Explanation of the Code Behind Our Pokémon Database

**Purpose**

There are literally hundreds of databases online about Pokémon. Nevertheless, ours was intended to stand out among this homogenous mass. Ours has a simple but ground-breaking feature. Ours has cuteness values.

Every single pokemon in our database, aside from the usual data about its name, abilities, type, etc., has also been assigned a “cuteness value”. One of the team members looked at all 898 pokemons, spanning the 8 generations released thus far, providing an “*objective”* judgement of their “cuteness”.

Using a combination of Python, SQL, PHP, HTML and CSS we could mould our vision into form.

**Webscraper**

The first thing we had to do, is collect the data for our database. By programming a webscraper in Python using ‘BeautifulSoup’ from the module ‘bs4’. What this does is essentially copy the entire source code of the html page, such that it can be sifted through. However, to actually “enter” the websites using Python we use ‘urllib.request’ from the module ‘urlopen’. These are the two preliminary modules we need to start webscraping, allowing us to “scrape” all the data we need from publicly available websites. In hindsight, it would probably have been far easier to simply download some of the tables containing all this data that are available online, collating them into one. Then again, this approach seemed more fun, giving us an excuse to use Python.

Now that we have the entirety of the source code, getting the data we need is only a matter of filtering through the text. The source code is not stored as a string, but as a ‘BeautifulSoup’ object, enabling us to use all the methods that follow along with that specific type of objects of that class. The initial page we go through, pokemondb.net/pokedex/national, contains all pokemon and links to individual pages with each of them, from which we can get all the relevant data. We first access this aforementioned website, collecting all the links to the individual pokemon, and then we make a request to each of these pages through a for loop.

Making use of these methods that objects of the class ‘BeautifulSoup’ has, and also a lot of the .find function in Python, we go through and collect all the relevant data from these individual pages. [Please note, the following part of the explanation relates to code and a file provided by our teacher to help us with this project] The data we need is identifiable by certain characteristics. Something like the national ID of the pokemon, to give an example, is always surrounded by <td><strong> and </strong></td>, so we know where to locate it. This is done for each data point and all pokemon, and then pasted into a csv file. What we need, however, is not a csv file containing this data, but an SQL table.

**Creating the SQL Table**

The way in which we would create the SQL database is by printing out the entire command necessary for 1) creating the SQL-table itself and 2) all the insertions for the pokemon. The command to create the sql-table itself, specifying the different columns and the values they can assume is simply done with a print statement in the start of the file. The second part is slightly more challenging, as it, unlike the first part which required a single SQL command ‘CREATE’, requires 898 INSERT commands for all the Pokémon.

To do this we programmed a python script that could give us a command to effectively transfer the data from the csv file we had obtained through the webscraping. First, we import the function ‘reader’ from the ‘csv’ module, which enables us to read through the created csv file (although it is not strictly necessary to use this module, it does make it slightly easier, turning the csv file into a ‘reader’ object facilitating iteration over it). We do this in a for loop, so there is an iteration for each INSERT command we have to make, corresponding with the number of rows in the csv file and the number of Pokémon.

Because much of the data was still contained in unnecessary HTML tags like <a> or <span>, before we print out the command for the insertion, we first clean the different data values in the row a bit. Most of it was carried out without any errors, except for the data values containing the Pokémon’s abilities. The issue here was that up to three abilities were distributed across three columns in the csv file, with little rhyme or reason. Most Pokémon, although the very issue arises in the fact that it is only “most”, had two abilities in one column. This made it quite difficult to tabulate through the strings and find the ability names which we needed. In the end, we did tinker a solution which captured a very large majority of the abilities, requiring us only to manually fix some 1% of entries (apprx. 10 or so) in the SQL table, because they did not fit the patterns we had identified for the cleaning.

Looking back at it, it would have been far easier to learn regular expressions to sort through strings like this, or have been slightly more diligent with our webscraping. Initially, we had figured we would just copy the entire thing into our website - <a> tags and everything – but this did not work with the website we envisioned, and we did not want to go back and redo the webscraping.

Now, to create the SQL table all that was necessary was to copy-paste the output of this python file into ‘mySQL’ in Wampserver.

One issue still persisted, however. A few Pokémon had several ‘forms’. The image-links to pokemon.db that had the same core-part with the Pokémon’s name added at the end, did not work with these few. Their image-link on Pokémon ended with a suffix like ‘-origin’, so we had to manually fix these links in the SQL table.

While the problems cited above may seem inconsequential, if our database was meant to continuously be updated, it would require manual verification of the new data, to check whether it was correct. It would not be the very seamless transition that is otherwise the ideal for databases of this nature.

**Designing the Individual Websites**

Our overarching design philosophy, although it is perhaps a bit misleading to glorify it by labelling it as such, was to go for sheer minimalism. We had some prior experience with HTML and CSS, but we all felt that there were a lot of messy interactions with both coding languages; that a small change, could unexpectedly break the whole website. By embracing a minimalist philosophy, we could achieve a very “clean” look that did not look too messy, while requiring the minimal amount of coding in HTML and CSS. Partly out of some degree of incompetence, but mostly because of the limited scope of the project and time constraint.

Some of the websites were programmed with PHP. The individual pages for each Pokémon that appear when a user clicks on them in the big database on our home page used PHP. By making use of $\_GET, when the user clicked on a Pokémon in the table they would be redirected towards another php page, storing the information of which Pokémon the user clicked on. This new php page would then be a template that was filled out with data from the entry of the clicked-on Pokémon in the SQL table by making the following request in PHP ($data contains the name of the Pokémon clicked on in the previous page):



By doing this, we could make do with creating a single php page, instead of 898 .html files.

**Connection to Wamp and extracting data from SQL**

The website’s database is created by a PHP script that adds every needed piece of information from each row of the SQL file into a table. First, a connection must be made with Wampserver’s local host which can be made in essentially one line of code, we cache the table stored on the local host with the data we web scraped as a variable. The next step is to make a new variable where we filter out/specify the data we want by passing an SQL request into the local host database. Neither of these two variables are in a format that can be converted to string explicitly since they are variables of type sqli\_object.

To convert the specified data into string, we need to loop through the variable with the function fetch\_assoc() which converts the specified data into an array of arrays with specified keys. Each array which also acts as an element contains all the data in a given row of the table of the specified data. Essentially, we are printing the data for each pokémon we want to display in a html file by using a foreach loop to go through every element of a non-standard array. And to declare which part of the element of this larger array we need to use a specific key which will be the same as the column name. Hence, the following code would output a html that lists all the names of the pokémon from our local host which match the conditions of our sql request:

while($row = $element->fetch\_assoc()){ echo “$row[name]”; }

**Search mechanism and SQL request**

In the site for the Pokémon database, there are 3 ways in which the user can specify which Pokémon they want to see appear on the database: they can search the name or identifier, the type of the pokémon or the range in which their cuteness must lie. The choices of the user are saved using a html form of method GET which reinitiates the page with the exception that the directory will include the users choices. Through php we can then read the choices made by the user and can in turn use them alter the SQL request to show a database that only matches the data the user specified.

In the case of the search bar there is a text input that is submitted when the user hits the enter key. As a consequence of this, the database is reinitiated and runs through the code only with the exception that in the directory, the value of search will be specified and hence $\_GET{search} will have a non–null value. When this happens we attribute this value to the SQL request so that the specified data matches the information provided by the user. We do this by simply creating an if statement that checks if there was a submitted value for the searchbar, if there wasn’t, nothing is added the SQL request and if there was, the search result is added although with other to make a correct SQL request.

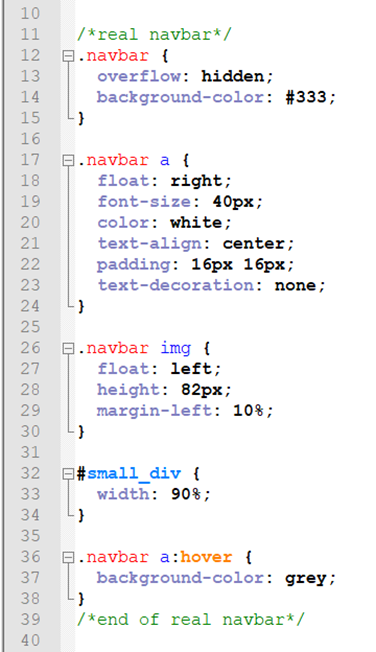
$search = " WHERE name LIKE " . "'$\_GET[search]%'” ;

For additional search queries, when adding extra conditions to the SQL we need to check whether the condition needs to be initiated by a WHERE or AND conjunction depending on whether there was already a condition appended to the SQL request.

For the Pokémon type checkboxes, the user can select a maximum of two types at a time, this is because all Pokémon can only have 1 or 2 types. In addition to this, all of the checkboxes have the same class and name meaning $\_GET{type} will return an array and so to append the conditions for type to the SQL request we have to loop through all of the elements in the array (there might only be one) and append each part to the value that will be added to the SQL request.

**The Navigation Bar**

A comprehensive navigation bar is essential when creating a good website. The website is split into 3 pages (shop, about us, and database), all of which have their separate links on the navbar. The div class *.navbar* first and foremost creates a grey background colour to the top navigation. Inside, the div id *#small\_div* makes it so that the later contents of the navbar will only take up 90% of the total screen width (leaving 10% of space). Furthermore, all the links inside the navbar are formatted using *.navbar a*, (a, as in the hyperlink tag). Note the *text-decoration: none* property, which removes the default line under hyperlinks. This is where the *overflow: hidden* property in *.navbar* comes into play; without it, the navbar completely disappears. This is since the contents of it are being floated (to the right), and thus are taken out of the normal document flow and do not take up space. Because the contents take up no space *.navbar* has no height and disappears. Adding *overflow: hidden* triggers a new block formatting context that prevents *.navbar* from ‘collapsing’ when it has floated contents. An additional property is given, *.navbar* *a: hover* which changes the colour of the links upon hovering over them with the cursor. Finally, an image of a Pokémon ball is sourced onto the left side of the navbar using *.navbar img* with a left margin of 10% to indent it slightly, the same way the links are indented from the left-hand side.





**About Us Page**

A *#main\_div* centers the main contents of the page followed by a first heading and an <hr> with div class *#divider* thematic break. Each member of the team has his own ‘info card’ embedded next to a Pokémon professor of choice. As the name suggests, div ID #info encloses the relevant text, which is separated into several paragraphs within the div class *.title* giving it a distinct grey look. Under each member’s personal biography there is a ‘Contact’ button which opens the computer’s mailing app with the relevant email address already written with the following html code: 

A further distinction had to be made between div IDs *#picture* and *#picture\_right* for picture placements relative to their respective *#info* contents. The *.the\_people* div class was an attempt to group the pictures together with their ‘info cards’ but didn’t work (yet) and currently serves no purpose.

For some reason, the <br> tag did not work as we wanted it to, so a new blank div *.clear* was created to add a page break between each person.

**Shop Page**

The format of the shop is similar to the About Us page. 6 Pokémon Bodybuilder figurines are presented in their *.card* div class. Each has side margins of 2% to centre 3 divs per row. Note the different image width in the html file, which is due to the original picture sizes of the different bodybuilders being different, and as such needed adapting to be all on the same row. The button ‘Add to cart’ takes the user to the Amazon webpage of the respective figurine.

