COMP1811 – Python Project Report

Your Name:	**Brendon Shymanskyi**	Student ID	001419181
Partner's name:	**Mykhaylo Zhyhan**	Student ID	**001398148**

1. Brief statement of features you have completed

THIS SECTION SHOULD BE THE SAME FOR ALL GROUP MEMBERS

1.1 Circle the parts of the coursework you have **fully completed and are fully working**. Please be accurate.	**Features** **F1a:** i ☒ ii ☒ F1b: i ☒ ii ☒ **F2a:** i ☒ ii ☒ F2b: i ☒ ii ☒ **F3a:** i ☒ ii ☒ iii ☒ F3b: i ☒ ii ☒
1.2 Circle the parts of the coursework you have **partly completed or are partly working.**	**Features** **F1a:** i ☐ ii ☐ F1b: i ☐ ii ☐ **F2a:** i ☐ ii ☐ F2b: i ☐ ii ☐ **F3a:** i ☐ ii ☐ iii ☐ F3b: i ☐ ii ☐
Briefly explain your answer if you circled any parts in 1.2	

2. Concise List of Bugs and Weaknesses

*A concise list of bugs and/or weaknesses in your work (if you don't think there are any, then say so). Bugs that are declared in this list will lose you fewer marks than ones that you don't declare! (**100-200 words**, but word count depends heavily on the number of bugs and weaknesses identified.)*

THIS SECTION SHOULD BE COMPLETED INDIVIDUALLY FOR F1 AND F2 AND AS A GROUP FOR F3.

2.1 Bugs

List each bug plus a brief description. A bug is code that causes an error or produces unexpected results.

We fixed all the bugs we could find.

2.2 Weaknesses

List each weakness plus a brief description. A weakness is code that only works under limited scenarios and at some point produces erroneous or unexpected results or code/output that can be improved.

The code works as planned. We did not notice any weaknesses. However, we improved it several times and changed some parts, because we forgot to add an if or else statement for conditions in several places in conditional statements, for example, if there are no grandchildren, it displayed ‘no grandchildren found’, but we got an error or when we selected the wrong option, we got Value Error, but everything was fixed and worked as planned. We tried not to make any mistakes and keep the code strong enough to avoid any difficulties. And we are sure that the error will not be so easy to find.

3. Description of the features implemented

*Describe your implementation design and the choices made (e.g. choice of data structures, custom data types, code logic, choice of functions, etc) and indicate how the features developed were integrated. (**200-400 words***

THIS SECTION SHOULD BE COMPLETED INDIVIDUALLY FOR F1 AND F2 AND AS A GROUP FOR F3.

3.1 My Feature

Design description for implemented feature…

F2 – Feature 2:

Our code is divided into six different files: ‘main.py’, ‘list.py’, ‘person.py’, ‘relationship.py’, ‘Family_Tree.py’, and ‘family_tree.csv’.

Each of us created classes, inside those each of us had implemented some functions that were required in order to complete our individual features.

We decided to use classes for all the functions of the following tasks, so for task 2 I used the classes we created, ‘Person’, ‘Relationship’ and ‘FamilyTree’.

After my teammate and I decided to merge the branches of our family tree, we created a function ‘get_branch’ to find a branch from a person so that we could use each other's combined branches in future tasks.

We have created extra functions for interactivity of our program, for example ‘average_age_per_person’ just to know that average age of everyone not only of dead people.

We have the main file ‘main.py’, which calls all the functions we need from other files, but with a developed design.

F2a – Siblings and Cousins:

Using the created functions for finding parents and children, I found siblings and added them to the list.

To find cousins, I searched for my parents, their brothers using the ‘find_siblings’ function, and then the children of parents' brothers, and added them to the list.

Each function could use previous functions to prevent duplication of code and reduce its length in general.

F2b – Birthdays and Birthdays in Common:

At first, I created a function that return the birthday of one person from the specified values in the list, then I used it in functions that contain sorted and unsorted birthdays. Now I can use this function to find birthdays and sorted birthdays calendar of only one branch using **‘get_branch’** function.

F3a – Average Age:

We used the **‘get_branch’** function to find the ages of people in only one branch. At first, we needed to find the age of one person, so we created the function ‘get_age’ using the previous for finding the birthday, and then used it to find the age of all people in the list. We also created a function to find the average age of only dead people.

F3b – Number of Children

The ‘average_children_per_person’, ‘total_children_count’ was created to fulfil this feature. As all other features we have created options in our menus that would output the features, for example the number of children for each individual will be displayed when we select an individual and choose to see their children there would be a total number of children that the selected person has. Regarding others sub features you can see average age at death and average children per person this would be displayed in menu called “Find the information of all recorded people”.

4. Classes and OOP Features

*List the classes you developed and provide an exposition on the choice of classes, class design, and OOP features implemented. List all the classes used in your program and include the attributes and behaviours for each. You may use a class diagram to illustrate these classes – do not include the class code here. Your narrative for section 4.2 should describe the design decisions you made, and the OOP techniques used (abstraction, encapsulation, inheritance/polymorphism). **Note**: stating definitions here will not get you marks, you must clearly outline how you implemented the techniques in your code and WHY. (**400-600 words**)*

THIS SECTION SHOULD BE COMPLETED INDIVIDUALLY FOR F1 AND F2 AND AS A GROUP FOR F3.

4.1 List of Classes Used with a Brief Explanation

a. F2:

Class Person:

The ‘Person’ class is created to initialise a person for our family tree, it contains such attributes as ‘name’, ‘birthday’, ‘gender’ and ‘death’ day. This class is closely related to the ‘relationship’ class, which creates relationships between people.

Class Relationship:

The ‘Relationship’ class was created in order to create relationships between several people. And ‘Spouse’, ‘Parent’ and ‘Child’ relationships were included in the class and were subclasses or child classes.

CLASS FAMILYTREE:

In this class, only an array for members was created. It contains only functions from tasks that relate to people in general, not to a specific person, such as 'average_age_at_death' or 'average_children_per_person'. In addition, there were functions for adding relationships between people, adding a member to a list, and a function for searching for a person by name.

b. F3:

CLASS FAMILYTREE:

Function to calculate total children and average number of children per person were added to class 'FamilyTree'. First function calculates total number of children of every person and second one calculates an average number of children that every person has, we took all the children and devided per total number of all people in family members.

Also we had added functions for calculating the age at what the deceased people have died and then we have calculated the average of these ages for all dead people and another function to separate a list in two branches: Otto's and Cornelia's.

4.2 BRIEF EXPLANATION OF CLASS DESIGN AND OOP FEATURES USED

a. CLASS PERSON:

Class 'Person' was used with attributes: 'name' which is attributed to name of the person, 'birthday' which attributes to persons birthday in format of integer, 'gender' which is attributed to persons gender whether is male or female, 'death_day '- attributes to date when a person was dead or if alive we indicate as -1, and relationship which attributes to a list of relationships that are predefined in our list.

The 'Person' class includes functions for retrieving relationships, such as parents, children, and siblings, making it responsible for managing individual person-specific data.. All attributes in this class are encapsulated. Also, the class has a list of relationships to which people's relationships with each other will be added.

Contains methods that belong to a specific person and all sub-features from my feature, such as returning and displaying siblings of individuals and returning their cousins. Also includes the method for calculating an age.

b. CLASS RELATIONSHIP:

Contains the initialization of two people, designed to unite them through a relationship, such as a parental, child, or spousal relationship. It is a parent class for three child classes, which are 'Spouse', 'Parent', and 'Child' relationships. It was decided that using this class would be ideal for relationships between people, rather than adding attributes to the 'Person' class. For this class was used the 'abstract method'.

c. Class FamilyTree:

Contains functions for adding relationships, in this class we stored the list of members and functions that add people with relationships to the list. Functions to calculate total children and average number of children per person were added to this class. First function calculates total number of children of every person and second one calculates an average number of children that every person has, we took all the children and divided per total number of all people in family members. We decided to split it up to make the code more understandable and not to scatter it in files that are not logical for it.

5. Code for the Classes Created

*Add the **code for each of the classes you have implemented yourself** here. If you have contributed to parts of classes, please highlight those parts in a different colour and label them with your name. Copy and paste relevant code - actual code please, no screenshots! Make it easy for the tutor to read. Add an explanation if necessary – though your in-code comments should be clear enough. You will lose marks if screenshots are provided instead of code. **DO NOT provide a listing of the entire code. You will be marked down if a full code listing is provided, or you include the code as a screenshot**.*

THIS SECTION SHOULD BE COMPLETED INDIVIDUALLY FOR F1 AND F2 AND AS A GROUP FOR F3.

5.1 Class Person:

```
class Person:
    def __init__(self, name: str, birthday: int, gender: str, death_day: int):  #
initialize the person's attributes
        self._name = name
        self._birthday = birthday
        self._gender = gender
        self._death_day = death_day

        self._relationship = []

    def __str__(self):  # returns a string representation of a person
        return self._name

    def __repr__(self):  # helps to represent the relationship and understand the object
        return self._name
```

```
def find_siblings(self):
    """Return siblings of the selected person.
    finds siblings through parents' children"""
    siblings = []
    parents = self.find_parents()   # variable for parents, using 'find_parents' function
    for parent in parents:
        for relationship in parent.get_relationships():   # loop for relationships in
parents, using 'get_relationships' function
            if not isinstance(relationship, ParentRelationship):    # if the relationship
is not a parent relationship
                continue
            if relationship.person2 != self and relationship.person2 not in siblings:
                siblings.append(relationship.person2)
    return siblings
```

```
def find_cousins(self):
    """Return cousins of the selected person.
    Loop, that goes through parents, then siblings of parents and find their children"""
    cousins = []
    parents = self.find_parents()
    for parent in parents:
        siblings = parent.find_siblings()
        for sibling in siblings:
            children = sibling.find_children()
            for child in children:
                cousins.append(child)
    return cousins
```

```
# static method: a method that belongs to a class, but not associated with a class object
@staticmethod
def get_date(date):
    """Converts a date to integers"""
    day = int(str(date)[-8:-6])
```

```python
    month = int(str(date)[-6:-4])
    year = int(str(date)[-4:])
    return day, month, year
```

```python
def get_birthday(self):
    """"Return birthday of the selected person."""
    birth_day, birth_month, birth_year = self.get_date(self._birthday)
    return birth_day, birth_month, birth_year
```

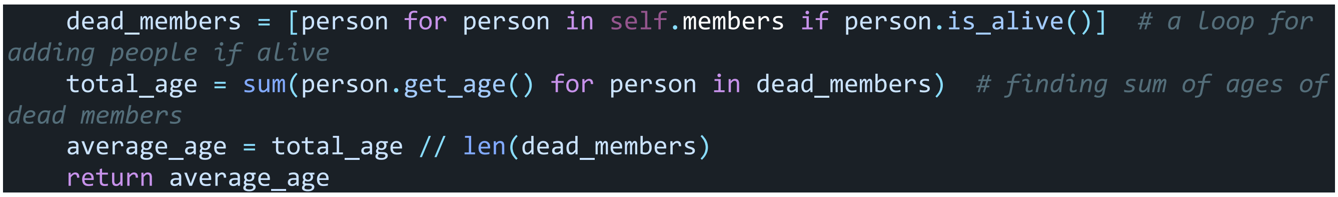
```python
def get_death_day(self):
    """"Return death day of the selected person if person is no longer alive."""
    if self._death_day == -1:
        return "Person is still alive."
    else:
        death_day, death_month, death_year = self.get_date(self._death_day)
        return death_day, death_month, death_year
```

```python
def get_age(self):
    """Return age of the selected person.
    Using current date/date of death and date of birth to calculate age of person."""
    current_day, current_month, current_year = self.current_datetime()  # converting into variables
    birth_day, birth_month, birth_year = self.get_birthday()  # converting into variables
    if self.is_alive():  # if the person is alive - the current date is subtracted by the date of birth.
        current_age_years = current_year - birth_year
        if (birth_month > current_month) or (birth_month == current_month and birth_day > current_day):
            current_age_years -= 1
    else:  # if the person is dead - the date of death is subtracted by the date of birth
        death_day, death_month, death_year = self.get_death_day()
        current_age_years = death_year - birth_year
        if (birth_month > death_month) or (birth_month == death_month and birth_day > death_day):
            current_age_years -= 1
    return current_age_years
```

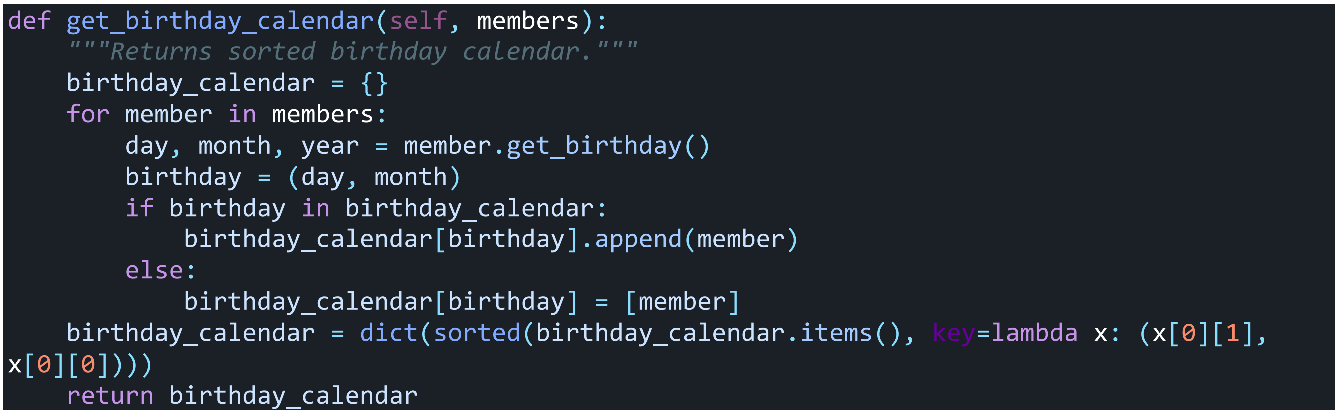
5.2 Class FamilyTree:

```python
class FamilyTree:
    members = []  # list of all members of the family tree

def average_age_per_person(self):
    """Calculates the average age per person."""
    total_members = len(self.members)
    total_age = sum(person.get_age() for person in self.members)  # adding all ages
    average_age = total_age // total_members
    return average_age
```

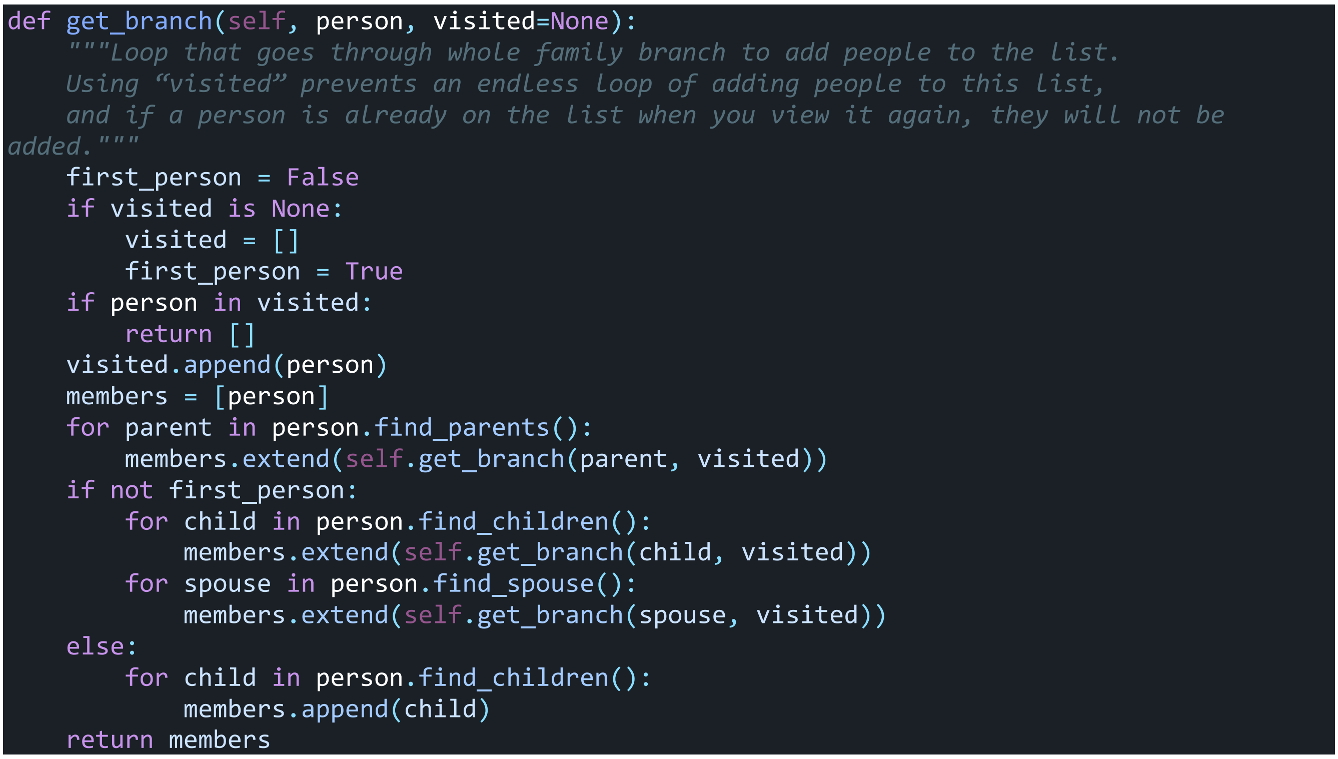
```python
def average_age_at_death(self):
    """Calculates the average age at death."""
```



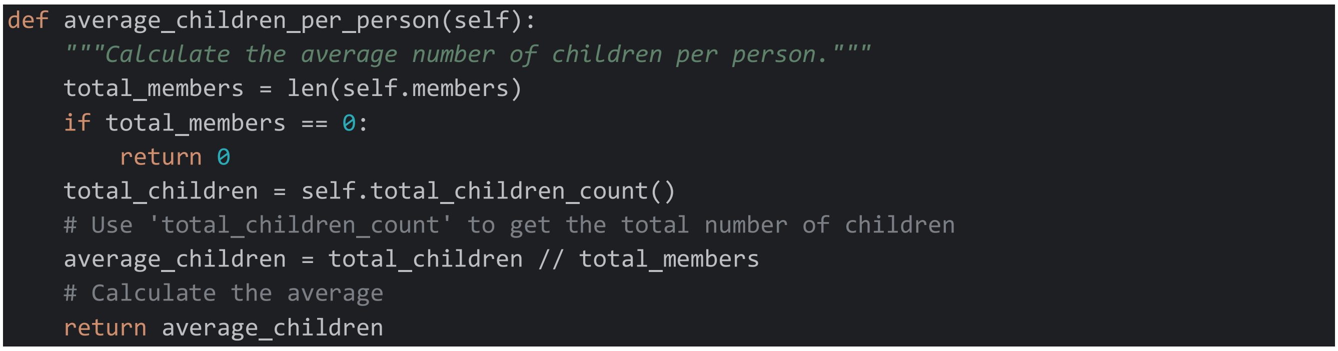
```
    dead_members = [person for person in self.members if person.is_alive()]  # a loop for
adding people if alive
    total_age = sum(person.get_age() for person in dead_members)  # finding sum of ages of
dead members
    average_age = total_age // len(dead_members)
    return average_age
```



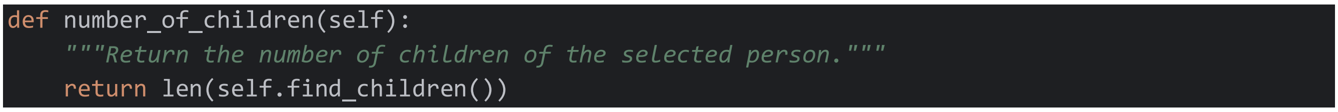
```
def get_birthday_calendar(self, members):
    """Returns sorted birthday calendar."""
    birthday_calendar = {}
    for member in members:
        day, month, year = member.get_birthday()
        birthday = (day, month)
        if birthday in birthday_calendar:
            birthday_calendar[birthday].append(member)
        else:
            birthday_calendar[birthday] = [member]
    birthday_calendar = dict(sorted(birthday_calendar.items(), key=lambda x: (x[0][1],
x[0][0])))
    return birthday_calendar
```



```
def get_branch(self, person, visited=None):
    """Loop that goes through whole family branch to add people to the list.
    Using “visited” prevents an endless loop of adding people to this list,
    and if a person is already on the list when you view it again, they will not be
added."""
    first_person = False
    if visited is None:
        visited = []
        first_person = True
    if person in visited:
        return []
    visited.append(person)
    members = [person]
    for parent in person.find_parents():
        members.extend(self.get_branch(parent, visited))
    if not first_person:
        for child in person.find_children():
            members.extend(self.get_branch(child, visited))
        for spouse in person.find_spouse():
            members.extend(self.get_branch(spouse, visited))
    else:
        for child in person.find_children():
            members.append(child)
    return members
```



```
def average_children_per_person(self):
    """Calculate the average number of children per person."""
    total_members = len(self.members)
    if total_members == 0:
        return 0
    total_children = self.total_children_count()
    # Use 'total_children_count' to get the total number of children
    average_children = total_children // total_members
    # Calculate the average
    return average_children
```



```
def number_of_children(self):
    """Return the number of children of the selected person."""
    return len(self.find_children())
```

6. TESTING

Describe the process you took to test your code and to make sure the program functions as required. ***Make sure you include a test plan and demonstrate thorough testing of your own code as well as the integrated code****.*

6.1 THIS SECTION SHOULD BE COMPLETED INDIVIDUALLY FOR F1 AND F2 AND AS A GROUP FOR F3.

6.1 F2

Test plan for F2…

Initially, we tested the code by simply calling each new function and checking if it worked correctly. After we had written most of the code, we decided to use unit testing, which has made a big difference to our code testing.

Test plan table:

Features	Test case	Inputs	Expected output	Actual output	Pass/Fail	Corrective Action
F2a(i)	find_siblings	Cornelia Emmersohn	Lina Chen, Adeline Chen	Lina Chen, Adeline Chen	pass	Valid

F2a(ii)	find_cousins	Cornelia Emmersohn	An Chen, Aki Chen, Guozhi Chen	An Chen, Aki Chen, Guozhi Chen	pass	Valid
F2b(i)	get_birthday	Melina Emmersohn	7, 3, 2008	7, 3, 2008	pass	Valid
F2b(ii)	get_birthday_calendar	Cornelia Emmersohn	(2, 2): Ara Song, etc.	(2, 2): Ara Song, etc.	pass	Valid
F3a(iii)	average_age_at_death	All members	48	48	pass	Valid
F3b(i)	average_children_per_person	All members	1	1	pass	Valid
F3b(ii)	number_of_children	Cornelia Emmersohn	2; 3	2; 3	pass	Valid

I tested most of my methods with ‘unittest’, all tests work stably and give a positive result when they are run.

These are examples of unittest:

Here we create a class for the unit test:

```
class TestFamilyTree(unittest.TestCase):
```

This function tests the ‘find_person_by_name’ method:

```
def test_find_person_by_name(self):
    person = family_tree.find_person_by_name("Cornelia Emmersohn")
    self.assertEqual(person.name, "Cornelia Emmersohn")
```

Tests the `find_siblings` method:

```
def test_find_siblings(self):
    siblings = family_tree.find_person_by_name("Cornelia Emmersohn").find_siblings()
    self.assertEqual(siblings, [family_tree.find_person_by_name("Lina Chen"),
                                family_tree.find_person_by_name("Adeline Chen")])
```

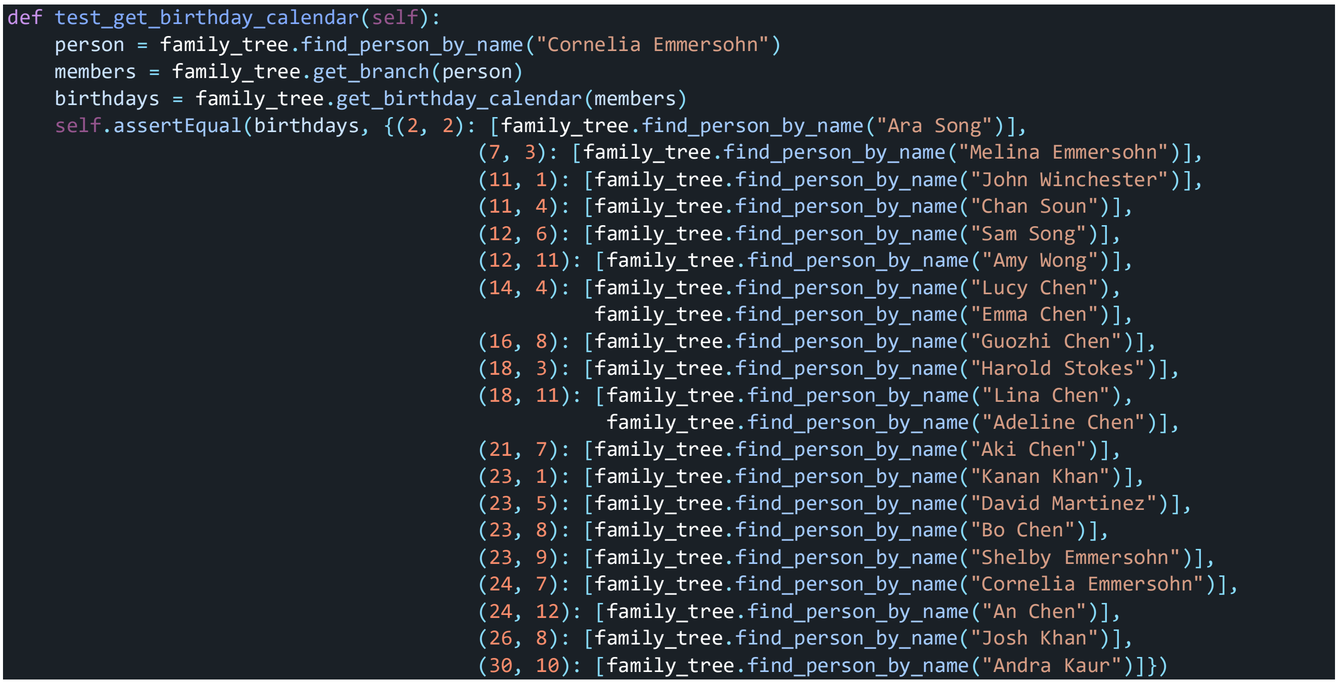
Tests the ‘find_cousins’ method:

```
def test_find_cousins(self):
    cousins = family_tree.find_person_by_name("Cornelia Emmersohn").find_cousins()
    self.assertEqual(cousins, [family_tree.find_person_by_name("An Chen"),
                               family_tree.find_person_by_name("Aki Chen"),
                               family_tree.find_person_by_name("Guozhi Chen")])
```

Tests the ‘get_birthday’ method:

```
def test_get_birthday(self):
    person = family_tree.find_person_by_name("Melina Emmersohn")
    birthday = person.get_birthday()
    self.assertEqual(birthday, (7, 3, 2008))
```

Tests the ‘get_birthday_calendar’ method:

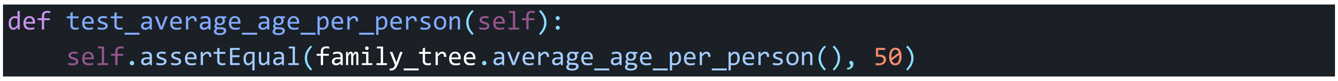


```python
def test_get_birthday_calendar(self):
    person = family_tree.find_person_by_name("Cornelia Emmersohn")
    members = family_tree.get_branch(person)
    birthdays = family_tree.get_birthday_calendar(members)
    self.assertEqual(birthdays, {(2, 2): [family_tree.find_person_by_name("Ara Song")],
                                        (7, 3): [family_tree.find_person_by_name("Melina Emmersohn")],
                                        (11, 1): [family_tree.find_person_by_name("John Winchester")],
                                        (11, 4): [family_tree.find_person_by_name("Chan Soun")],
                                        (12, 6): [family_tree.find_person_by_name("Sam Song")],
                                        (12, 11): [family_tree.find_person_by_name("Amy Wong")],
                                        (14, 4): [family_tree.find_person_by_name("Lucy Chen"),
                                                  family_tree.find_person_by_name("Emma Chen")],
                                        (16, 8): [family_tree.find_person_by_name("Guozhi Chen")],
                                        (18, 3): [family_tree.find_person_by_name("Harold Stokes")],
                                        (18, 11): [family_tree.find_person_by_name("Lina Chen"),
                                                   family_tree.find_person_by_name("Adeline Chen")],
                                        (21, 7): [family_tree.find_person_by_name("Aki Chen")],
                                        (23, 1): [family_tree.find_person_by_name("Kanan Khan")],
                                        (23, 5): [family_tree.find_person_by_name("David Martinez")],
                                        (23, 8): [family_tree.find_person_by_name("Bo Chen")],
                                        (23, 9): [family_tree.find_person_by_name("Shelby Emmersohn")],
                                        (24, 7): [family_tree.find_person_by_name("Cornelia Emmersohn")],
                                        (24, 12): [family_tree.find_person_by_name("An Chen")],
                                        (26, 8): [family_tree.find_person_by_name("Josh Khan")],
                                        (30, 10): [family_tree.find_person_by_name("Andra Kaur")]})
```

6.2 F3

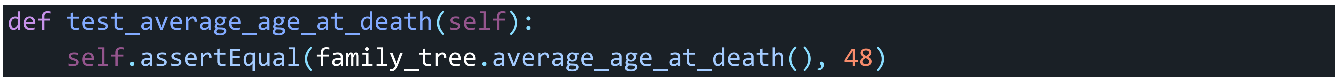
Test plan for F3…

Tests the ‘average_age_per_person’ method:



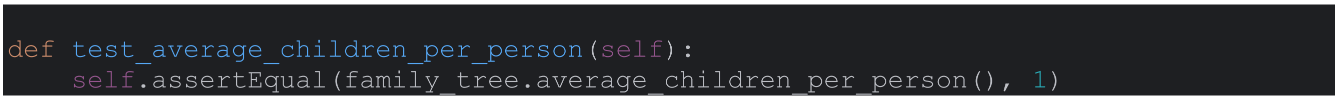
```python
def test_average_age_per_person(self):
    self.assertEqual(family_tree.average_age_per_person(), 50)
```

Tests the ‘average_age_at_death’ method:



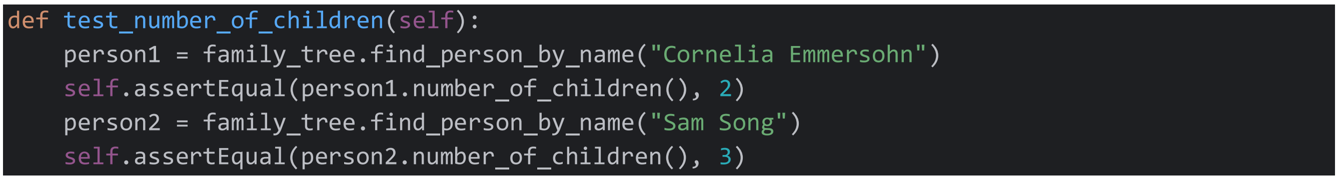
```python
def test_average_age_at_death(self):
    self.assertEqual(family_tree.average_age_at_death(), 48)
```

Tests the ‘find_average_children_per_person’ method:



```python
def test_average_children_per_person(self):
    self.assertEqual(family_tree.average_children_per_person(), 1)
```

Tests the ‘number_of_children’ method:



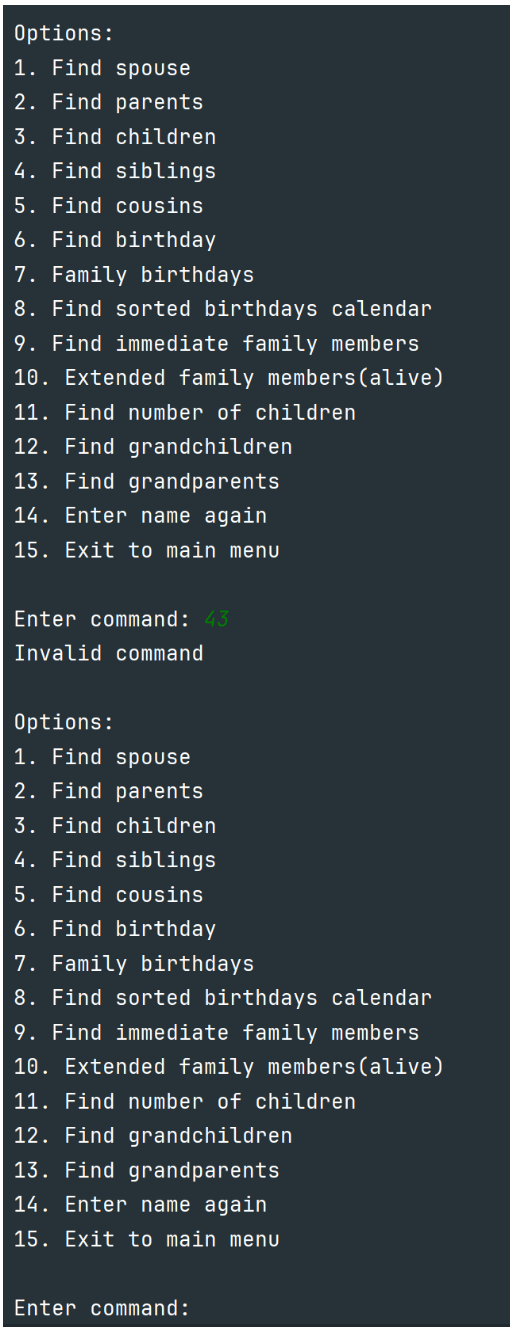
```python
def test_number_of_children(self):
    person1 = family_tree.find_person_by_name("Cornelia Emmersohn")
    self.assertEqual(person1.number_of_children(), 2)
    person2 = family_tree.find_person_by_name("Sam Song")
    self.assertEqual(person2.number_of_children(), 3)
```

This is a function to call all unit test methods:

```
if __name__ == "__main__":
    unittest.main()
```

If we check the code manually, you will see that the code will not contain errors when entering incorrect values into the terminal. And the code will always remain in operation.

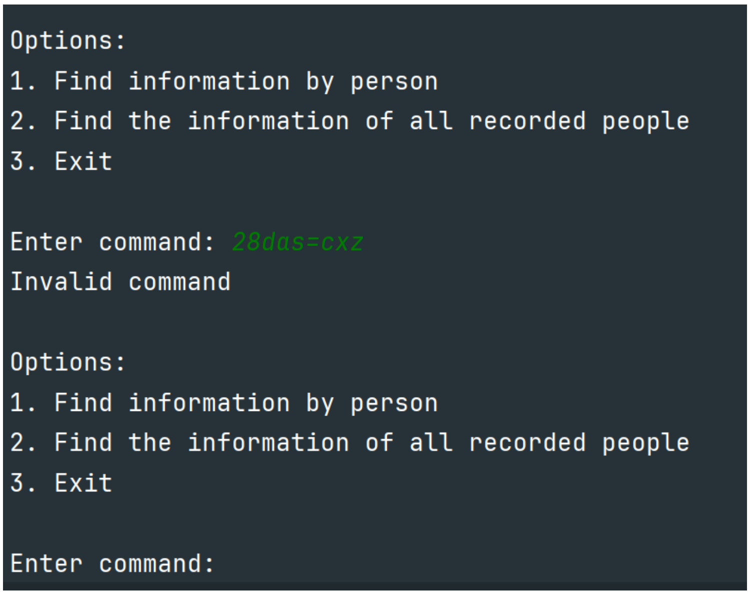
And here's what happens if I enter something wrong:



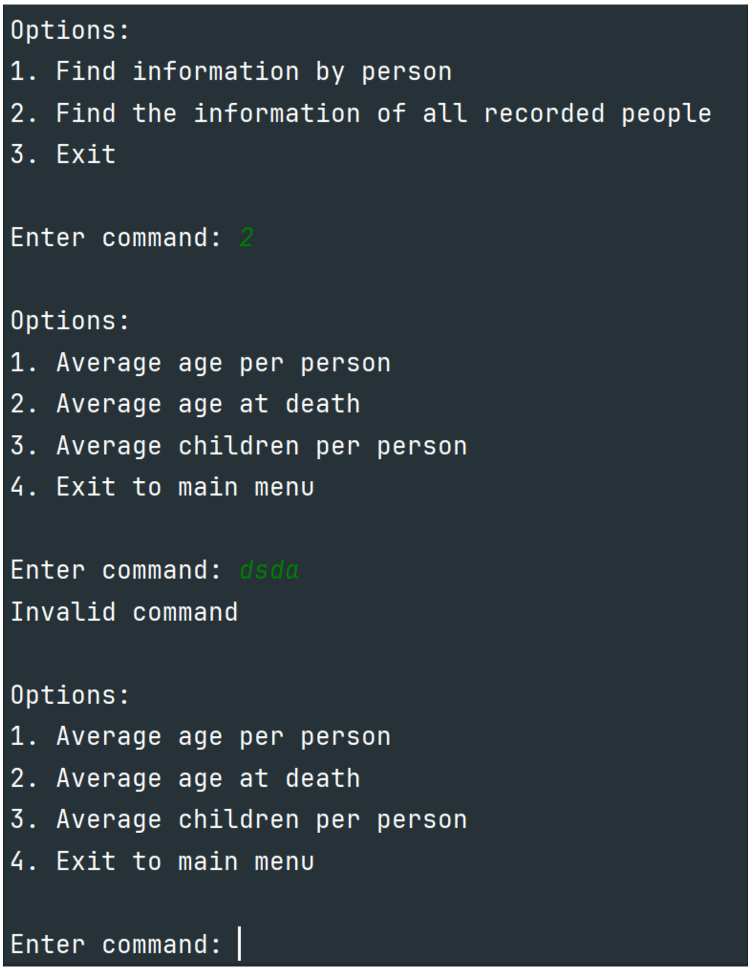
```
Options:
1. Find spouse
2. Find parents
3. Find children
4. Find siblings
5. Find cousins
6. Find birthday
7. Family birthdays
8. Find sorted birthdays calendar
9. Find immediate family members
10. Extended family members(alive)
11. Find number of children
12. Find grandchildren
13. Find grandparents
14. Enter name again
15. Exit to main menu

Enter command: 43
Invalid command

Options:
1. Find spouse
2. Find parents
3. Find children
4. Find siblings
5. Find cousins
6. Find birthday
7. Family birthdays
8. Find sorted birthdays calendar
9. Find immediate family members
10. Extended family members(alive)
11. Find number of children
12. Find grandchildren
13. Find grandparents
14. Enter name again
15. Exit to main menu

Enter command:
```

So the code will continue to work in any case.

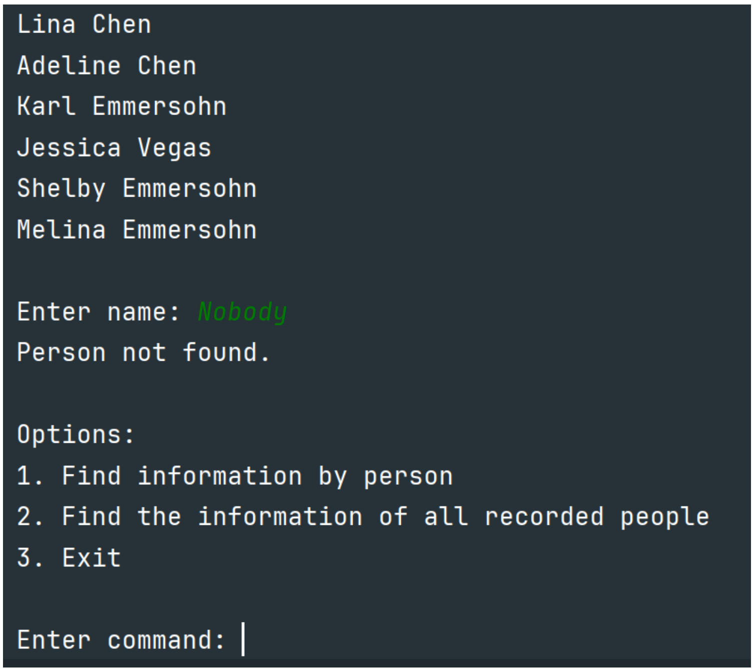
Here is more examples:



```
Options:
1. Find information by person
2. Find the information of all recorded people
3. Exit

Enter command: 28das=cxz
Invalid command

Options:
1. Find information by person
2. Find the information of all recorded people
3. Exit

Enter command:
```

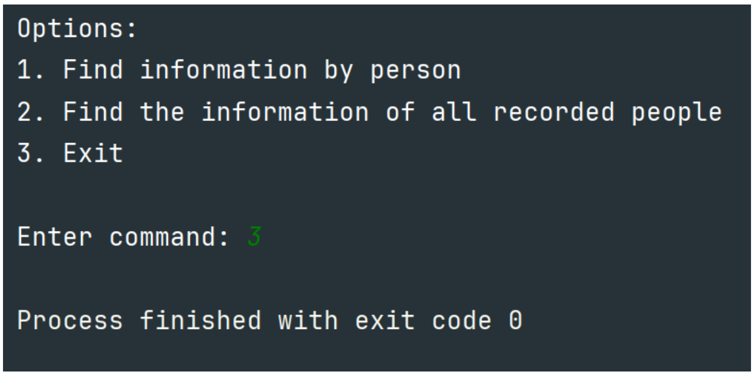


```
Options:
1. Find information by person
2. Find the information of all recorded people
3. Exit

Enter command: 2

Options:
1. Average age per person
2. Average age at death
3. Average children per person
4. Exit to main menu

Enter command: dsda
Invalid command

Options:
1. Average age per person
2. Average age at death
3. Average children per person
4. Exit to main menu

Enter command:
```



```
Lina Chen
Adeline Chen
Karl Emmersohn
Jessica Vegas
Shelby Emmersohn
Melina Emmersohn

Enter name: Nobody
Person not found.

Options:
1. Find information by person
2. Find the information of all recorded people
3. Exit

Enter command: 
```

The code will stop working only when we exit it ourselves:



```
Options:
1. Find information by person
2. Find the information of all recorded people
3. Exit

Enter command: 3

Process finished with exit code 0
```

7. Annotated Screenshots Demonstrating Implementation

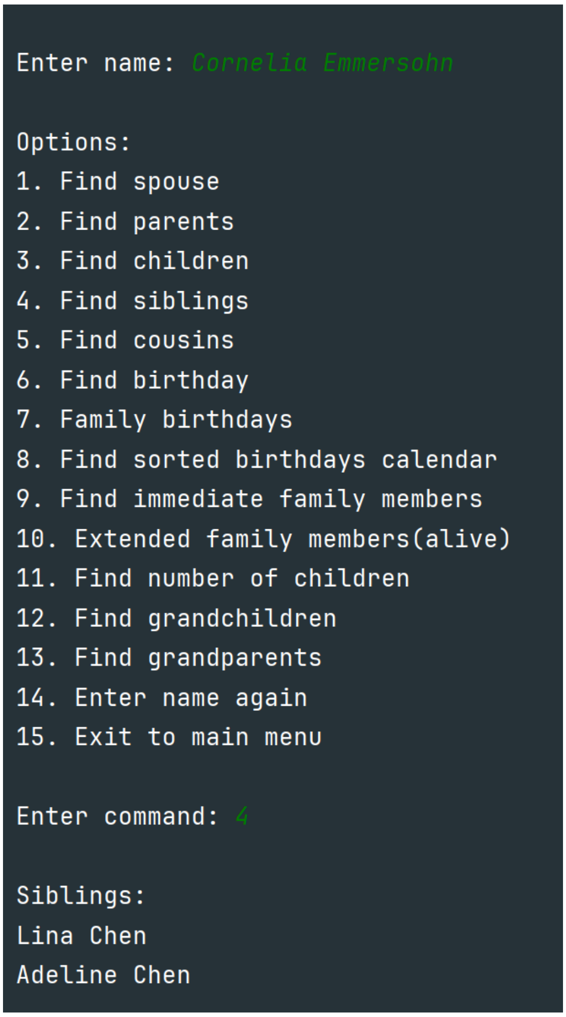
Provide screenshots that demonstrate the features implemented running – i.e. showing the output produced by all of the subfeatures. Annotate each screenshot and if necessary, provide a brief description for ***each*** *(****up to 100 words****) to explain the code in action.*

THIS SECTION SHOULD BE COMPLETED INDIVIDUALLY FOR F1 AND F2 AND AS A GROUP FOR F3.

7.1 Feature F2

i. F2a.i- Screenshots …

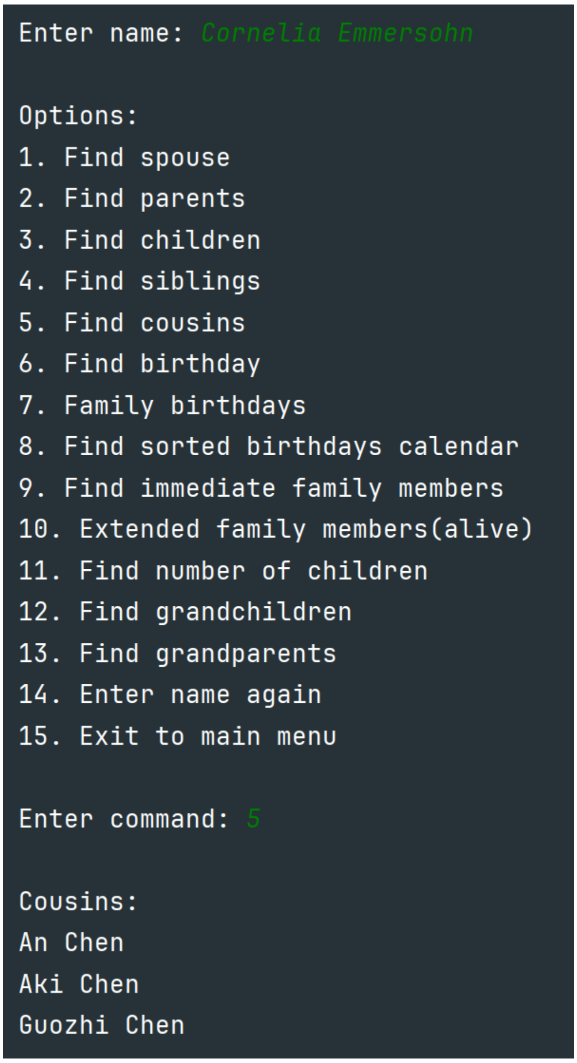
This is an example of how the 'Find siblings' function works. For this result, we need to select 'Find information by person' in the menu and enter Cornelia's full name into the terminal.



```
Enter name: Cornelia Emmersohn

Options:
1. Find spouse
2. Find parents
3. Find children
4. Find siblings
5. Find cousins
6. Find birthday
7. Family birthdays
8. Find sorted birthdays calendar
9. Find immediate family members
10. Extended family members(alive)
11. Find number of children
12. Find grandchildren
13. Find grandparents
14. Enter name again
15. Exit to main menu

Enter command: 4

Siblings:
Lina Chen
Adeline Chen
```

ii. F2a.ii- Screenshots …

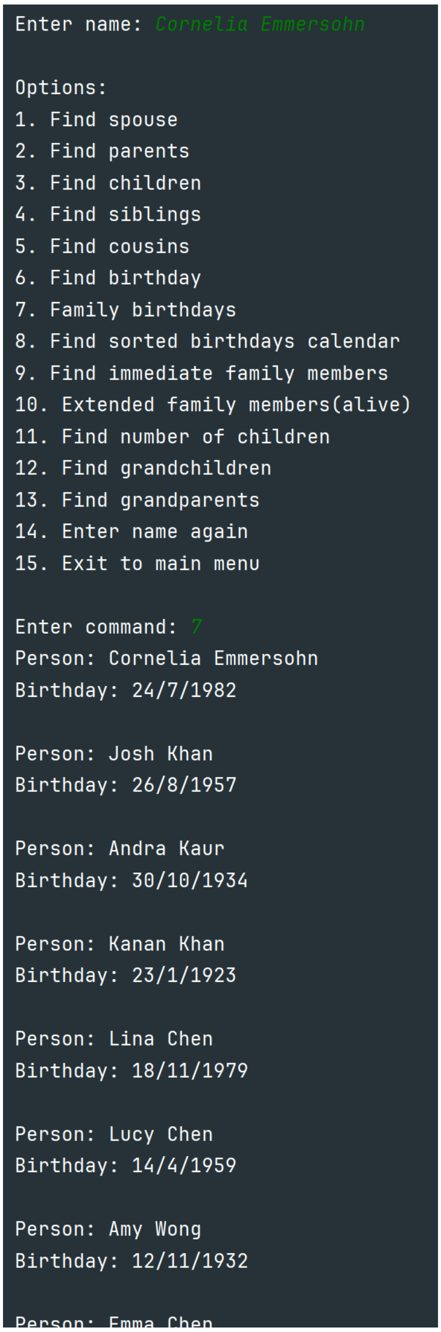
Example of how 'Find cousins' function works. We can find this function in the same menu as 'Find siblings'.



```
Enter name: Cornelia Emmersohn

Options:
1. Find spouse
2. Find parents
3. Find children
4. Find siblings
5. Find cousins
6. Find birthday
7. Family birthdays
8. Find sorted birthdays calendar
9. Find immediate family members
10. Extended family members(alive)
11. Find number of children
12. Find grandchildren
13. Find grandparents
14. Enter name again
15. Exit to main menu

Enter command: 5

Cousins:
An Chen
Aki Chen
Guozhi Chen
```

iii. F2B.I- SCREENSHOTS …

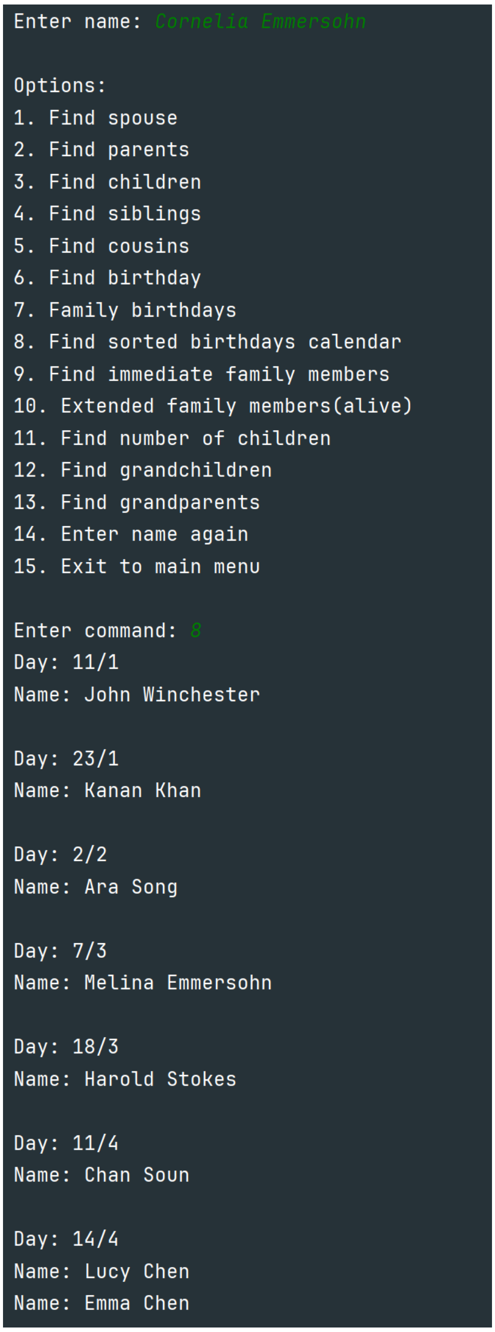
‘Family birthdays’ function determines the birthdays of all members of my branch.



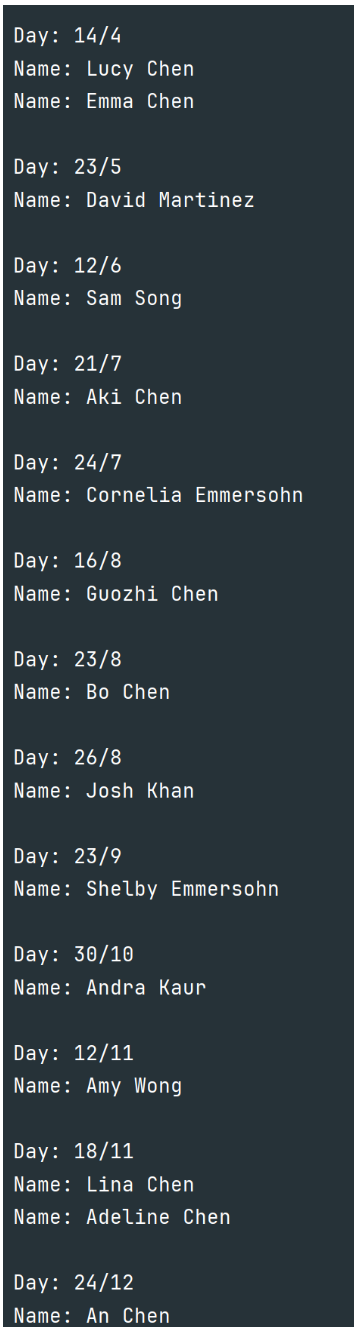
```
Enter name: Cornelia Emmersohn

Options:
1. Find spouse
2. Find parents
3. Find children
4. Find siblings
5. Find cousins
6. Find birthday
7. Family birthdays
8. Find sorted birthdays calendar
9. Find immediate family members
10. Extended family members(alive)
11. Find number of children
12. Find grandchildren
13. Find grandparents
14. Enter name again
15. Exit to main menu

Enter command: 7
Person: Cornelia Emmersohn
Birthday: 24/7/1982

Person: Josh Khan
Birthday: 26/8/1957

Person: Andra Kaur
Birthday: 30/10/1934

Person: Kanan Khan
Birthday: 23/1/1923

Person: Lina Chen
Birthday: 18/11/1979

Person: Lucy Chen
Birthday: 14/4/1959

Person: Amy Wong
Birthday: 12/11/1932

Person: Emma Chen
```

iv. F2B.II- SCREENSHOTS …

‘Find sorted birthdays calendar’ function finds sorted birthdays in ascending order. The output table contains only days and months. And if two people have the same birthday, they are recorded under the same birthday.



```
Enter name: Cornelia Emmersohn

Options:
1. Find spouse
2. Find parents
3. Find children
4. Find siblings
5. Find cousins
6. Find birthday
7. Family birthdays
8. Find sorted birthdays calendar
9. Find immediate family members
10. Extended family members(alive)
11. Find number of children
12. Find grandchildren
13. Find grandparents
14. Enter name again
15. Exit to main menu

Enter command: 8
Day: 11/1
Name: John Winchester

Day: 23/1
Name: Kanan Khan

Day: 2/2
Name: Ara Song

Day: 7/3
Name: Melina Emmersohn

Day: 18/3
Name: Harold Stokes

Day: 11/4
Name: Chan Soun

Day: 14/4
Name: Lucy Chen
Name: Emma Chen
```

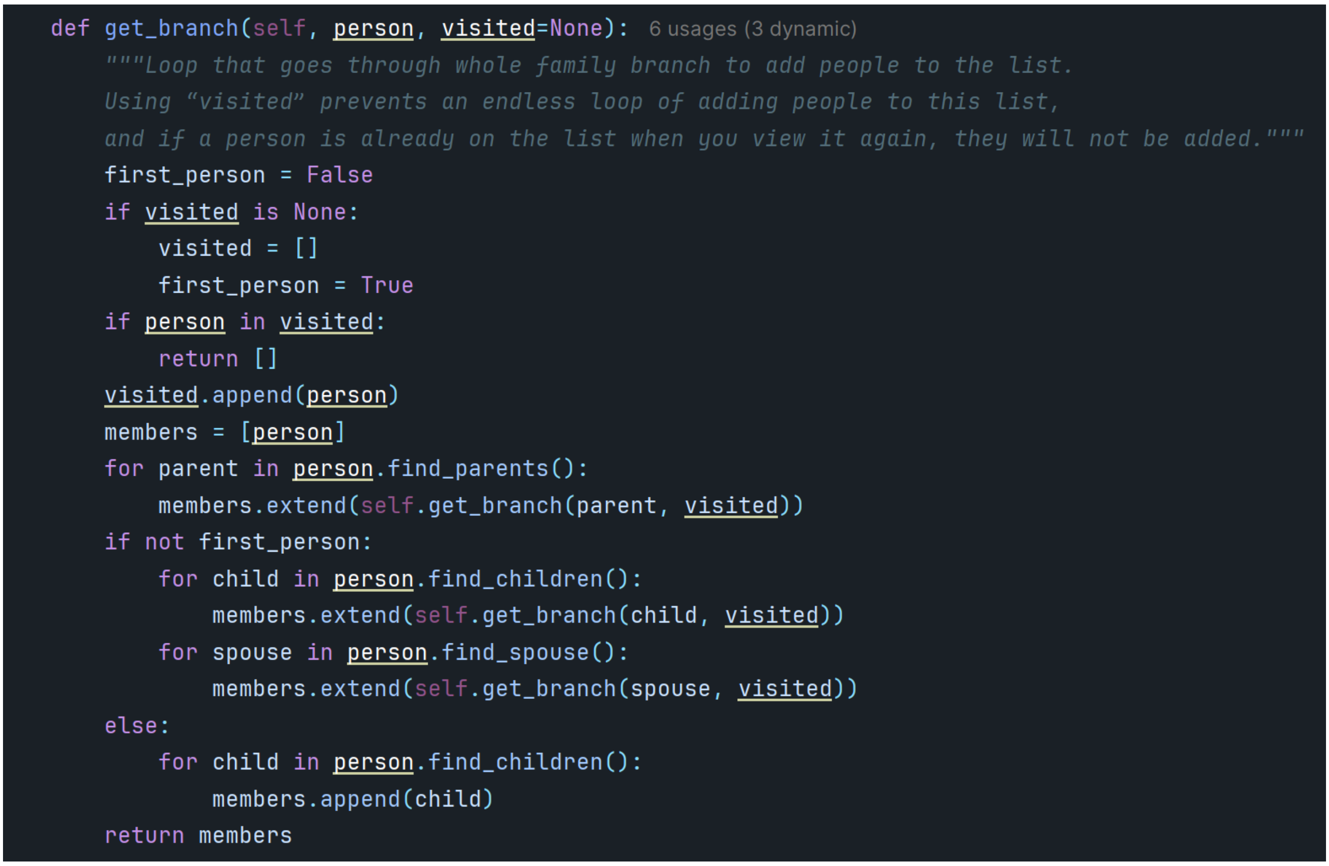


```
Day: 14/4
Name: Lucy Chen
Name: Emma Chen

Day: 23/5
Name: David Martinez

Day: 12/6
Name: Sam Song

Day: 21/7
Name: Aki Chen

Day: 24/7
Name: Cornelia Emmersohn

Day: 16/8
Name: Guozhi Chen

Day: 23/8
Name: Bo Chen

Day: 26/8
Name: Josh Khan

Day: 23/9
Name: Shelby Emmersohn

Day: 30/10
Name: Andra Kaur

Day: 12/11
Name: Amy Wong

Day: 18/11
Name: Lina Chen
Name: Adeline Chen

Day: 24/12
Name: An Chen
```

7.2 Feature F3

i. F3a.i- screenshots ...

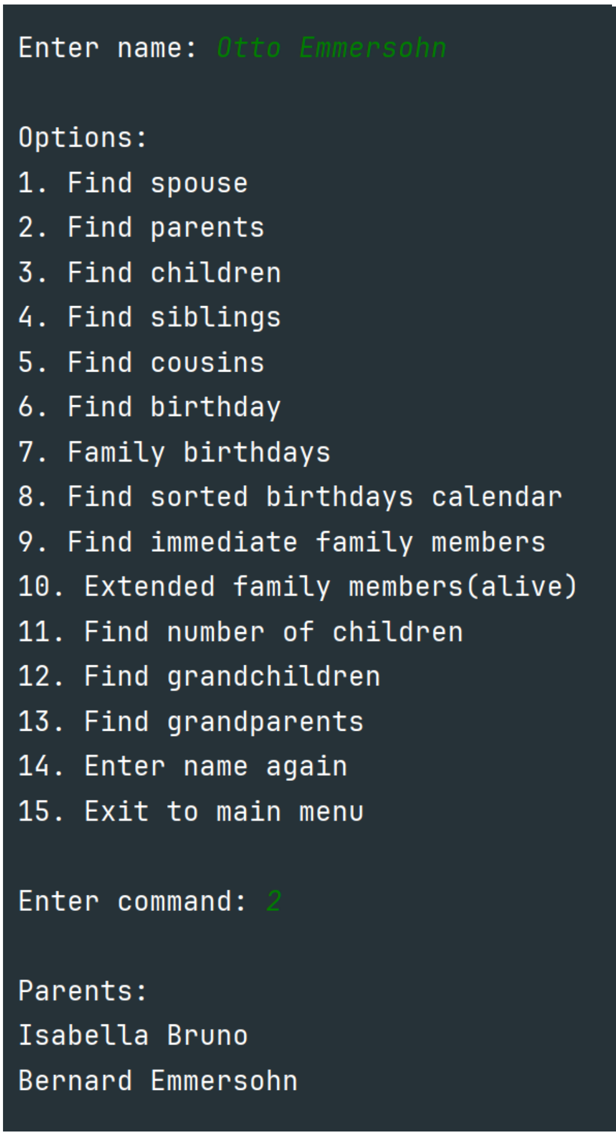
We decided to combine our branches at the very beginning of our code, so the entire family tree including the two branches is in one list. So we created a function to find the branch of only one person. Below is a screenshot of the function.



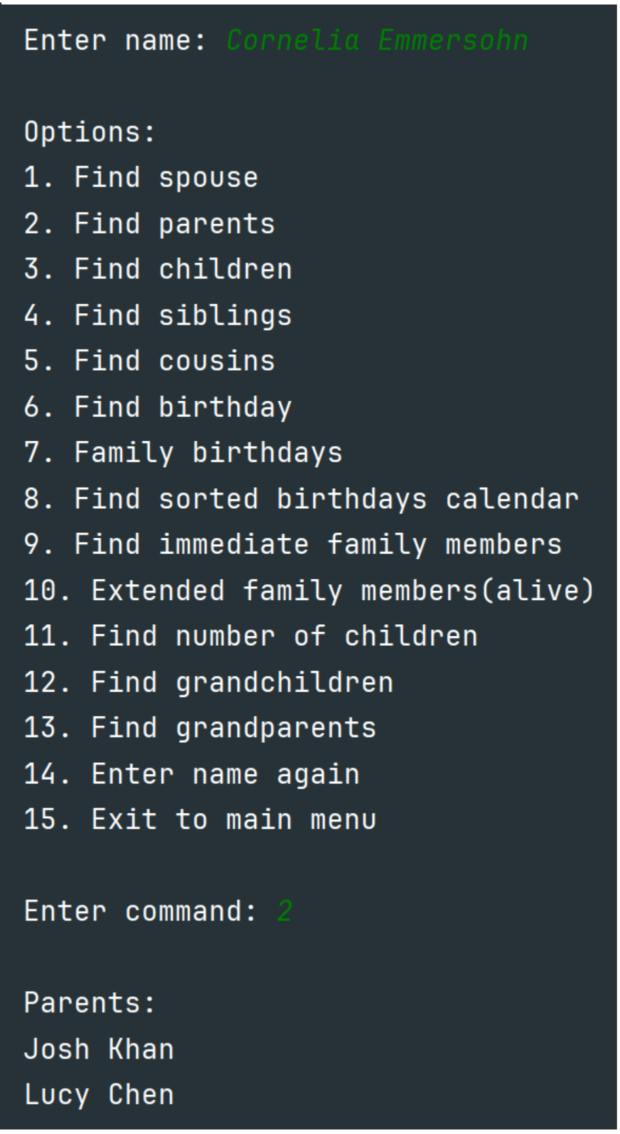
```
def get_branch(self, person, visited=None):  6 usages (3 dynamic)
    """Loop that goes through whole family branch to add people to the list.
    Using “visited” prevents an endless loop of adding people to this list,
    and if a person is already on the list when you view it again, they will not be added."""
    first_person = False
    if visited is None:
        visited = []
        first_person = True
    if person in visited:
        return []
    visited.append(person)
    members = [person]
    for parent in person.find_parents():
        members.extend(self.get_branch(parent, visited))
    if not first_person:
        for child in person.find_children():
            members.extend(self.get_branch(child, visited))
        for spouse in person.find_spouse():
            members.extend(self.get_branch(spouse, visited))
    else:
        for child in person.find_children():
            members.append(child)
    return members
```

ii. F3a.ii- screenshots ...

Given that our branches are combined, we can choose any person from both branches. These screenshots show Cornelia and Otto's parents from completely different branches.



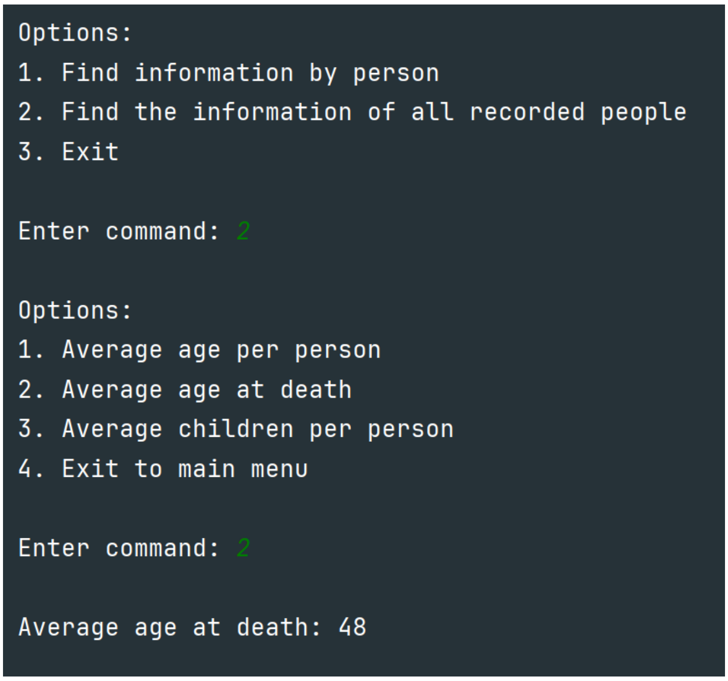
```
Enter name: Otto Emmersohn

Options:
1. Find spouse
2. Find parents
3. Find children
4. Find siblings
5. Find cousins
6. Find birthday
7. Family birthdays
8. Find sorted birthdays calendar
9. Find immediate family members
10. Extended family members(alive)
11. Find number of children
12. Find grandchildren
13. Find grandparents
14. Enter name again
15. Exit to main menu

Enter command: 2

Parents:
Isabella Bruno
Bernard Emmersohn
```



```
Enter name: Cornelia Emmersohn

Options:
1. Find spouse
2. Find parents
3. Find children
4. Find siblings
5. Find cousins
6. Find birthday
7. Family birthdays
8. Find sorted birthdays calendar
9. Find immediate family members
10. Extended family members(alive)
11. Find number of children
12. Find grandchildren
13. Find grandparents
14. Enter name again
15. Exit to main menu

Enter command: 2

Parents:
Josh Khan
Lucy Chen
```

iii. F3A.III- SCREENSHOTS …

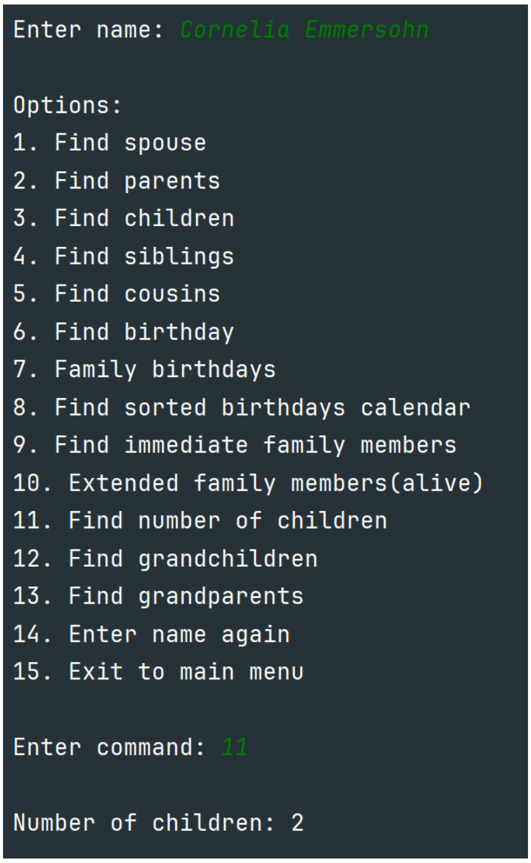
‘Average age at death’ function which is located in second option, finds the average age of people at death.



```
Options:
1. Find information by person
2. Find the information of all recorded people
3. Exit

Enter command: 2

Options:
1. Average age per person
2. Average age at death
3. Average children per person
4. Exit to main menu

Enter command: 2

Average age at death: 48
```

iv. F3B.I- SCREENSHOTS ...

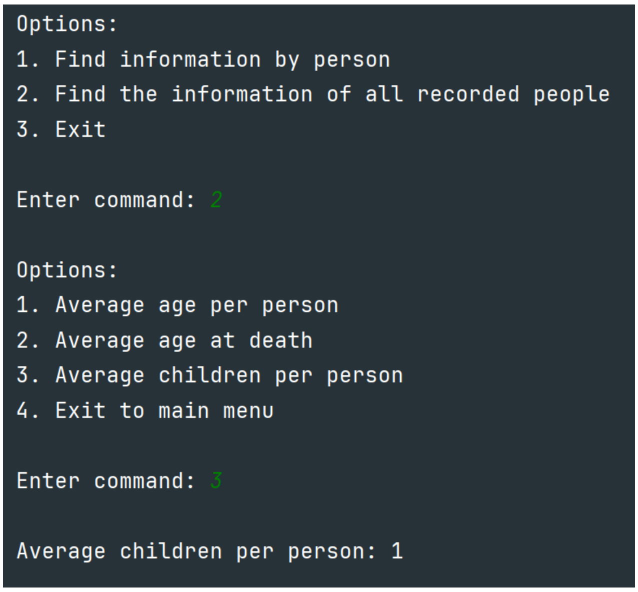
‘Find number of children’ function finds number of children of selected individual.



```
Enter name: Cornelia Emmersohn

Options:
1. Find spouse
2. Find parents
3. Find children
4. Find siblings
5. Find cousins
6. Find birthday
7. Family birthdays
8. Find sorted birthdays calendar
9. Find immediate family members
10. Extended family members(alive)
11. Find number of children
12. Find grandchildren
13. Find grandparents
14. Enter name again
15. Exit to main menu

Enter command: 11

Number of children: 2
```

v. F3B.II- SCREENSHOTS ...

‘Average children per person’ function.



```
Options:
1. Find information by person
2. Find the information of all recorded people
3. Exit

Enter command: 2

Options:
1. Average age per person
2. Average age at death
3. Average children per person
4. Exit to main menu

Enter command: 3

Average children per person: 1
```

8. OpenAI Comparison

Provide any code generated using OpenAI along with a listing of the code you initially wrote from scratch in a table showing the generated and your code side-by-side for each feature. Examine and explain the generated code's design, describing its quality and efficiency compared to the initial code you wrote. The narrative must also describe how you used the generated code to improve your own code or describe how the generated code may be improved.

9. SELF-ASSESSMENT

Please assess yourself objectively for each section shown below and then enter the total mark you expect to get. Marks for each assessment criterion are indicated between parentheses.

Code development (70)

a. Features Implemented [40] (group work and integration will be assessed here)

Partner A or Partner B features (up to 20)

Sub-features have not been implemented – 0
Attempted, not complete or very buggy – 1 to 5
Implemented and functioning without errors but not integrated – 6 to 10
Implemented and fully integrated but buggy – 11 to 16
Implemented, fully integrated and functioning without errors – 17 to 20

Group Features (up to 20)

Sub-features has not been implemented – 0
Attempted, not complete or very buggy – 1 to 3
Implemented and functioning without errors but not integrated – 4 to 8
Implemented and fully integrated but buggy – 9 to 14
Implemented, fully integrated and functioning without errors – 15 to 20

For this criterion I think I got: 40 out of 40

b. Use of OOP techniques [20]

Abstraction (up to 7)

No classes have been created – 0
Classes have been created superficially and not instantiated or used – 1
Classes have been created but only some have been instantiated and used – 2 or 3
Useful classes and objects have been created and used correctly – 4 or 5
The use of classes and objects exceeds the specification – 6 or 7

Encapsulation (up to 7)

No encapsulation has been used – 0
Class variables and methods have been encapsulated superficially – 1 to 3
Class variables and methods have been encapsulated correctly – 4 to 6
The use of encapsulation exceeds the specification – 6 to 8

Inheritance or polymorphism (up to 6)

No inheritance or polymorphism has been used – 0
Inheritance or polymorphism has been used superficially – – 1 or 2
Inheritance or polymorphism has been used correctly – 3 or 4
The use of inheritance or polymorphism exceeds the specification – 5 or 6

For this criterion I think I got: 15 out of 20

c. Quality of Code [10]

Code Duplication (up to 4)

Code contains too many unnecessary code repetition – 0
Regular occurrences of duplicate code – 1
Occasional duplicate code – 2
Very little duplicate code – 3
No duplicate code – 4

PEP8 Conventions and naming of variables, methods and classes (up to 3)

PEP8 and naming convention has not been used – 0
PEP8 and naming convention has been used occasionally – 1
PEP8 and naming convention has been used regularly – 2
PEP8 convention used professionally and all items have been named correctly – 3

In-code Comments (up to 3)

No in-code comments – 0
Code contains occasional in-code comments – 1
Code contains useful and regular in-code comments – 2
Thoroughly commented, good use of docstrings, and header comments describing.py files – 3

For this criterion I think I got: 8 out of 10

2. Documentation (20)

Design (up to 10) clear exposition about the design and decisions for OOP use

The documentation cannot be understood on first reading or is mostly incomplete – 0
The documentation is readable, but a section(s) are missing – 1 to 3
The documentation is complete – 4 to 6
The documentation is complete and of a high standard – 7 to 10

Testing (10)

Testing has not been demonstrated in the documentation – 0
A test plan has been included but is incomplete – 1 or 2
A test plan has been included with some appropriate test cases – 3 to 6
A full test plan has been included with thorough test cases and evidence of carrying it out – 7 to 10

For this criterion I think I got: 15 out of 20

3. Screencast - Acceptance Test (10)

Recorded demonstration of code and accompanying explanatory commentary (up to 10)

Not submitted or no work demonstrated or lacking commentary – 0
Work demonstrated not to expected standard, unclear commentary, superficial team contribution – 1 to 3
Work demonstrated to expected standard, sufficient commentary and team contribution – 4 to 7
Work demonstrated exceeded the standard expected – 8 to 10

For this criterion I think I got: 6 out of 10

I think my overall mark would be: 84 out of 100

Appendix A: Code Listing

*Provide a complete listing of all the *.py files in your PyCharm project. Make sure your code is well commented and applies professional Python convention (refer to PEP 8 for details). The code listed here must match that uploaded to Moodle. Please copy and paste the actual code – no screenshots please! You will lose marks if screenshots are provided instead of code. Clearly label the parts each partner created with their name and SID.*

These are the colors of work done will be selected in code below:
Mykhaylo Zhyhan, ID: 001398148 - Red
Brendon Shymanskyi, ID: 001419181 - Blue
Together - Green

File ‘main.py’:

```
# this .py file is created in order to embed the main functions of the application, which transport functions from other .py files

from list import family_tree


def find_spouse(family_tree, name):
    """Output spouse of a person."""
    spouse = family_tree.find_person_by_name(name).find_spouse()
    if spouse:
        print("\nSpouse:")
        print(*[f"{spouse.name}" for spouse in spouse], sep='\n')  # Print the names of the spouse separated(sep='\n') by a newline.
    else:
        print("\nNo spouse found.")


def print_parents(family_tree, name):
    """Output parents of a person."""
    parents = family_tree.find_person_by_name(name).find_parents()
    if parents:
        print("\nParents:")
        print(*[f"{parent.name}" for parent in parents], sep='\n')  # Print the names of the parents separated(sep='\n') by a newline.
    else:
        print("\nNo parents found.")


def find_children(family_tree, name):
    """Output children of a person."""
    children = family_tree.find_person_by_name(name).find_children()
    if children:
        print("\nChildren:")
        print(*[f"{child.name}" for child in children], sep='\n')  # Print the names of the children
separated(sep='\n') by a newline.
    else:
        print("\nNo children found.")


def find_siblings(family_tree, name):
    """Output siblings of a person."""
    siblings = family_tree.find_person_by_name(name).find_siblings()
    if siblings:
        print("\nSiblings:")
        print(*[f"{sibling.name}" for sibling in siblings], sep='\n')  # Print the names of the siblings
separated(sep='\n') by a newline.
    else:
        print("No siblings found.")


def find_cousins(family_tree, name):
    """Output cousins of a person."""
    cousins = family_tree.find_person_by_name(name).find_cousins()  # Variable for cousins, created to record
cousins.
    if cousins:
        print("\nCousins:")
        print(*[f"{cousin.name}" for cousin in cousins], sep='\n')  # Print the names of the siblings
separated(sep='\n') by a newline.
    else:
        print("No cousins found.")


def find_birthday(family_tree, name):
    """Output birthday of a person."""
    birthday = family_tree.find_person_by_name(name).get_birthday()
    if birthday:
        print(f"\nBirthday: {birthday[0]}/{birthday[1]}/{birthday[2]}")  # Outputs birthday if is known.
    else:
        print("No birthday found.")


def family_birthdays_in_branch(family_tree, name):
    """Output birthdays of all people in a branch."""
    person = family_tree.find_person_by_name(name)  # creates a variable for person
    branch = family_tree.get_branch(person)
    for member in branch:
        day, month, year = member.get_birthday()  # Variable for day, month, year - helps to display birthdays
individually.
        print(f"Person: {member.name}")
        print(f"Birthday: {day}/{month}/{year}")
        print("")


def find_sorted_birthdays_calendar(family_tree, name):
    """Output birthdays calendar sorted by month and day only."""
    person = family_tree.find_person_by_name(name)  # creates a variable for person
    branch = family_tree.get_branch(person)
    calendar = family_tree.get_birthday_calendar(branch)
    for date in calendar:
        print(f"Day: {date[0]}/{date[1]}")  # Prints the day and month of birth using the selected values in the
array.
        for member in calendar[date]:  # Prints the member's name in members of branch.
            print(f"Name: {member.name}")
        print("")


def find_immediate_family_members(family_tree, name):
    """Output immediate family members of a person."""
    person = family_tree.find_person_by_name(name)  # creates a variable for person
    parents, spouse, children, siblings = person.immediate_family_members()  # Search for a family members
and assign to a variables
    if parents:
        print("\nParents:")
        print(*[f"{parent.name}" for parent in parents], sep='\n')  # Print the names of the parents separated by a
newline.
    if spouse:
        print("\nSpouse:")
        print(*[f"{spouse.name}" for spouse in spouse], sep='\n')
    if children:
        print("\nChildren:")
        print(*[f"{children.name}" for children in children], sep='\n')
    if siblings:
        print("\nSiblings:")
        print(*[f"{siblings.name}" for siblings in siblings], sep='\n')


def extended_family_members(family_tree, name):
    """Output extended family members of a person."""
    person = family_tree.find_person_by_name(name)  # creates a variable for person
    extended_family = person.extended_family_members()  # variable for extended family
    if extended_family:
        print("\nExtended Family:")
        print(*[f"{member.name}" for member in extended_family], sep='\n')
    else:
        print("No extended family found.")
```

```
def number_of_children(family_tree, name):
    """Output number of children of a person."""
    children = family_tree.find_person_by_name(name).number_of_children()
    if children:
        print(f"\nNumber of children: {children}")
    else:
        print("No children found.")


def find_grandchildren(family_tree, name):
    """Output grandchildren of a person."""
    person = family_tree.find_person_by_name(name)  # creates a variable for person
    grandchildren = person.find_grandchildren()
    if grandchildren:
        print("\nGrandchildren:")
        print(*[f"Name: {grandchild.name}" for grandchild in grandchildren], sep='\n')
    else:
        print("No grandchildren found.")


def find_grandparents(family_tree, name):
    """Output grandparents of a person."""
    person = family_tree.find_person_by_name(name)  # creates a variable for person
    grandparents = person.find_grandparents()
    if grandparents:
        print("\nGrandparents:")
        print(*[f"{grandparent.name}" for grandparent in grandparents], sep='\n')
    else:
        print("No grandparents found.")


def average_age(family_tree):
    """Return average age of all people in the family."""
    return f"\nAverage age: {family_tree.average_age_per_person()}"


def average_age_at_death(family_tree):
    """Return average age at death of all people in the family."""
    return f"\nAverage age at death: {family_tree.average_age_at_death()}"


def average_children(family_tree):
    """Return average number of children per person."""
    return f"\nAverage children per person: {family_tree.average_children_per_person()}"


# array of options for the first/start menu
# lambda is used to call functions that were written above
start_options = [
    ["Find information by person", lambda: use_options()],
    ["Find the information of all recorded people", lambda: family_menu()],
    ["Exit", lambda: exit()],
]


# this array contains the options used to find family relationships associated with the selected person
member_options = [
    ["Find spouse", lambda name: find_spouse(family_tree, name)],
    ["Find parents", lambda name: print_parents(family_tree, name)],
    ["Find children", lambda name: find_children(family_tree, name)],
    ["Find siblings", lambda name: find_siblings(family_tree, name)],
    ["Find cousins", lambda name: find_cousins(family_tree, name)],
    ["Find birthday", lambda name: find_birthday(family_tree, name)],
    ["Family birthdays", lambda name: family_birthdays_in_branch(family_tree, name)],
    ["Find sorted birthdays calendar", lambda name: find_sorted_birthdays_calendar(family_tree, name)],
    ["Find immediate family members", lambda name: find_immediate_family_members(family_tree, name)],
    ["Extended family members(alive)", lambda name: extended_family_members(family_tree, name)],
    ["Find number of children", lambda name: number_of_children(family_tree, name)],
    ["Find grandchildren", lambda name: find_grandchildren(family_tree, name)],
    ["Find grandparents", lambda name: find_grandparents(family_tree, name)],
    ["Enter name again", lambda name: use_options()],
    ["Exit to main menu", lambda name: start_menu()],
]


# array contains the options wich calculates averages per person
family_options = [
    ["Average age per person", lambda: print(average_age(family_tree))],
    ["Average age at death", lambda: print(average_age_at_death(family_tree))],
    ["Average children per person", lambda: print(average_children(family_tree))],
    ["Exit to main menu", lambda: start_menu()],
]


def start_menu():
    """Outputs first menu options."""
    while True:
        print("\nOptions: ")
        for i, option in enumerate(start_options, 1):  # loop for each option in the enumarated array
            print(f"{i}. {option[0]}")  # prints the enumarated options(1 - number; and 2 - option)
        try:  # try this code
            command = int(input("\nEnter command: "))
            # the number entered cannot be greater than the maximum number of option counts, or not an integer
            if command > len(start_options) or not isinstance(command, int):
                print("Invalid command")
                continue
        except ValueError:  # if something is wrong in the previous code in 'try' - this code will be executed
            print("Invalid command")
            continue
        start_options[command - 1][1]()  # using the number entered to call the function from the array above


def use_options():
    """Outputs second menu options."""
    member_list = [member.name for member in family_tree.members]
    print("\nWelcome to Otto's and Cornelia's Family Tree, please select a person: ")
    print(*member_list, sep='\n')  # prints the separated array of member list
    name = input("\nEnter name: ")
    if family_tree.find_person_by_name(name):  # if a person exists
        while True:
            print(f"\nOptions: ")
            for i, option in enumerate(member_options):  # loop for each option in the enumarated array
                print(f"{i + 1}. {option[0]}")  # print the enumarated options(1 - number; and 2 - option)
            try:  # try this code
                command = int(input("\nEnter command: "))
                # the number entered cannot be greater than the maximum number of option counts, or not an
integer
                if command > len(member_options) or not isinstance(command, int):
                    print("Invalid command")
                    continue
            except ValueError:  # if something is wrong in the previous code in 'try' - this code will be executed
                print("Invalid command")
                continue
            member_options[int(command) - 1][1](name)  # using the number entered to call the function from the
array above
    else:
        print("Person not found.")
        start_menu()


def family_menu():
    """Outputs third menu options."""
    while True:
        print("\nOptions: ")
        for i, option in enumerate(family_options):  # loop for each option in the enumarated array
            print(f"{i + 1}. {option[0]}")  # print the enumarated options(1 - number; and 2 - option)
        try:  # try this code
            command = input("\nEnter command: ")
            # the number entered cannot be greater than the maximum number of option counts
            if int(command) > len(family_options):
                print("Invalid command")
                continue
        except ValueError:  # if something is wrong in the previous code in 'try' - this code will be executed
            print("Invalid command")
            continue
        family_options[int(command) - 1][1]()  # using the number entered to call the function from the array
above

family_tree.find_person_by_name("Otto Emmersohn")
start_menu()  # calling the start menu function
```

File ’list.py’:

```
# this .py file includes functions for adding people to the list
# .py icludes functions for saving the list to a csv file and reading from a csv file.


from person import Person
from FamilyTree import FamilyTree
import csv


def save_csv(family_tree):
    """Saves the family tree to a CSV file."""
    with open("family_tree.csv", "w") as file:
        writer = csv.writer(file, lineterminator="\n")  # lineterminator eliminates the empty line
        writer.writerow(["Name", "Date of Birth", "Gender", "Death Day"])  # writing rows to the csv file
        for member in family_tree.members:
            birth_day, birth_month, birth_year = member.get_birthday()  # converting into variables
            if not member.is_alive():
                death_day, death_month, death_year = member.get_death_day()
                death = f"{death_day}/{death_month}/{death_year}"
            else:
                death = "Alive"
            writer.writerow([
                member.name,
                f"{birth_day}/{birth_month}/{birth_year}",
                member._gender,
                death
            ])


def read_csv(path = "family_tree.csv"):
    """Reads a CSV file and adds the people to the family tree."""
    family_tree = FamilyTree()
    with open("family_tree.csv", "r") as file:
        reader = csv.reader(file)
        next(reader)  # Skip the header row.
        for row in reader:
            name, date_of_birth, gender, death_day = row
            date_of_birth = int(date_of_birth.replace("/", ""))  # replacing '/' with an empty string
            try:
                death_day = int(death_day.replace("/", ""))  # replacing '/' with an empty string
            except:
                death_day = -1
            # adding the person to the family tree
            family_tree.add_member(Person(name, date_of_birth, gender, death_day))
    return family_tree



# Adds a person to the family tree using the function 'add_member' from the FamilyTree class.

# family_tree = FamilyTree()

# family_tree.add_member(Person("Cornelia Emmersohn", 24_07_1982, "Female", -1))
# family_tree.add_member(Person("Otto Emmersohn", 19_05_1980, "Male", -1))
#
# family_tree.add_member(Person("Ara Song", 2_02_1948, "Female", 3_02_1972))
# family_tree.add_member(Person("John Winchester", 11_01_1947, "Male", 27_11_1995))
# family_tree.add_member(Person("Amy Wong", 12_11_1932, "Female", -1))
# family_tree.add_member(Person("Bo Chen", 23_08_1932, "Male", 9_11_2001))
# family_tree.add_member(Person("Andra Kaur", 30_10_1934, "Female", 23_05_2012))
# family_tree.add_member(Person("Kanan Khan", 23_01_1923, "Male", 30_05_2022))
#
# family_tree.add_member(Person("Emma Desposito", 5_07_1900, "Female", 17_03_2000))
# family_tree.add_member(Person("Nico Romano", 22_05_1897, "Male", 27_04_1964))
#
# family_tree.add_member(Person("Anna Romano", 12_01_1930, "Female", 6_02_1998))
# family_tree.add_member(Person("Deniele Bruno", 11_08_1928, "Male", 25_10_2024))
# family_tree.add_member(Person("Ada Hoffman", 28_02_1931, "Female", 4_03_1987))
# family_tree.add_member(Person("Gunther Emmersohn", 14_03_1928, "Male", 28_05_1957))
#
# family_tree.add_member(Person("Sam Song", 12_06_1969, "Male", -1))
# family_tree.add_member(Person("Emma Chen", 14_04_1970, "Female", -1))
# family_tree.add_member(Person("David Martinez", 23_05_1957, "Male", 12_07_1980))
# family_tree.add_member(Person("Lucy Chen", 14_04_1959, "Female", -1))
# family_tree.add_member(Person("Josh Khan", 26_08_1957, "Male", -1))
# family_tree.add_member(Person("Isabella Bruno", 19_03_1950, "Female", -1))
# family_tree.add_member(Person("Bernard Emmersohn", 31_10_1948, "Male", 16_04_2002))
# family_tree.add_member(Person("Zakk Emmersohn", 31_10_1956, "Male", -1))
#
# family_tree.add_member(Person("Chan Soun", 11_04_1988, "Male", -1))
# family_tree.add_member(Person("An Chen", 24_12_1991, "Female", -1))
# family_tree.add_member(Person("Aki Chen", 21_07_1995, "Male", 22_07_1995))
# family_tree.add_member(Person("Guozhi Chen", 16_08_2009, "Male", 19_03_2019))
# family_tree.add_member(Person("Harold Stokes", 18_03_1982, "Male", -1))
# family_tree.add_member(Person("Lina Chen", 18_11_1979, "Female", -1))
# family_tree.add_member(Person("Adeline Chen", 18_11_1979, "Female", 1_01_2014))
# family_tree.add_member(Person("Karl Emmersohn", 19_05_1985, "Male", -1))
#
# family_tree.add_member(Person("Jessica Vegas", 21_02_2000, "Female", -1))
# family_tree.add_member(Person("Shelby Emmersohn", 23_09_2001, "Male", -1))
# family_tree.add_member(Person("Melina Emmersohn", 7_03_2008, "Female", -1))

family_tree = read_csv()  # reads the csv file and adds the people from there to the family_tree variable
# print(family_tree.members)  # to make sure it works

# Adds a spouse and parent-child relationship between people using the functions from the FamilyTree class.

family_tree.add_spouse_relationship("Ara Song", "John Winchester")
family_tree.add_parent_relationship(["Sam Song"], "Ara Song", "John Winchester")

family_tree.add_spouse_relationship("Amy Wong", "Bo Chen")
family_tree.add_parent_relationship(["Emma Chen", "Lucy Chen"], "Amy Wong", "Bo Chen")

family_tree.add_spouse_relationship("Andra Kaur", "Kanan Khan")
family_tree.add_parent_relationship(["Josh Khan"], "Andra Kaur", "Kanan Khan")

family_tree.add_spouse_relationship("Emma Desposito", "Nico Romano")
family_tree.add_parent_relationship(["Anna Romano"], "Emma Desposito", "Nico Romano")

family_tree.add_spouse_relationship("Anna Romano", "Deniele Bruno")
family_tree.add_parent_relationship(["Isabella Bruno"], "Anna Romano", "Deniele Bruno")

family_tree.add_spouse_relationship("Ada Hoffman", "Gunther Emmersohn")
family_tree.add_parent_relationship(["Bernard Emmersohn", "Zakk Emmersohn"], "Ada Hoffman", "Gunther Emmersohn")


family_tree.add_spouse_relationship("Sam Song", "Emma Chen")
family_tree.add_parent_relationship(["An Chen", "Aki Chen", "Guozhi Chen"], "Sam Song", "Emma Chen")

family_tree.add_spouse_relationship("David Martinez", "Lucy Chen")
family_tree.add_spouse_relationship("Josh Khan", "Lucy Chen")
family_tree.add_parent_relationship(["Lina Chen", "Adeline Chen", "Cornelia Emmersohn"], "Josh Khan", "Lucy Chen")

family_tree.add_spouse_relationship("Isabella Bruno", "Bernard Emmersohn")
family_tree.add_parent_relationship(["Otto Emmersohn", "Karl Emmersohn"], "Isabella Bruno", "Bernard Emmersohn")
family_tree.add_spouse_relationship("Chan Soun", "An Chen")

family_tree.add_spouse_relationship("Harold Stokes", "Lina Chen")

family_tree.add_spouse_relationship("Cornelia Emmersohn", "Otto Emmersohn")
family_tree.add_parent_relationship(["Shelby Emmersohn", "Melina Emmersohn"], "Cornelia Emmersohn", "Otto Emmersohn")

family_tree.add_spouse_relationship("Jessica Vegas", "Shelby Emmersohn")


# save_csv(family_tree)
```

File ‘person.py’:

```
# this file includes the Person class and the methods that belong to it


from relationship import ChildRelationship, ParentRelationship, SpouseRelationship


# class Person is used to initialize a person's details
# Encapsulation implemented
class Person:
    def __init__(self, name: str, birthday: int, gender: str, death_day: int):  # initialize the person's attributes
        self._name = name
        self._birthday = birthday
        self._gender = gender
        self._death_day = death_day

        self._relationship = []

    def __str__(self):  # returns a string representation of a person
        return self._name

    def __repr__(self):  # helps to represent the relationship and understand the object
        return self._name

    # appends the relationship to the list
    def add_relationship(self, relationship):
        self._relationship.append(relationship)

    # decorator for the name, using for accessing the name of the person without calling the method
    @property
    def name(self):
        return self._name

    def get_relationships(self):
        """Return relationships for each person in the list."""
        return self._relationship

    def find_parents(self):
        """Return parents of the selected person."""
        parents = []
        for relationship in self._relationship:
            if isinstance(relationship, ChildRelationship):  # if the relationship is a child relationship
                parents.append(relationship.person2)
        return parents

    def find_siblings(self):
        """Return siblings of the selected person.
        finds siblings through parents' children"""
        siblings = []
        parents = self.find_parents()  # variable for parents, using 'find_parents' function
        for parent in parents:
            for relationship in parent.get_relationships():  # loop for relationships in parents, using
'get_relationships' function
                if not isinstance(relationship, ParentRelationship):  # if the relationship is not a parent relationship
                    continue
                if relationship.person2 != self and relationship.person2 not in siblings:
                    siblings.append(relationship.person2)
        return siblings

    def find_children(self):
        """Return children of the selected person."""
        children = []
        for relationship in self._relationship:
            if isinstance(relationship, ParentRelationship):  # if the relationship is a parent relationship
                children.append(relationship.person2)
        return children

    def find_cousins(self):
        """Return cousins of the selected person.
        Loop, that goes through parents, then siblings of parents and find their children"""
        cousins = []
        parents = self.find_parents()
        for parent in parents:
            siblings = parent.find_siblings()
            for sibling in siblings:
                children = sibling.find_children()
                for child in children:
                    cousins.append(child)
        return cousins

    def find_spouse(self):
        """Return spouses of the selected person."""
        spouse = []  # adds spouse to the array
        for relationship in self._relationship:
            if isinstance(relationship, SpouseRelationship):  # if the relationship is a spouse relationship
                spouse.append(relationship.person2)
        return spouse

    def get_birthday(self):
        """Return birthday of the selected person."""
        birth_day, birth_month, birth_year = self.get_date(self._birthday)
        return birth_day, birth_month, birth_year

    def get_death_day(self):
        """Return death day of the selected person if person is no longer alive."""
        if self._death_day == -1:
            return "Person is still alive."
        else:
            death_day, death_month, death_year = self.get_date(self._death_day)
            return death_day, death_month, death_year

    # static method: a method that belongs to a class, but not associated with a class object
    @staticmethod
    def get_date(date):
        """Converts a date to integers"""
        day = int(str(date)[-8:-6])
        month = int(str(date)[-6:-4])
        year = int(str(date)[-4:])
        return day, month, year

    def find_grandchildren(self):
        """Return a list of grandchildren (Person objects) of the current person.
        Loop, that goes through children of parents and find their children."""
        grandchildren = []
        for child in self.find_children():
            grandchildren.extend(child.find_children())
        return grandchildren

    def find_grandparents(self):
        """Adding grandparents to the list
        using loop that goes through parents of parents"""
        grandparents = []
        for parent in self.find_parents():
            grandparents.extend(parent.find_parents())
        return grandparents
```

```
def immediate_family_members(self):
    """Saving people in immediate family members: parents, spouse, children and siblings."""
    parents = self.find_parents()
    spouse = self.find_spouse()
    children = self.find_children()
    siblings = self.find_siblings()
    return parents, spouse, children, siblings

def number_of_children(self):
    """Return the number of children of the selected person."""
    return len(self.find_children())

def is_alive(self):
    """Check if selected person is alive"""
    if self._death_day == -1:  # if instead of the date of death there is '-1' in the list, the person is alive
        return True
    else:
        return False

def extended_family_members(self):
    """Return a list of extended family members of selected person."""
    extended_family = []  # list for all extended family members
    relations = [
        self.find_parents(),
        self.find_spouse(),
        self.find_children(),
        self.find_siblings(),
        [sibling for parent in self.find_parents() for sibling in parent.find_siblings()],  # adding siblings of parents
        self.find_cousins()
    ]

    for relation_list in relations:
        for relation in relation_list:
            if relation.is_alive():
                extended_family.append(relation)  # adds member to extended family
    return extended_family

#static method returns current day, month, and year to be able to calculate average age in other functions
@staticmethod
def current_datetime():
    """Using datetime module to get current date and convert it to int"""
    import datetime
    date_time = datetime.datetime.now()
    current_day = int(str(date_time.day))
    current_month = int(str(date_time.month))
    current_year = int(str(date_time.year))
```

```
        return current_day, current_month, current_year

    def get_age(self):
        """Return age of the selected person.
        Using current date/date of death and date of birth to calculate age of person."""
        current_day, current_month, current_year = self.current_datetime()  # converting into variables
        birth_day, birth_month, birth_year = self.get_birthday()  # converting into variables
        if self.is_alive():  # if the person is alive - the current date is subtracted by the date of birth.
            current_age_years = current_year - birth_year
            if (birth_month > current_month) or (birth_month == current_month and birth_day > current_day):
                current_age_years -= 1
        else:  # if the person is dead - the date of death is subtracted by the date of birth
            death_day, death_month, death_year = self.get_death_day()
            current_age_years = death_year - birth_year
            if (birth_month > death_month) or (birth_month == death_month and birth_day > death_day):
                current_age_years -= 1
        return current_age_years
```

File ‘FamilyTree.py’:

```
# this .py file contains a FamilyTree class that is responsible for all members of the family tree
# includes a list of members and methods for adding members and relationships
# contains method for finding a branch


from relationship import ParentRelationship, ChildRelationship, SpouseRelationship


class FamilyTree:
    members = []  # list of all members of the family tree

    def add_member(self, member):
        """Adds a member to the family tree."""
        self.members.append(member)

    def add_relationship(self, relationship_type, person1_name, person2_name):
        """Adds a relationship between two people."""
        person1 = self.find_person_by_name(person1_name)
        person2 = self.find_person_by_name(person2_name)
        if person1 is None or person2 is None:
            print(f"{person1_name} or {person2_name} do not exist")
            return

        relationship = relationship_type(person1, person2)
        person1.add_relationship(relationship)
```

```
def add_spouse_relationship(self, person1_name, person2_name):
    """Adds a spouse relationship between two people."""
    self.add_relationship(SpouseRelationship, person1_name, person2_name)
    self.add_relationship(SpouseRelationship, person2_name, person1_name)

def find_person_by_name(self, name):
    """Finds a person by name."""
    for person in self.members:
        if person.name == name:
            return person
    return None

def add_parent_relationship(self, children, parent1, parent2):
    """Adds parent-child relationships between children and two parents."""
    for child in children:
        self.add_relationship(ParentRelationship, parent1, child)
        self.add_relationship(ParentRelationship, parent2, child)
        self.add_relationship(ChildRelationship, child, parent1)
        self.add_relationship(ChildRelationship, child, parent2)

def get_branch(self, person, visited=None):
    """Loop that goes through whole family branch to add people to the list.
    Using “visited” prevents an endless loop of adding people to this list,
    and if a person is already on the list when you view it again, they will not be added."""
    first_person = False
    if visited is None:
        visited = []
        first_person = True
    if person in visited:
        return []
    visited.append(person)
    members = [person]
    for parent in person.find_parents():
        members.extend(self.get_branch(parent, visited))
    if not first_person:
        for child in person.find_children():
            members.extend(self.get_branch(child, visited))
        for spouse in person.find_spouse():
            members.extend(self.get_branch(spouse, visited))
    else:
        for child in person.find_children():
            members.append(child)
    return members

def get_birthday_calendar(self, members):
    """Returns sorted birthday calendar."""
    birthday_calendar = {}
```

```
        for member in members:
            day, month, year = member.get_birthday()
            birthday = (day, month)
            if birthday in birthday_calendar:
                birthday_calendar[birthday].append(member)
            else:
                birthday_calendar[birthday] = [member]
        birthday_calendar = dict(sorted(birthday_calendar.items(), key=lambda x: (x[0][1], x[0][0])))
        return birthday_calendar

    def average_age_per_person(self):
        """Calculates the average age per person."""
        total_members = len(self.members)
        total_age = sum(person.get_age() for person in self.members)  # adding all ages
        average_age = total_age // total_members
        return average_age

    def average_age_at_death(self):
        """Calculates the average age at death."""
        dead_members = [person for person in self.members if person.is_alive()]  # a loop for adding people if
alive
        total_age = sum(person.get_age() for person in dead_members)  # finding sum of ages of dead members
        average_age = total_age // len(dead_members)
        return average_age

    def total_children_count(self):
        """Calculate the total number of children."""
        total_children = sum(len(person.find_children()) for person in self.members)
        return total_children

    def average_children_per_person(self):
        """Calculate the average number of children per person."""
        total_members = len(self.members)
        if total_members == 0:
            return 0
        total_children = self.total_children_count()
        # Use 'total_children_count' to get the total number of children
        average_children = total_children // total_members
        # Calculate the average
        return average_children
```

File ‘relationship.py’:

```
# this file contains the Relationship class as a parent class and three subclasses
# designed to represent connections between people
```

```
from abc import ABC, abstractmethod


class Relationship(ABC):
    """Parent class. Represents a relationship between people."""
    def __init__(self, person1, person2):  # initializes two people for a relationship
        self.person1 = person1
        self.person2 = person2

    # The decorator that exposes only essential information and functionalities, hiding complex details
    @abstractmethod
    def __str__(self):  # returns a string representation of the relationship
        return f"{self.person1} in relationship with {self.person2}"

    # the decorator that exposes only essential information and functionalities, hiding complex details
    @abstractmethod
    def __repr__(self):  # helps to represent the relationship and understand the object
        return f"{self.person1} in relationship with {self.person2}"


class SpouseRelationship(Relationship):
    """Child class of 'relationship'. Represents a spouse relationship between two people."""
    def __init__(self, person1, person2):
        super().__init__(person1, person2)  # super() is used to use the parent constructor

    def __str__(self):  # returns a string representation of the relationship
        return f"{self.person1} is in marriage with {self.person2}"

    def __repr__(self):  # helps to represent the relationship and understand the object
        return f"{self.person1} is in marriage with {self.person2}"


class ParentRelationship(Relationship):
    """Child class of 'relationship'. Represents a parent relationship between people."""
    def __init__(self, person1, person2):
        super().__init__(person1, person2)  # super() is used to use the parent constructor

    def __str__(self):  # returns a string representation of the relationship
        return f"{self.person1} is a parent of {self.person2}"

    def __repr__(self):  # helps to represent the relationship and understand the object
        return f"{self.person1} is a parent of {self.person2}"


class ChildRelationship(Relationship):# child class for child relationship
    def __init__(self, person1, person2):
        super().__init__(person1, person2)  # super() is used to use the parent constructor

    def __str__(self):  # returns a string representation of the relationship
        return f"{self.person1} is a child of {self.person2}"

    def __repr__(self):  # helps to represent the relationship and understand the object
        return f"{self.person1} is a child of {self.person2}"
```