists, meaning if there is any cell with vegetob value greater than 0, the energy gain for each erbast in 'social_part_of_the_herd' is confronted with 'oncell_vegetob'.
If it is more convenient for the social part of the herd to move, meaning it would gain more energy, then it is moved and the energy of each component is changed accordingly, but what to do with the 'non_social_part_of_the_herd'? We decided to disperse each component of the 'non_social_part_of_the_herd' singularly in the neighboring cells using a while loop where using the pop() method we were able to consider and move creatures one by one.
It is important to note that a cell may be completely encircled by water; in such an instance, the list of neighbors would remain empty, thereby resulting in the death of all erbasts residing on that island, both social and non social.
In the case where the cell, amongst the neighbors, with the highest vegetob value, the 'max_vegetob_cell' as we named it, has vegetob value = 0 or the comparison between the 'max_vegetob_cell' and the 'oncell_vegetob' is won by the second one, the herd eats the vegetob that is on the cell it is standing on and a variable, 'social_stay', which is initialized to *False* every time a herd is found, is changed to *True* indicating that the social part of the herd hasn't moved, thus there is no need to disperse the non social part.

The function ‘move_pride’ works similarly to ‘move_herd’ but still there are some differences between the two.
The main idea is the same, we need to find the most remunerating erbast, or herd, to be eaten, either it is in one of the neighboring cells or on the cell the pride is currently on. Again the social and non social part of the pride are considered and treated in the same way we treated the social and non social part of the herd, again the ‘social_stay’ variable was utilized.

The struggle for supremacy

The main differences between the two functions are two. The first one regards the fact that if a pride moves to a cell which was already occupied by another pride a fight will take place. In order to determine the winner we created an auxiliary function called ‘pride_fight’ which works in a simple but effective way. The two lists containing the two prides are passed into the function which begins by sorting them, using the sorting method and r*everse = True,* in order to have at index 0 the carviz with the highest energy value, which we called ‘champion’. Then using a while loop the champions of the two prides are confronted, in terms of energy, and the one with the highest energy value wins. The winning champion loses some of its energy due to the fight, while the other champion is removed from its pride and a new champion is selected. The process continues until one of the two pride lists is empty at which point what remains of the winning pride is returned and the cell carviz list is changed accordingly.

A bored pride

The second difference involves again the ‘bored’ attribute of the Carviz class, we felt it was unnecessary for erbasts since it will never happen that a erbast gets “bored”.
Essentially a variable called ‘active_pride’ was implemented and initialized to False, every time the pride actually did something, meaning it moved to eat for example, the variable’s value was changed to *True*, but if this never happened during a day, meaning the ‘active_pride’ variable stayed *False*, then the pride was bored on that day and, if there exists a social part of the pride since it is the part that would move, the ‘check_bored_pride’ function is called. The function sorts the ‘social_part_of_the_pride’ list in order to have at index 0 the most energetic Carviz, which becomes the ‘leader_car’. The function then checks the ‘bored’ attribute value of the ‘leader_car’ and if that is >= than ‘BORED’ then the social part of the pride is moved to a new randomly chosen cell.

Growing with time

Another very important function is the ‘grow’ function which was necessary to deal with two constraints, the first one was that the vegetob needed to grow by a certain amount every day on every cell that has no erbasts on it, we chose to increase its value by one for every day that passes. The second constraint regarded the aging of the creatures, ’AGE’ is increased by one every day for every creature, and every it reaches a value that is a multiple of 10, and we checked it using the ‘%’ operator, the energy of that creature is decreased by the value of the variable ‘AGING_ERB’ in case of an erbast and ‘AGING_CAR’ in case of a carviz.

A matter of life and death

The last function that is called during the “day” is the function called ‘death_birth’ whose name already reveals much of its purpose. Again we loop through every single cell of the grid and once we find a cell with erbasts or carviz on it some checks are performed. The first check involves the variable ‘MAX_HERD’ or ‘MAX_PRIDE’ that is essentially the variable stating what is the maximum number of components for a group of creatures, if the number of components of the group is greater than the value of the variable, by using the slice operators,the second half of the list is removed and only the first half survives.
Then there are two more checks to be performed. The first is concerned with the ‘MAXLIFE’ attribute mentioned above; once a creature's age reaches the value of ‘MAXLIFE’ the creature, symbolically, dies.
When a creature dies of age it generates a prole of two children both with the same energy, which is equal to half the energy of the dead parent, which are randomly appended to the erbast or carviz list of a neighboring cell. The second way a creature could die is if its energy reaches 0 in which case again a prole composed of two children is generated, this time with a randomly chosen but still equal energy, that are again randomly appended to the erbast or carviz list of a neighboring cell.
The last way a creature could die is if all of the neighboring cells with material = ‘ground’ had vegetob value = 100, death by overwhelment.
In order to check if this is the case we store all the vegetob values of the neighboring cells in a list which is then passed to the ‘all_elements_are_equal’ function which checks if all the cells have the same vegetob value and returns True if they do. Then if the function returns *True* and the vegetob value of any component of the ‘vegetob_values’ list is 100, we choose to check the first one but it is indifferent, then the creature's list of the circled cell is cleared using the *clear()* method.

The Graphic Portion

For the data visualization we used the *matplotlib* library, after setting up the axes and the main figure we had to import the data from our main file. It is necessary to create a few functions to interpret all the data.

Setting up initial data

Using the 'set_ground_water_values' we were able to convert the material of each cell into a value, 600 for ground and 500 for water. In this part we couldn't use the *range (1, NUMCELL-1, 1)*, which was used almost all of the time, we had to iterate through the whole grid, because it is important to also check the borders.
We implemented two functions, 'count_erbast' and 'count_carviz', to construct a list, called 'list_erbast' or 'list_carviz' which essentially is a list where each element represents the number of erbasts or carviz in a particular cell, which is simply the length of the erbast or carviz list of the cell. The two functions use the two lists to create a grid using the *numpy.array().reshape()* method, which is then returned.

Visualizing data

Now that all the data are well structured for the visualization, we have to decide how to represent them.
The first idea was to use a colormap of greens and edit it to add different colors, like brown or blue.

The method consisted in creating a new color blue in RGBA with the *np.array* method, then setting this value to be the last in the colormap, using this tecniche we were able to identify a water cell and set a range of the colormap using a min_value and a max_value, to represent the cell.

```
greens = plt.colormaps["Greens"]
newcolors = greens(np.linspace(0, 1, 256)) #devides the colormap in 256 equal parrts
blue = np.array([55 / 256, 77 / 256, 196 / 256, 1]) #color blue in RGBA
newcolors[255] = blue #set the last value of the cmap to be blue

cmap02 = ListedColormap(newcolors) #update the new cmap
```

After a few considerations this method was very complex and not easy to use so we decided to adopt a different strategy.

To represent the vegetob value a colormap of greens is perfect, we can normalize it in a range of 0-100 and we are done.

To represent the water cells and ground cells we used the usual for loops to look for the value 500 or 600, an 'img_array' is set to a certain color using the RGBA values, the 'img_array' is a variable created with *numpy.zeros* who creates *NumPy* array filled with zeros and every element of the array has four values, in our case RGBA.

```
#set up an array of zeros with dimention row, col, every element has four values, in our case RGBA
    img_array = np.zeros((rows, cols, 4))  # 4 channels (RGBA)
    for row in range(rows):
        for col in range(cols):
            if grid_ground_water[row][col].value == 500: #found a water cell
                img_array[row][col] = [0, 0, 1, 1] #set up the color of the cell, using the RGBA values, Blue, with opaticy at max

            if grid_ground_water[row][col].value == 600: #found a ground cell
                img_array[row][col] = [0.54, 0.27, 0.07, 1] #set up the color of the cell, using the RGBA values, Brown, with opaticy at max

                #increment the number of ground cells found
                NUM_GROUND_CELLS += 1

            if grid_ground_water[row][col].vegetob:

                #if a cell with vegetob is found, use the colormap of greens to rappresent it
                img_array[row][col] = cmap_greens(norm_greens(grid_ground_water[row][col].vegetob)) #the values of the vegetob need to be normalize from 0 to 100

                #update he number of vegetob presents in the all grid
                VEGETOB_NUM += grid_ground_water[row][col].vegetob
```

We used 'terrain_artists' to show the data in the figure, with *terrain_artist.set_data(img_array)* the array of dimension row, col is represented.

To draw the erbasts and the carviz we had to use 2 markers, we decided to represent both species shaped as circles whose radius changes in relation to the number of erbasts or carviz in the cell. (The carviz marker is slightly moved the the bottom right corner, so both markers are visible if on the same cell)

Daily update the grid

Using the *scatter* method we create 2 artists and initialize the size of both to 0.
These artists will be updated in the 'update_plot' function, inside the function for each carviz and erbast we create 3 lists, one for the x coordinates, one for the y coordinates and the last one for the number of creatures. For the entire grid we check if there is a creature, append his position into the x and y lists and append the number of creatures / 10 in the other list.
Using this way we have all the data we need to update their position and size, position is set using the s*et_offsets()* method, implemented in the *matplotlib* library, and the x y values, while size is set with the number of creatures in the cell, it is important to implement a 'scaling_factor', that is calculated by dividing 1 by the maximum between rows and cols, in this way the markers will always be propositioned with the number of cells that we have. If we didn't have this 'scaling_factor', every time we changed the value of 'NUMCELL', the marker wouldn't adapt to the new size of the cell.
For the title we decided to write the number of days, months, years in real time, this was possible thanks to the 'update_time_units' and 'format_time' functions.

A few useful interactive buttons

We wanted to make the user experience interesting by adding pause, resume and speed control buttons. Using the matplotlib.widgets library we created 4 buttons, 'pause_button', 'resume_button', 'speedup_button' and 'speeddown_button', each one linked to its own function to stop, resume and change the speed of the animation. Fortunately matplotlib implemented a function to stop or start an animation so this task was trivial, for the speed we had to create a function that would change the interval of the animation, but if the interval is equal to 0, the animation stops, that's why we added a check: if by clicking the 'speedup_button' the speed reaches the value 0, then the speed is not changed and a label "Maximum speed attained" appears on the screen, so in that case the animation can only be slowed down.

Speaking of animation, all the simulation has been animated using the matplolib.aniamtion.FuncAnimation and everything is updated with the 'update_plot' function.

Population graph

To represent the population graph, we initially added it as a second window, but we soon realized that having two separate windows was not only inconvenient but also memory-intensive. Using *fig, (ax, ax_population) = plt.subplots(1, 2, figsize = (10, 5))* we divide one single window into 2 figures, one for the main world animation and another for the population plot.
To plot the population data we needed to count the number of erbast, carviz and vegetob during the day and insert them in a list each, using the 'count_creatures' function we set a counter for the list of creatures that we have, then using the append function we filled the 'LIST_ERB'. For the vegetob it was a bit different, we created a variable 'VEGETOB_NUM' that increases for every time there is a vegetob, then we had to count the number of ground cells and divide the 'VEGETOB_NUM' by 'NUM_GROUND_CELLS'.
After this, the difficult part was done, using the *ax.plot* method we represented each list, giving them a color and a label, we then added a legend to better explain the data.
After plotting the data and the simulation, we found the need to improve the visual aesthetics. We opted for a gray background to enhance the visibility of the yellow erbast color and set the button color to white with a gray hover effect to maintain the theme. Additionally, we chose to remove the ticks and numbers from the main simulation graph for a cleaner appearance.

The continent is in your hands

While coding the simulation, we came to understand the significance of the variables like for example the ones that control the maximum energy achievable by each erbast or carviz, as well as the 'PREDATOR_ADVANTAGE' and more in general the overall difficulty in creating a balanced simulation. Modifying these variables has a profound impact on the fate of the creatures. To make the simulation even more interactive we decided to implement a slider, an easy and intuitive way to control every variable mentioned before, using eight buttons to select the desired variable, the user is able to control each one of them.

We utilized the well-designed, fast, and user-friendly *PyQt5* Python library to create the slider and the buttons. During the process, we encountered some challenges: initially, we intended to implement four sliders, one for every variable. However, we discovered a bug in the *PyQt5* library where, once a slider was set to a position, the next one would jump to the last-used slider value. Initially, we suspected there was an issue with our code, but after some research, we determined that it was a bug within the library itself.

To avoid this problem we opted to use eight buttons that will edit not only the text under the slider, to show which variable is being modified, but also the actual variable in the main code.

Thanks to this implementation the user is able to interact with the simulation and change the values to get a fairer, or less fair, fight between the creatures.

Possible improvements

The code of course has a lot of room for improvement and we wanted to underline the ones that came to mind during the writing process.
The first one should be a relatively simple implementation, which in our opinion would help make the code faster and more efficient. When traversing the whole list we are considering both cells with *material = 'ground'* and cells with *material = 'water'*, while if a way was found to not consider the cells with *material = 'water'* ,when for example we look for erbasts to move, it would mean less cells to be checked making for lower time complexity.

A second possible improvement is concerned with giving creatures some more intelligence about the environment around them, meaning that instead of considering only the cells that directly surround them as neighbors we could also consider the cells that are at a higher distance from the current cell, in other words we could consider checking the neighbors of the neighbors.
In this way boredom for example would be less of an issue and also both erbasts and carviz would benefit from the change since they could look for the optimal cell to move to amongst a wider range of cells.

On the other hand it is important to note that this would highly increase the memory and time complexity of the program.
Moreover the hunting capabilities of the carviz species could be enhanced by implementing an algorithm that finds the shortest path from the carviz position to the closest erbast, for example the *Dijkstra's algorithm* that with *Breadth First Search* would be perfect for the task.

Another possible addition, which was not implemented because of the already present struggles of the carviz species, is the possibility for prides to fight each other even if they are not on the same cell. To carry out the fight we would have again used the 'pride_fight' function so it would have been just a matter of implementing a new function for checking if there are other prides in the neighboring cells.

The last two improvements, which are less functional and more aesthetic, are the following: adding the possibility of performing a cell inspection where the user could see what was actually happening on the inspected cell. While the second one, and this would be purely aesthetic but as they say you should also please the eye, would be to change the appearance of carviz and erbasts from circles to actual images or drawings.

Tools and references

Of course during the writing process we had some doubts and problems, and as programmers suggest we searched for the solution of the issues online, on websites like stackoverflow, we are unable to remember and list all the time we asked for the web advice. Also friends and our colleagues were extremely helpful in some occasions.

Link related to *PyQt5* slider bug: https://bugreports.qt.io/browse/QTBUG-98093

Possibile C++ implementation to solve *PyQt5* slider bug: https://stackoverflow.com/questions/69890284/qslider-in-qt-misbehaves-in-new-macos-monterey-v12-0-1-any-workaround/69890285#69890285