TECHNICAL DOCUMENTATION

BLOOM (Barcode Linker for Organism and Ontology Mapping): *In Silico* Evaluation of DNA Barcode Accuracy and Efficiency for Taxa Identification in Plant DNA Metabarcoding.

Neudek, M., Serrano Velazquez, M., Spriggs, M., Twigger, K.,

Cranfield University, Cranfield, MK43 0AL, England

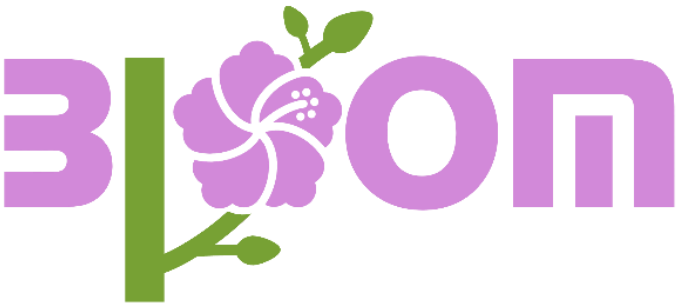

Table of contents

1. Introduction

BLOOM (Barcode Linker for Organism and Ontology Mapping) is a Python application that aims to help biologists select more efficient and accurate universal DNA barcodes for wet-lab experiments. The design principles followed throughout the development are modularity, accessibility, and scalability.

This document aims to gather the technical aspects of the code and explain its structure, core functionalities, and features. Although the information presented is extensive, additional material can be found in the comments written in the scripts and in the application's GitHub repository: https://github.com/ms2206/BLOOM. The libraries used for this project are described in Table 1.

Table 1. Main languages and libraries used in the development of BLOOM

Tool/library	Version	Purpose	Source
Python	3.9.13	Application's code language	https://www.python.org/
Biopython	1.85	To use its *Entrez* module to access NCBI database, the Blast module to access NCBI's BLAST web tool and the Align and Seq packages to align sequences using the user's device resources	https://www.python.org/
csv	1.0	To write and save csv files	https://docs.python.org/3/library/csv.html
ete3	3.1.3	To create and display interactive taxonomy trees	https://docs.python.org/3/library/csv.html
PyQt5	5.15.11	To develop a Graphical User Interface	https://pypi.org/project/PyQt5/
re	2.2.1	To find scientific names in alignment titles	https://docs.python.org/3/library/re.html
yaml	6.02	To read yaml files	https://docs.python.org/3/library/re.html

2. General structure and main file

The structure of the app files consists of different files organised into folders based on their functionality. Although the deployable app is a single .exe file, the structure described in Figure 1 is available in the GitHub repository.

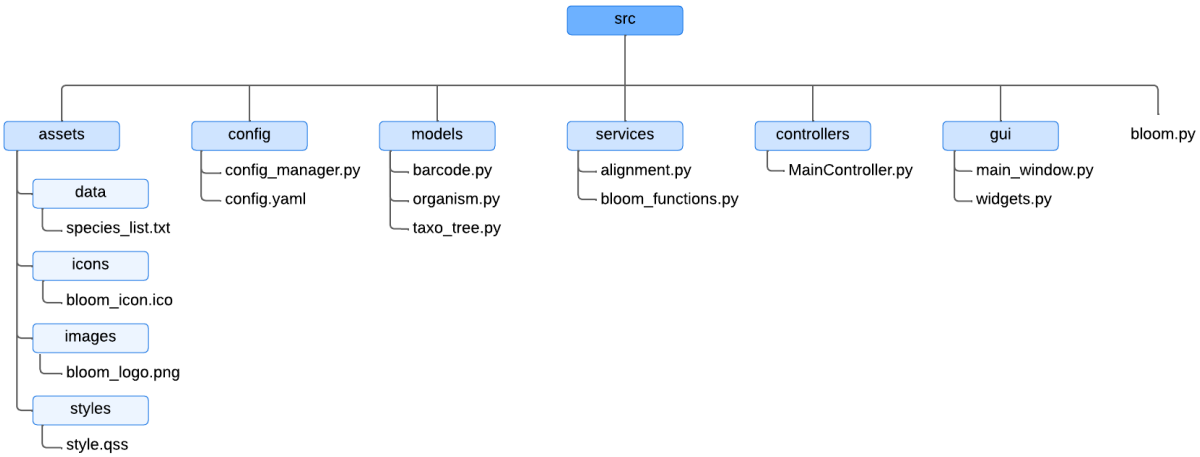

Figure 1. File structure of BLOOM src repository

The “bloom.py” file contains the declaration of the PyQt app and the main function. In this file, the `MainController` and the `MainWindow` instances are created. The user must run this script to run the whole app.

3. Models

The “models” folder contains 4 functional classes divided into 3 scripts. As seen in Figure 2, instances of the `TaxoTree`, `Barcode` and `Organism` classes are created within an instance of the `MainController` class.

3.1 The Barcode class

This class stores information for a single barcode sequence found in NCBI. This information contains:

- The barcode type (trnL-UAA or trnL-P6)
- The trimmed nucleotide sequence
- The forward and reverse primer sequences
- The location of the nucleotides that match the primers of the barcode
- The header or headers of the FASTA records of the sequence uploaded to NCBI.

To create an instance of this class, the barcode type, the raw nucleotide sequence, the FASTA header corresponding to this sequence, and the primers’ sequences are required. When creating an instance, the raw sequence is immediately trimmed to remove nucleotides outside the primers. To do so, the private function `_clean_sequence()` calls the `find_primers()` function from the “alignment.py” service file to get the indexes of the nucleotides that match the primers. It then calls the function `_get_trim_limits()` to determine the minimum and maximum values of those indexes to mark the trimming limits. Finally, the sequence is trimmed, and the primer indexes are updated to match the new sequence.

The header is initially stored as the first element of a list. This allows storing multiple headers in the same `Barcode` instance. When the barcodes are filtered to remove duplicates, headers from the eliminated objects are stored in the only instance kept. Even though the current version of the app does not use this feature, future versions of the app may use this to show the user all the NCBI entries that share the exact same barcode sequence.

To make the class more flexible, it has two built-in types:

- `__int__()` returns the number of headers stored in the instances, which translates to the number of duplicates found for that barcode sequence.
- `__str__()` returns the nucleotide sequence.

3.2 The Organism class

This class stores information about the organism to be studied. When creating an instance of this class with the organism’s scientific name, the taxonomical ID and taxonomy lineage are immediately retrieved from NCBI. This class uses the functions `get_taxa_id()` and `get_taxonomy()` from the “bloom_functions.py” service file.

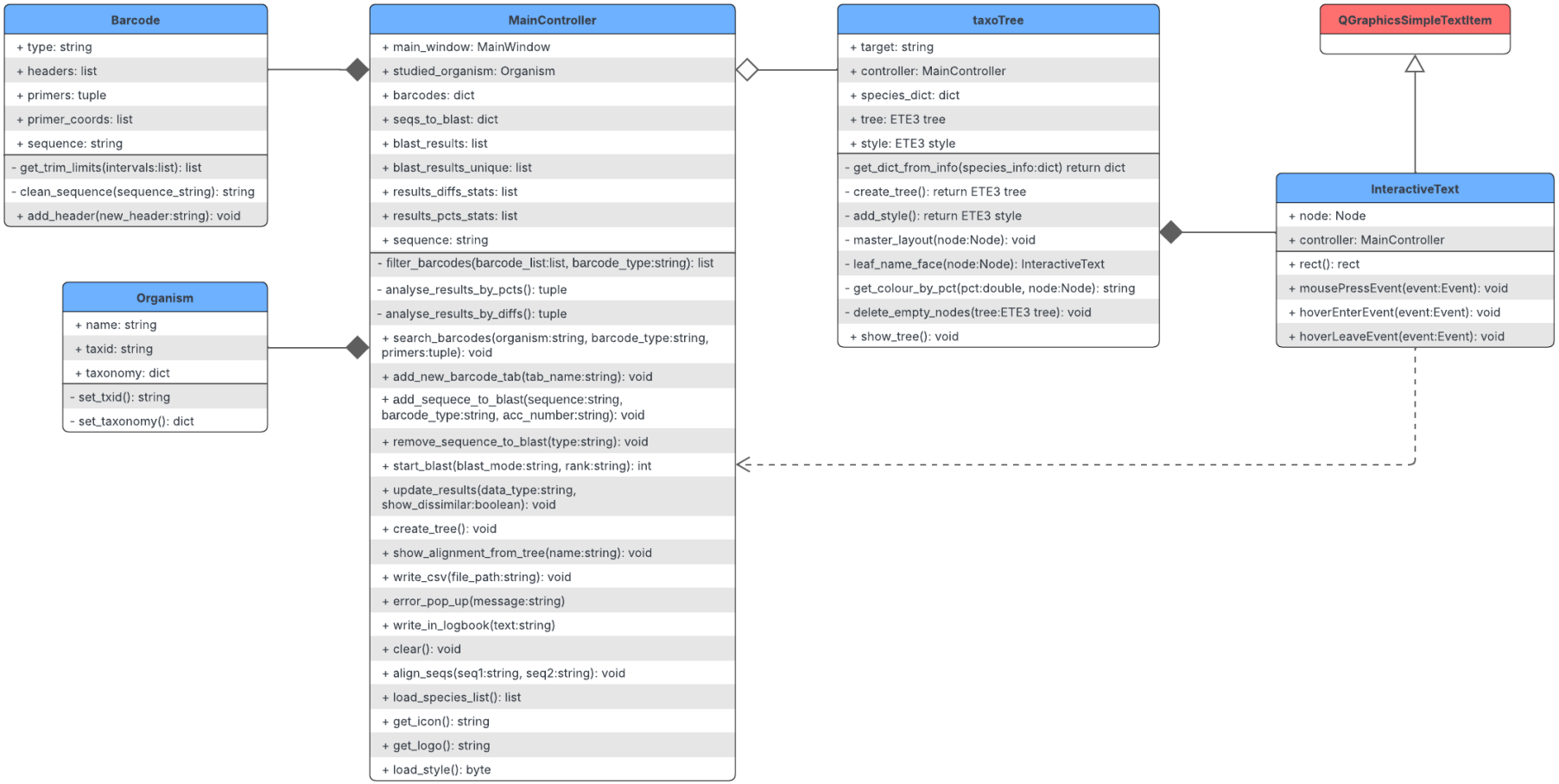

Figure 2. UML diagram for the functional classes in the “models” folder and the MainController class

3.3 The TaxoTree class

This class contains the functions necessary to create and style an ETE3 interactive tree. The tree is created using the results for unique species stored in the instance of the `MainController` class. These results are filtered to get a dictionary matching each species name with its identity percentage with the function `_get_dict_from_info()`. Later, using the NCBITaxa package, the taxonomical ID and taxonomy lineage are obtained for each species, as shown in Figure 3. These are later used to build the branches in the tree. Branches do not have specific lengths because the main purpose of this tree is to organise the organisms found by taxonomical group.

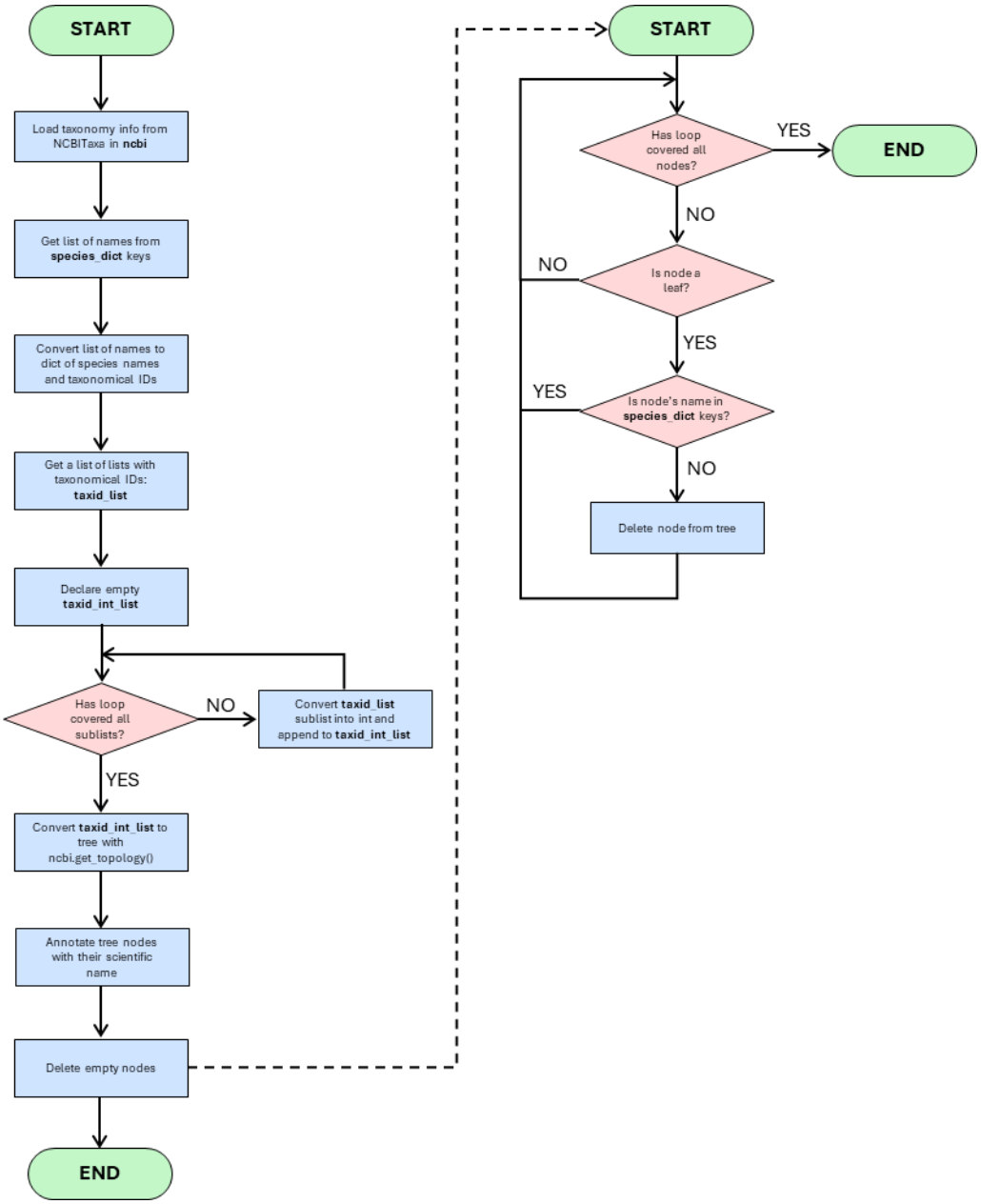

Figure 3. Flowcharts for the construction of the taxonomy tree object

After creating the tree, the style is applied using the function `_add_style()`. The style is a rectangular tree with the blue names for the taxonomical groups and black for the species names in the leaves. Leaves have a background colour which varies depending on the identity percentage, as seen in

Figure 4. The leaf with the studied organism is blue, however, if the input name is a scientific synonym of the original name stored in the NCBI database, the leaf may not appear in blue colour. Leaves also have an interactive element shaped as a label with the identity percentage. These interactive elements are instances of the `InteractiveText` class.

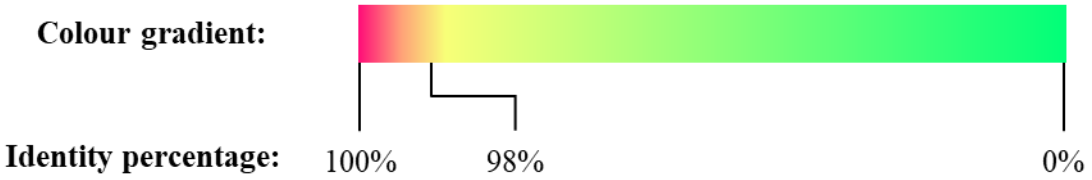

Figure 4. Background colour gradient based on the identity percentage

The only public function `show_tree()` displays the tree in an ETE3 toolkit pop-up window based on the PyQt5 library.

3.4 The InteractiveText class

This class is described in the taxo_tree.py file. It inherits from the PyQt5 class `QGraphicsSimpleTextItem`. Even though it is a custom graphical class, it is grouped with the GUI's widgets because it is only used to provide an interactive element for the `TaxoTree` class.

Each `InteractiveText` instance is associated with a leaf node in the tree and is represented as a label with the identity percentage of that species. When the mouse is hovered over the label, it becomes bold to highlight it thanks to the `hoverEnterEvent()` and `hoverLeaveEvent()`. When the label is clicked, the MainController instance is called through the `mousePressEvent()` to create a pop-up window with all the alignments between the studied barcode sequence and all the sequences found from the selected species in the tree.

4. Services

The services folder contains the scripts with dedicated functions on the backend. These functions are divided into two scripts: alignment.py and bloom_functions.py.

4.1 The alignment.py functions

This file uses the package Align from BioPython to create two instances of the `Align.PairwiseAligner` class: `primer_aligner` and `long_aligner`. The first is used to find primers in a nucleotide sequence, while the second is used to compare sequences of similar length. Due to its different functionalities, each one has different properties, listed in Table 2. These properties are stored in the “config.yaml” file and read with the `ConfigManager` class described in section 7.1.

Table 2. Parameters set for each aligner object

Properties	primer_aligner	long_aligner
Mode	Local	Global
Match score	2	2
Mismatch score	-1	-1
Open gap score	-1	-0,5
Extend gap score	-0.8	-0.1

The `primer_aligner` instance rewards alignments with mismatches more than alignments with gaps since they are more common when finding primers in sequences. On the other hand, the `long_aligner` instance rewards gaps as InDels are more common when comparing two barcodes.

4.1.1. find_primers()

This function, described in Figure 5, returns the location of nucleotides in the barcode sequence where they match a nucleotide in the primer sequence. To do so, `primer_aligner` is used to look for the forward primer and the reverse complement of the reverse primer in the barcode sequence. The alignment objects (from Biopython’s align package) obtained are then evaluated to pick the best one, i.e. the one with the best score. If the alignment is good, the function `get_correct_intervals()` is called to obtain the correct matching coordinates.

4.1.2. get_correct_intervals()

Given a pairwise alignment object where the first (top) sequence is considered the reference, this function returns a list of intervals (tuples of integer indexes) corresponding to regions where the reference sequence is "correct", i.e. the top character matches the bottom character. Gaps in the top sequence are skipped since they do not correspond to positions in the real (ungapped) reference sequence.

4.1.3. get_seqs_alignment()

This function returns an alignment object between two sequences using `long_aligner`. Sequences whose lengths differ greatly provide numerous possible alignments. Even though the align package’s

algorithm is extremely efficient, the function caps the number of alignments reviewed to 100 to prevent it from requiring considerable running time.

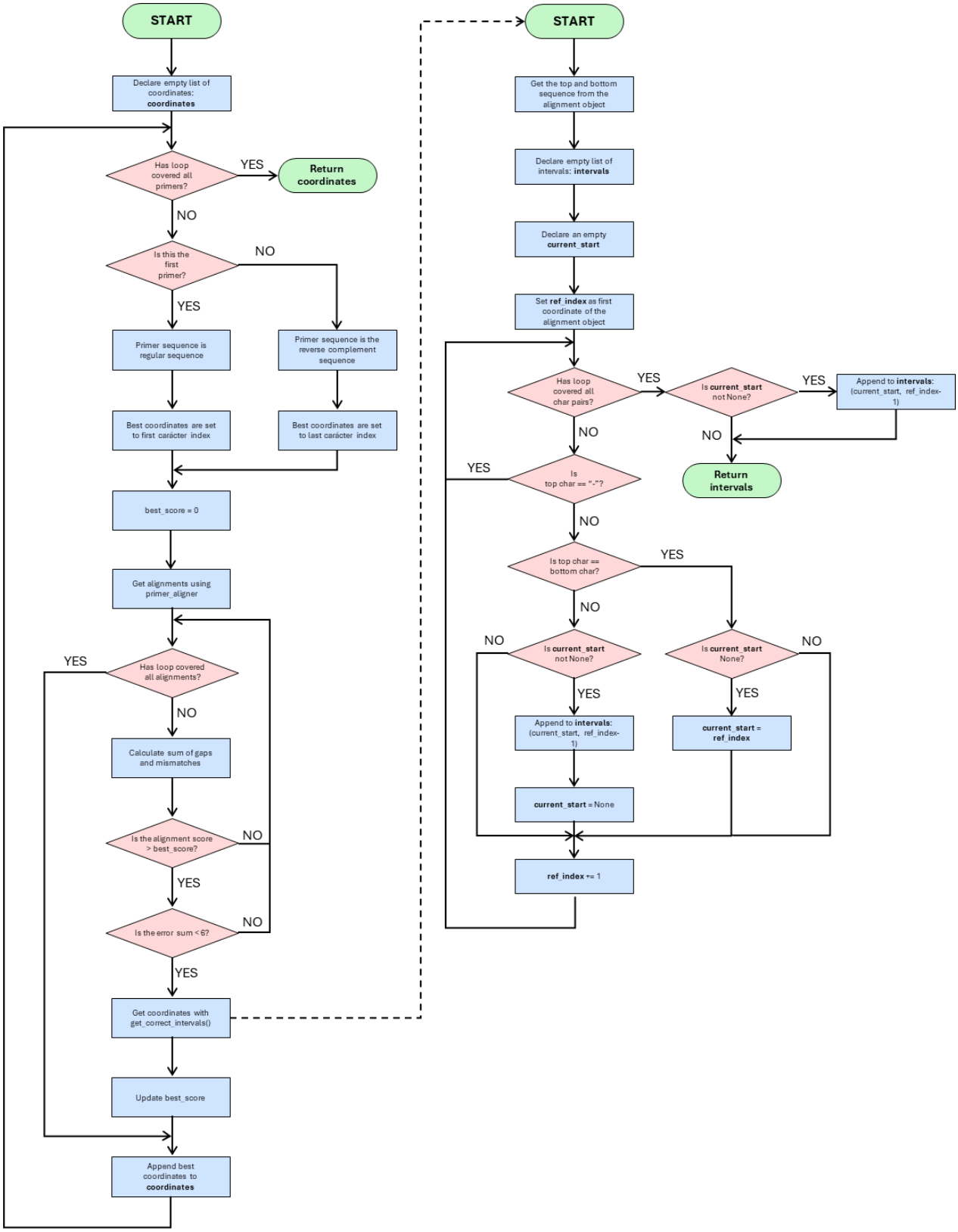

Figure 5. Flowcharts for the find_primers() and get_correct_intervals() functions

4.2 The bloom_functions.py functions

This file contains functions that get and manage the results from the NCBI web-bank and BLAST web-tool. To work with the NCBI's API, the functions use the packages Entrez, SeqIO, NCBIWWW and NCBIXML from the BioPython library. Additionally, the `ConfigManager` class retrieves an email address and API key from the configuration file. The current version of the app uses a mock email and an empty key, nevertheless, future versions of the app may ask the user to input the email and key from the user's NCBI account. These features allow NCBI to process more requests per second and send a message to the user's email address if an error occurs.

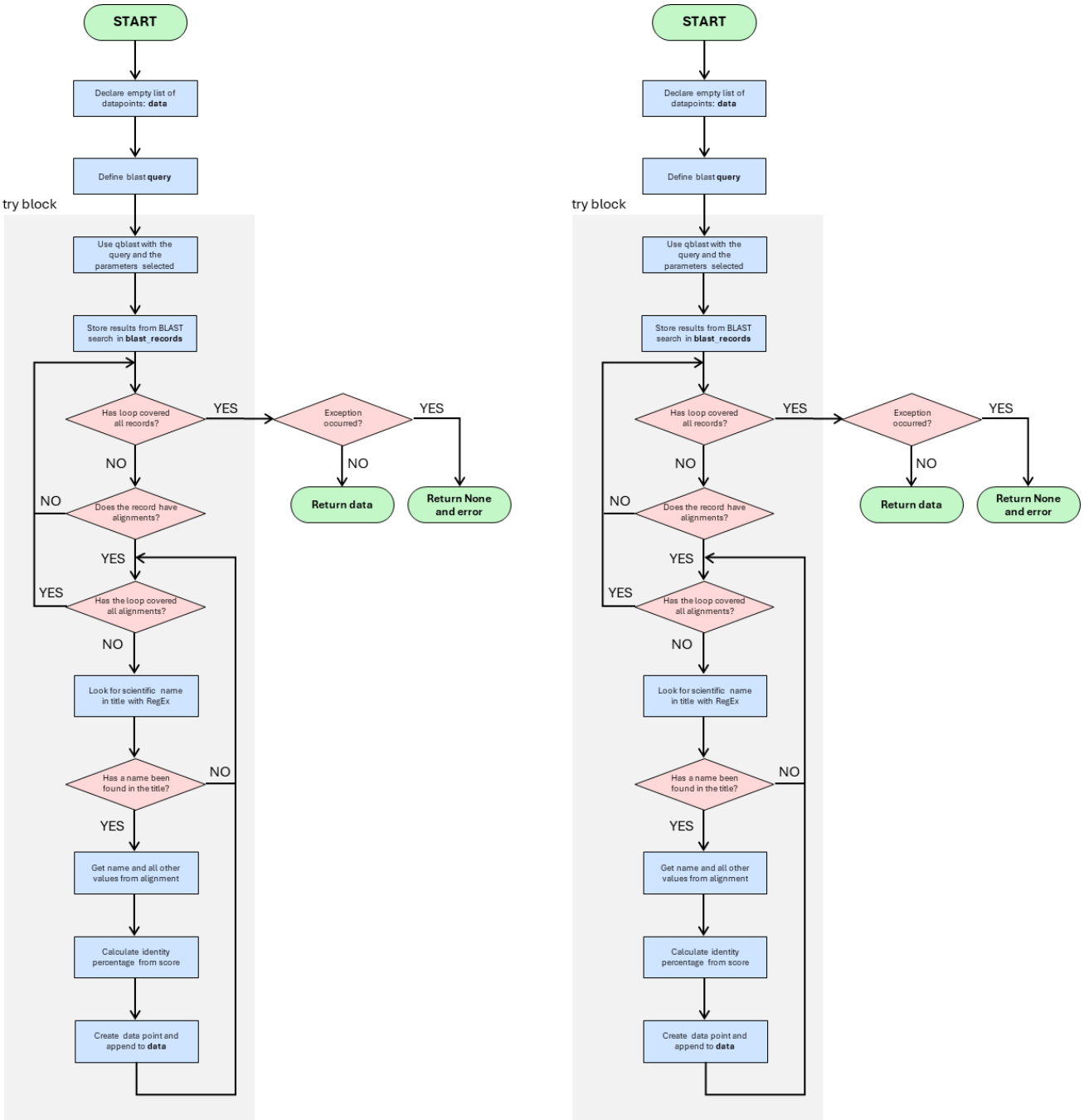

Figure 6. Flowcharts for the get_NCBI_sequences() function (left) and the blast() function (right)

4.2.1. get_NCBI_Sequences()

This function uses a barcode type and an organism taxonomic ID to search for barcode sequences in NCBI nucleotide database. In the case of trnL-UAA and trnL-P6, their sequences can be found in the “gene” feature in the FASTA records uploaded to NCBI. This may not be the case for other barcode types.

As described in Figure 6, once the sequences have been found, the function stores and returns a list of the sequences and FASTA headers. These sequences are then later used to create instances of the Barcode class. If an error is raised while executing the function, the try-except block returns an empty object and an error message.

4.2.2. get_taxonomy()

This function returns the taxonomical lineage of a given organism’s name from the NCBI taxonomy database. The returned lineage contains the taxonomical ID, general name and scientific name of all ranks preceding the species.

4.2.3. get_taxa_id()

This function returns the taxonomic ID of a given organism’s name from the NCBI taxonomy database. IDs are used to search for sequences instead of scientific names, as they are less prone to error.

4.2.4. blast()

This function uses the BLAST web tool to search for alignments of a given sequence and barcode type within a taxonomic rank range. The program used is “blastn”, however, the user may choose to use “Megablast” or “Discontiguous Blast”. The first option is faster, but it is optimised for similar sequences, while the latter can return more alignments as it is optimised for less similar sequences.

The hitlist size, the maximum number of hits retrieved from the tool, is set to 20000; nevertheless, blasting for the family rank (i.e. the greatest rank available in the tool) may return up to a few thousand hits. The score parameters are set to +1 for a match, -1 for a mismatch, -1 for a gap and -2 for extending a gap. Using these parameters allows the maximum possible score to be the same as the number of bases in the target sequence. The score is used to calculate the identity percentage with the following equation:

$$Identity\ \% = \frac{score}{\text{number of bases in target sequence}} \times 100\%$$

This identity percentage is not the same as that reported by BLAST. BLAST may report alignments with a perfect identity percentage but with a low score. This happens because BLAST is a local aligner, therefore it does not compare full sequences; it breaks them into small chunks and finds the best alignments for each. These chunks are called High Scoring Pairs (HSPs), and although they are shorter than the actual barcode sequence, if they align perfectly with the target sequence, BLAST reports a 100% identity, but they do not match the complete sequence. These results are misleading, therefore, HSPs must be filtered to provide accurate results.

When BLAST finds multiple HSPs for the same hit, it has found InDels in the alignment because these chunks are separated by hundreds of bases in the original genome. If the InDel is only a few bases long, BLAST will report it as a gap and not break the alignment into HSPs. Therefore, it can be inferred that multiple HSPs mean one or more InDels in the aligned gene.

The chosen identity threshold for considering a barcode sequence dissimilar to the target is 98%. This means that a sequence with more than 2% of the target's total number of bases differing from the target's bases. As an example, if the target's sequence has 1000 bases, the aligned sequence needs to have only 20 different bases to be dissimilar. Since BLAST reports HSPs when the InDel is hundreds of bases long, a hit with multiple HSPs can never be considered identical to the target sequence. Therefore, the `blast()` function only studies the first HSP (if there are various) to calculate the score and identity percentage.

The time this function takes to finish depends primarily on the BLAST servers. Performing multiple searches in a short time can lead to delays. Finally, results are stored in a dictionary and appended to a results list. This list is returned at the end of the function, as shown in Figure 6.

4.2.5. filter_data()

This function filters the data previously obtained with the `blast()` function to report only one entry per scientific name. If BLAST has found multiple hits for the same species, this function takes only the one with the highest identity percentage, as it is the worst-case scenario.

5. The GUI

The Graphical User Interface is divided into two files: “main_window.py” and “widgets.py”. The first contains the custom class `MainWindow`, and the second contains the description of all custom widgets whose instances are created in the `MainWindow` class. These relationships between classes are depicted in Figure 7. All components are based on the Python graphical library PyQt5.

5.1. The MainWindow class

This class inherits from `QMainWindow` and is the window that appears when the app is run. It contains the instances of some widgets described in the “widgets.py” file: an `InputModule`, an `OutputModule`, a `LogBook`, a `Header` and a `CustomMenuBar`.

Most methods operate on the app’s different widgets and are called by the `MainController` class. The `closeEvent()` function kills the app by force when it is closed. When the taxonomy tree is shown, it starts another main PyQt5 process. This process does not terminate when the app is closed, and it might continue running unnoticed by the user. This function ensures that all processes carried out by the app are terminated when the app is closed.

5.2. The CustomMenuBar class

It is a menu bar situated at the top of the main window. It has two submenus:

- **File**: allows to close the app, create a csv file with the results obtained from the barcode analysis and the option to clear all information in the app so that a new organism can be studied. This option resets as well the attributes in the MainController instance created.
- **Help**: displays a pop-up message with the link to the GitHub repository.

5.3. The LogBook class

This module contains a text box where all user’s actions and the time they started are logged. This helps the user keep track of what has been done and check the time an action takes to finish.

5.4. The Header class

This class contains the app’s title and a tool button that commands the main window to hide or show the input module when pressed. This makes the output module bigger, and the results can be seen clearly. The purpose of this header is mainly aesthetic.

5.5. The AlignmentPopup class

This class creates a pop-up window to display the alignment between two sequences. This class is used when comparing two barcodes or selecting a species in the taxonomic tree. An instance of this class takes as an argument an `Alignment` object from the BioPython module (when comparing barcodes) or a string concatenating multiple Alignment objects (when selecting a species in the tree).

5.6. The InputModule class

This class comprehends all the widgets where the user inputs information. This module is a tab widget divided into two bar tabs: the SearchTab and the BlastTab.

5.6.1. The SearchTab class

The `SearchTab` class incorporates all widgets to choose the parameters for the barcode search. These include the organism to study, the type of barcode and the sequence for the forward and reverse primers. The tab has an autocompleter for the organism's name so the user can get help writing it. The list of organisms is stored in the assets folder and is static. The barcode selection has a drop-down to select from a list of barcode types. Finally, the primer sequences are autocompleted depending on the barcode selected, although they can still be edited by the user.

The function `search_button_action()`, which is triggered when the search button is pressed, handles missing inputs or errors when calling functions in the MainController class and displays them as pop-up messages to prevent the app from crashing.

5.6.2. The BlastTab class

The `BlastTab` class incorporates the entry widgets to choose the parameters to BLAST a barcode sequence and to analyse results. It is divided into three sections:

- **BLAST section**. It contains widgets to select the parameters to analyse barcodes with the BLAST tool and a button to start it. The options for the BLAST mode are explained in section 4.2.3. The taxonomic ranks range from genus to family, family being the broadest rank within the possible options.
- **Data display section**. It contains widgets to select how to display the results obtained for better interpretation. The data can be shown either by the number of differences or by the identity percentage. The radio button for the "Show dissimilar hits" shows or hides all hits and species that are considered dissimilar so that the others can be reviewed more carefully.
- **Tree section**. It contains a button to create and show the taxonomy tree from the results obtained.

5.7. The OutputModule class

This class takes the largest screen space. It displays and reviews the barcodes fetched from NCBI and shows a generic analysis of the results obtained with the BLAST web tool. This module contains two types of tabs: the `BarcodesTab` and the `ResultsTab`.

5.7.1. The BarcodesTab class

This tab contains the barcodes found in NCBI for a specific organism and barcode type. When an instance of this class is created, it also creates instances of the `BarcodeCard` class to display the sequences, and their information found with the `populate_barcode_tab()` function. Whenever a barcode search is performed a new `BarcodesTab` instance is added to the output module. However, if the barcode type searched already has a tab, it will overwrite it.

Whenever a `BarcodeCard` is selected, the `BarcodesTab` class calls the controller instance with the `check_barcode_card()` function to store the selected card's sequence and accession number. They can be later used for BLAST and results visualisation. If the card is deselected, the function `uncheck_barcode_cards()` commands the controller to delete its stored information.

5.7.2. The BarcodesTab class

The `BarcodeCard` is a selectable widget via radio button that displays the information for each Barcode class instance obtained when searching for sequences in NCBI. The widget displays the following data:

- The number of duplicates i.e. the number of records uploaded to the NCBI database whose sequence is identical.
- The FASTA header of the NCBI record. Even though a Barcode instance may contain multiple headers, only one is displayed.
- The nucleotide sequence is displayed in a `Sequence` class. This custom class contained in the `BarcodeCard` instance allows highlighting the primers whenever the mouse hovers over it. Highlighting is possible thanks to the coordinates of the matching nucleotides, explained in section 3.1.

Whenever a card is selected with the radio button and the user right-clicks on another one, an `AlignmentPopup` object will be created and a window with the alignment will be shown, as explained in previous sections.

5.7.3. The ResultsTab class

The `ResultsTab` contains the initialisation of the widgets that will display the results obtained through BLAST. There is only one instance of this class, initially empty. The tab includes a title and two instances of the `CombinedChartWindow` class.

Whenever stats are created or modified for their display, the `add_stats()` function updates the widgets with the new information. The title is only updated when a new BLAST search is conducted. The first `CombinedChartWindow` instance displays the statistics of all hits in BLAST, and the second one displays results for unique species, that is, only one BLAST entry per different species.

5.7.4. The CombinedChartWindow class

This class contains a title indicating if the results are for all hits or just unique species, a donut plot using the `PieChartWidget` class and a bar plot using the `BarChartWidget` class.

The animation parameters are defined within this class. The animation updates every 16 ms, which is equivalent to approximately 60 fps. The animation takes a total of 1.5 seconds and restarts whenever the tab is changed back to the results tab.

The colours used for the graphs are obtained following a linear gradient between red and green, passing through yellow. This is achieved with the function `get_color_palette()`, which returns a list of `QColor` objects depending on the size of the list of values to display.

5.7.5. The PieChartWidget class

This class displays a donut chart with sectors representing the number of hits/species that have a specific number of differences or fall in a range of identity percentages. In the middle, there is a text label with the total number of hits/species recorded. Each sector's size is dependent on the value linked to it. Each sector has a colour that ranges from red (for similar hits) to green (dissimilar hits). The chart is animated with a pan for aesthetic purposes.

5.7.6. The BarChartWidget class

This class displays a histogram with bars representing how many hits/species have a specific number of differences or fall in a range of identity percentages. The X axis has a label to interpret what the bars represent. Each bar's height is dependent on the value linked to it. Each bar has a colour that ranges from red (for similar hits) to green (dissimilar hits). The chart is animated by gradually increasing the bars' heights for aesthetic purposes.

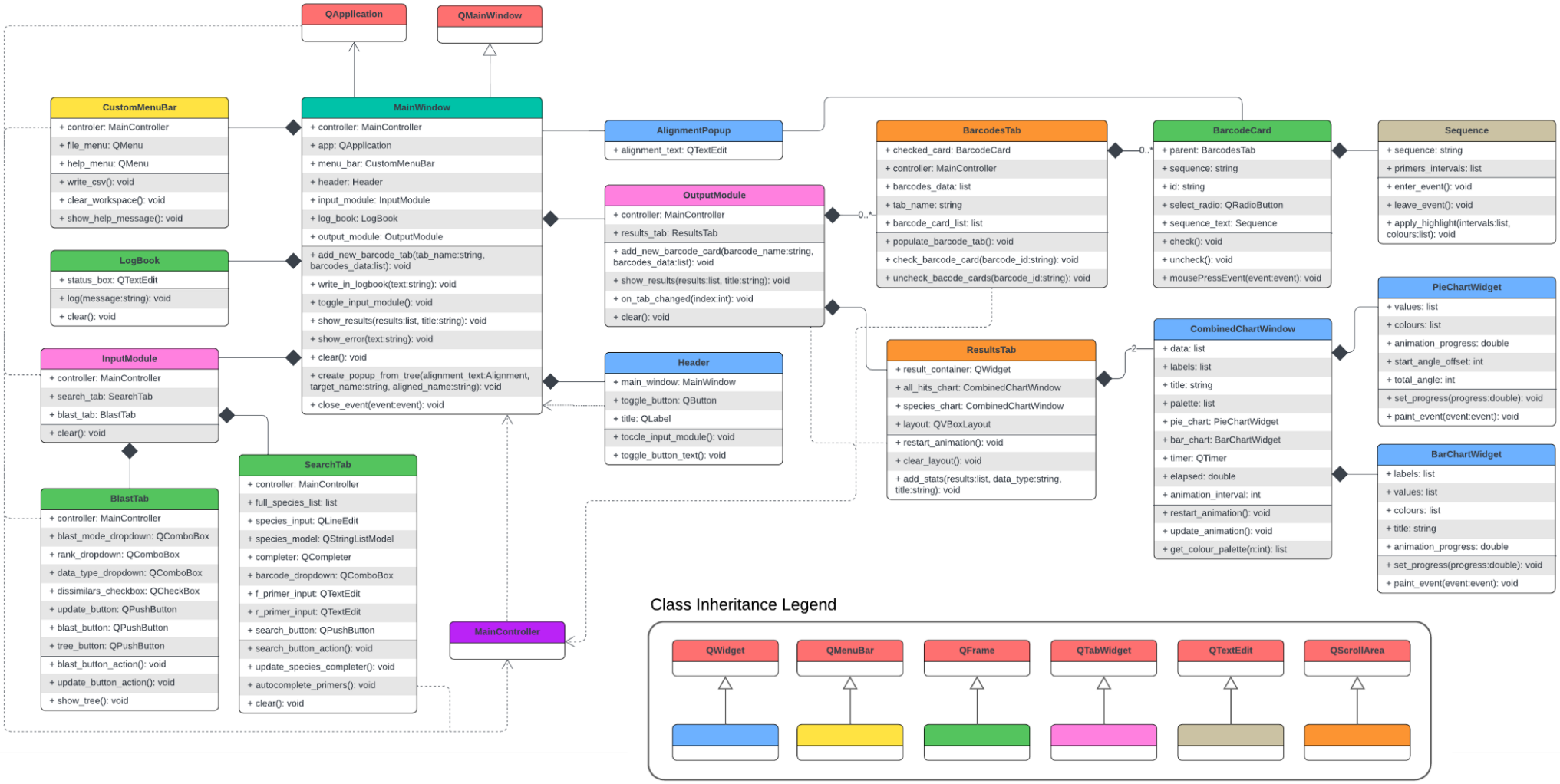

Figure 7. UML diagram for the Graphical User Interface

6. Controllers

The “controllers” folder contains the “MainController.py” file, where the `MainController` class is declared. This class serves as a middle layer between the GUI and the backend. Only one instance of this class is shared through files and is initially declared in the “bloom”.py script. This ensures that widgets can access data stored in its attributes. Figure 7 depicts how the GUI’s widgets use the class to call its functions, and Figure 6 shows how the controller uses the backend models and services to operate all commands coming from the GUI. The controller then calls function from the main window to operate over the widgets.

The class contains multiple methods. Nevertheless, they can be grouped based on their functionality.

6.1 Functions to search for barcodes

These functions are executed when the “Search” button is pressed. The function `search_barcodes()` uses the services from “bloom_functions.py” to get the sequences from NCBI, then calls the private function `_filter_barcodes()` to remove duplicate sequences, as shown in Figure 8. Finally, the function `add_new_barcode_tab()` commands the main window to create a new `BarcodesTab`. Barcodes are stored in a dictionary with the barcode types as keys.

6.2 Functions to store sequences and BLAST them

Functions `add_sequence_to_blast()` and `remove_sequence_to_blast()` add and remove, respectively the selected or unselected sequence from a `BarcodeCard` class.

The function `start_blast()` gets the necessary parameters and calls the `blast()` function from “bloom_functions.py” to get the alignment results for the selected sequences. As depicted in Figure 9, this function only works if only one barcode card is selected. In future versions, the app may evaluate a combination of barcodes, and it can benefit from the current feature of being able to store multiple selected barcodes to BLAST.

Once the results are obtained for all hits and unique species, functions `_analyse_results_by_diffs()` and `_analyse_results_by_pcts()` create lists of frequencies. These frequencies gather the number of hits/species that contain a specific number of differences or fall within a specific range of identity percentage. The threshold for the identity percentage to consider a hit dissimilar is 98%, while the threshold for the number of differences is calculated based on this percentage with the following formula:

$$threshold = \left\lceil number\ of\ bases\ in\ target\ sequence \times \frac{100 - identity\ threshold\ \%}{100} \right\rceil$$

Given that the identity percentage is calculated with the score, results may show a different number of hits/species between the number of differences graphs and the identity percentage graphs when dissimilar hits are not displayed.

The function `update_results()` is used to get the correct list of frequencies based on the visualisation parameters the user has inputted.

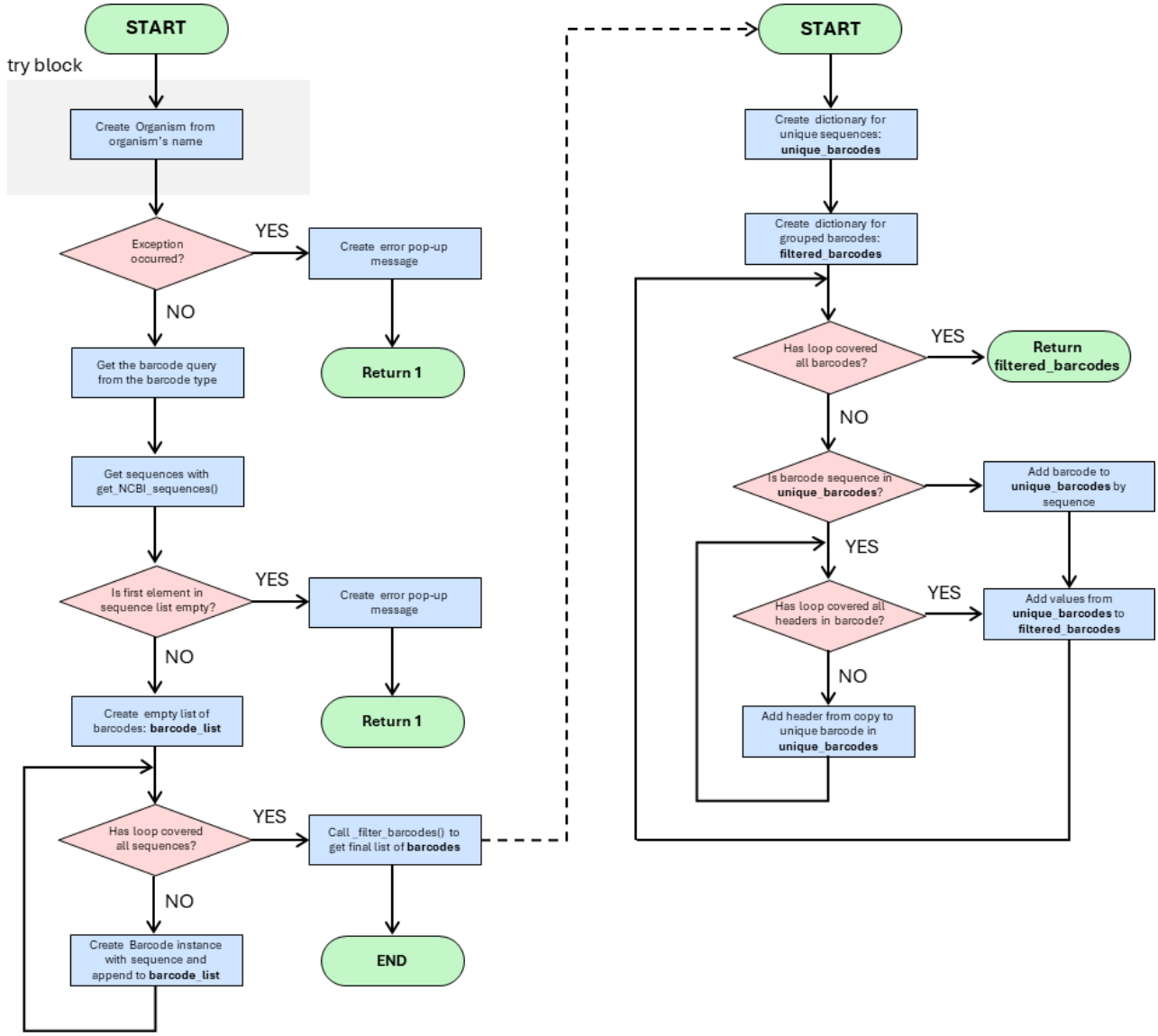

Figure 8. Flocharts for the search_barcodes() and _filter_barcodes() functions

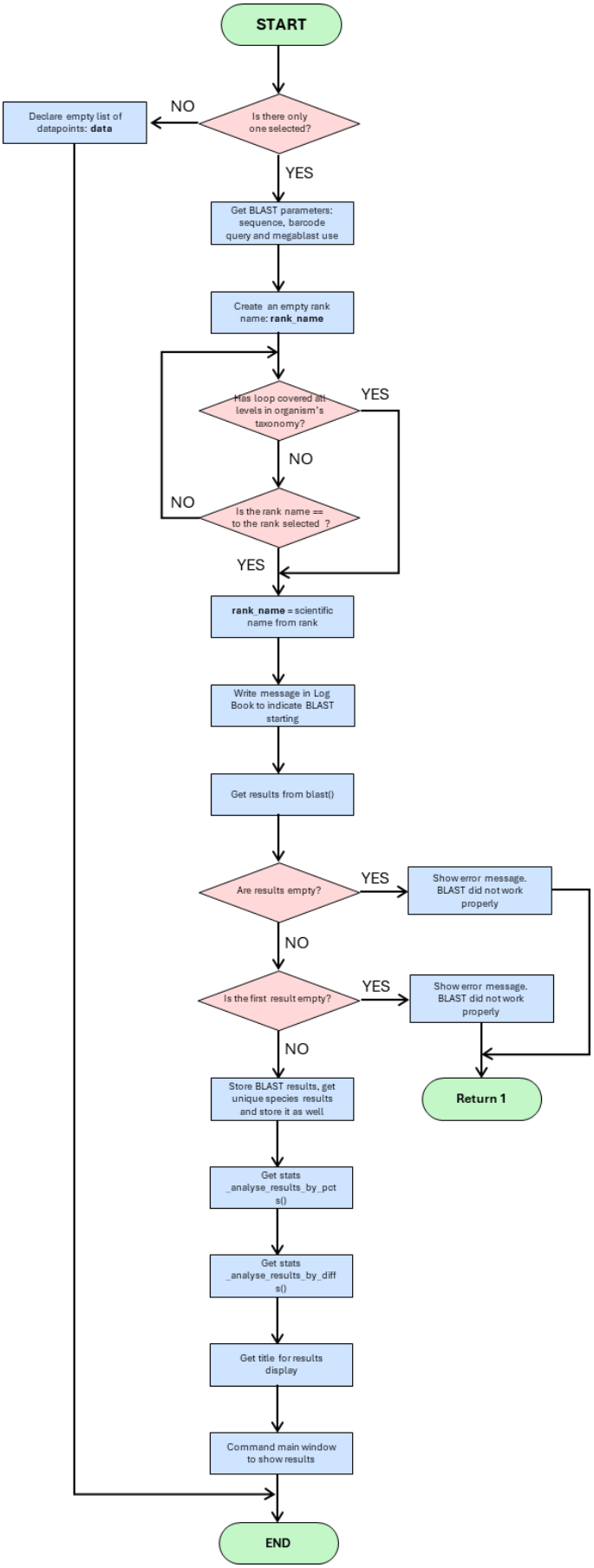

Figure 9. Flowchart for the start_blast() function

6.4 Functions for messaging

`error_pop_up()` is a function that commands the main window to create a pop-up error window for a given message. This function is called whenever a function encounters an error.

The function `write_in_logbook()` commands the main window to write in the `LogBook` object given message.

6.5 Other functions

The `write_csv()` function creates a csv file, writes the results for all hits obtained from BLAST and stores it in a path the user selects.

The `clear()` function resets every attribute and commands the main window to delete all information in the widgets.

The `align_seqs()` function calls the `get_seqs_alignment()` from the alignment.py service file to return an alignment object given two sequences.

The `load_species_list()` function reads and returns a list with all the species names stored in the “species_list.txt” file.

The `get_icon()`, `get_logo()` and `load_style()` functions return the path to the app icon, logo and style sheet, respectively. The current version does not use the `get_logo()` function.

7. Configuration and assets

The configuration and assets folder store constant values and files so they do not need to be hardcoded in the scripts. Configuration parameters are written in the “config.yaml” file and read with the `ConfigManager` class. Asset files are read or loaded with the `MainController` class.

7.1 The ConfigManager class

The `ConfigManager` class does not have any attributes as shown in Figure 10. It is just a utility class used to access configuration values. Instances of this class are created in other scripts but never inside other classes.

The function `get()` returns a parameter in a string given a key in dot notation. For example, to get the primer aligner’s match score, the argument will be: “primer_aligner.match_score”. There are dedicated functions that return a list of data from the barcode types stored: `get_barcode_names()`, `get_barcodes_queries()` and `get_barcodes_primers()`.

Configuration parameters are written in a yaml file, which is a common file format for storing this type of data.

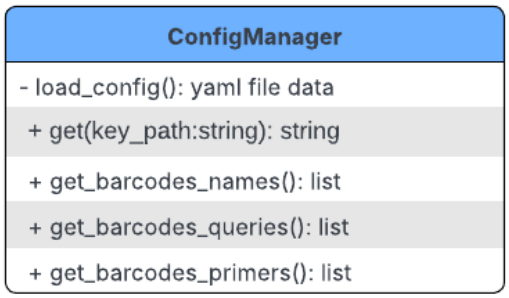

Figure 10. UML diagram for the ConfigManager class

7.2 The assets

Assets include complementary files to the app:

- **Icon and logo**. The icon is used for the app design. The logo is not used, nevertheless, it is provided.
- **Species list**. It is a text file with all the scientific names of the species recorded in the NCBI database. It is static; therefore, if new species are added to NCBI this list will need an update.
- **App style**. It is a style sheet containing all aesthetic parameters for the widgets. This file can be easily modified without changing the GUI’s code.

Figure 11. BLOOM icon (left) and logo (right)