FIT1045 Algorithms and programming in Python, OCT-2021
Programming Assignment

Assessment value: 22% (10% for Part 1 + 12% for Part 2)

Due: Week 6 (Part 1), Week 11 (Part 2)

This assignment is designed to practice and assess your capabilities to **solve complex algorithmic problems with Python** AND your ability to **verbally explain your solution**. **It is an individual assignment,** meaning that you are supposed to **solve it alone** during your self-study time, and are **not allowed to share or copy your solution or any part of it**.

Please ensure that you have read and understood the university's policies on plagiarism and collusion available at `http://www.monash.edu.au/students/policies/academic-integrity.html`. You will be required to agree to these policies when you submit your assignment. This implies that you are not allowed to post any of your code to any forum (including Ed).

However, please *do* use Ed to clarify task requirements, and to discuss other assignment-related questions that do not disclose parts of your solution. If you feel stuck and need more specific help, please approach your tutors. They won't solve the problems for you (and especially won't debug your code), but can provide you with general feedback and tell you if you are on the right track. While parts of the assignment might be challenging, we hope that completing it will be a fruitful and engaging learning experience for you.

Marks, Submission, and Interviews

To obtain marks for this assignment you have to submit two versions of a module file `familytree.py` (based on provided template) via Moodle, at the due dates for Part 1 and Part 2 respectively, AND attend the corresponding assignment interviews for each part. **Not attending the interview will result in 0 marks for the corresponding part of the assignment.**

Your module is not allowed to use imports except for the module `copy`.

Submission procedure

For each of your Part 1 and Part 2 submissions, submit only the module file `familytree.py` to Moodle, which should contain your solutions for that part. The file you submit should be based on the template file provided. Do not enter your name or student ID into the module file, and do not rename the file — upload it to Moodle with the same name as the template, `familytree.py`.

Interviews

Your assignment interviews will be held during your lab class in the week following your assignment submission, i.e. Week 7 for Part 1, and Week 12 for Part 2. During the interviews you must be able to explain your solution (otherwise any or all marks can be deducted). It is highly recommended that you keep your submission in a clean form with well decomposed and documented functions, and only use programming constructs that you can explain. If you are unable to explain your solution, the maximum number of marks that you can achieve for the part of the assignment is capped to:

- 0 marks if you don't attend the interview or don't participate in it in a meaningful way
- 25% of the maximally attainable marks if you are unable to explain the Python language elements you use in your code (e.g., you use a range but do not know what it is)
- 49% of the maximally attainable marks if you are unable to explain the purpose of parts of your submission (e.g., you cannot explain what an input to a function means or what output is computed for it).
- 69% of the maximally attainable marks if you are unable to explain how your solution works (e.g., you are unable to verbalise the overall algorithmic strategy that is used in a part of your solution).

Family Trees: Overview

In this assignment we create a Python module to implement some basic family tree software, where users can look up various relationships that exist in the database. We represent each entry in a family tree database as a list of three strings `[name, father, mother]`, where `name` is a person's name, `father` is the name of their father, and `mother` is the name of their mother. Where a particular relationship is unknown, the value `None` is used.

For example, consider the following family tree:

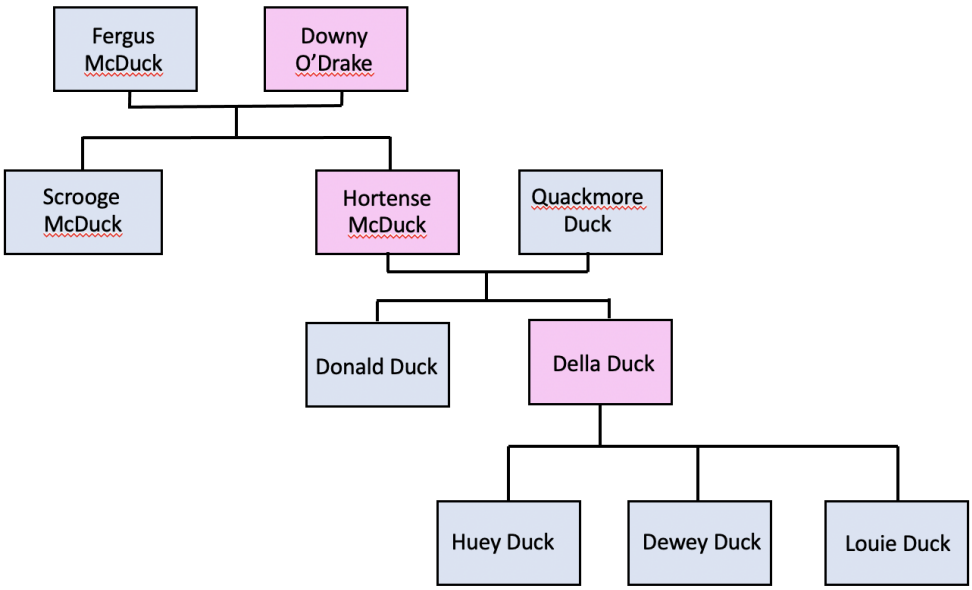

Figure 1: The Duck Family Tree

The relationships shown in this tree are represented using the following database in Python (note that this database is just for the purposes of illustration, and your module must work for other family tree databases as well):

```
>>> duck_tree = [['Fergus McDuck', None, None],
...              ['Downy ODrake', None, None],
...              ['Quackmore Duck', None, None],
...              ['Donald Duck','Quackmore Duck','Hortense McDuck'],
...              ['Della Duck', 'Quackmore Duck', 'Hortense McDuck'],
...              ['Hortense McDuck', 'Fergus McDuck', 'Downy ODrake'],
...              ['Scrooge McDuck', 'Fergus McDuck', 'Downy ODrake'],
...              ['Huey Duck', None, 'Della Duck'],
...              ['Dewey Duck', None, 'Della Duck'],
...              ['Louie Duck', None, 'Della Duck']]
```

Again, this is just one limited example of a family tree database. A larger database, `hobbit-family.txt`, has also been provided for you separately to read in and test your functions on [1]. As run-through examples, this assignment will use both the `duck_tree` database and the `hobbit-family` database. But it is important to remember that **your functions should work on any family tree that is represented in the manner described**. Consider creating your own examples to test your functions on.

[1] The `hobbit-family.txt` test file has been compiled using the family tree information at `http://lotrproject.com/hobbits.php`. All character names used in the file are by J.R.R. Tolkein.

Part 1: Simple Relationship Lookups (10%, due in Week 6)

The first part of this assignment involves writing functions that retrieve information about simple familial relationships in a family tree database. Each function in this section takes as input a person's name and a family tree database, and returns some information about that person which is stored in the database.

Task A: Read Family (1 mark)

Implement a function `read_family(filename)` with the following specification.

Input: A filename `filename` containing a family tree database stored as a text file, where each line is in the form: `name, father, mother`

Output: A family tree database containing the contents of the file stored in the required format.

For example, `read_family` should behave as follows with the provided test database, `hobbit-family.txt`:

```
>>> hobbits = read_family('hobbit-family.txt')
>>> len(hobbits)
119
>>> hobbits[118]
['Sancho Proudfoot', 'Olo Proudfoot', None]
```

Task B: Person Index, Father, Mother (1.5 marks)

For this task, your module has to contain at least the three mandatory functions `person_index`, `father`, and `mother`. The details are as follows:

Implement a function `person_index(person, family)` with the following specification.

Input: A string containing a person's name, `person`, and a family tree database `family` as specified above.

Output: The index value of that person's entry in the family tree database, or `None` if they do not have an entry in the tree.

For example, for the `duck_tree` database defined on page 2, `person_index` behaves as follows:

```
>>> person_index('Dewey Duck', duck_tree)
8
>>> person_index('Daffy Duck', duck_tree)
```

Implement a function `father(person, family)` with the following specification.

Input: A string containing a person's name, `person`, and a family tree database `family` as specified above.

Output: A string containing the name of that person's father, as defined in the database `family`, or `None` if the information is not in the database (including if the person themselves are not in the database).

Building on the same example, `father` would behave as follows:

```
 >>> father('Della Duck', duck_tree)
'Quackmore Duck'
>>> father('Huey Duck', duck_tree)

>>> father('Daffy Duck', duck_tree)
```

Implement a function `mother(person, family)` with the following specification.

Input: A string containing a person's name, `person`, and a family tree database `family` as specified above.

Output: A string containing the name of that person's mother, as defined in the database `family`, or `None` if the information is not in the database (including if the person themselves are not in the database).

For example, `mother` should behave as follows:

```
>>> mother('Hortense McDuck', duck_tree)
'Downy ODrake'
>>> mother('Fergus McDuck', duck_tree)

>>> mother('Daffy Duck', duck_tree)
```

Task C: Children (1.5 marks)

Implement a function `children(person, family)` with the following specification.

Input: A string containing a person's name, `person`, and a family tree database `family` as specified above.

Output: A list containing the names of all of `person`'s children that are stored in the database. If there is no information about any of `person`'s children in the database (including if the person themselves is not in the database), the function should return an empty list.

For example, `children` should behave as follows:

```
>>> duck_tree = [['Fergus McDuck', None, None],
...              ['Downy ODrake', None, None],
...              ['Quackmore Duck', None, None],
...              ['Donald Duck','Quackmore Duck','Hortense McDuck'],
...              ['Della Duck', 'Quackmore Duck', 'Hortense McDuck'],
...              ['Hortense McDuck', 'Fergus McDuck', 'Downy ODrake'],
...              ['Scrooge McDuck', 'Fergus McDuck', 'Downy ODrake'],
...              ['Huey Duck', None, 'Della Duck'],
...              ['Dewey Duck', None, 'Della Duck'],
...              ['Louie Duck', None, 'Della Duck']]
>>> sorted(children('Della Duck', duck_tree))
['Dewey Duck', 'Huey Duck', 'Louie Duck']
>>> children('Donald Duck', duck_tree)
[]
>>> sorted(children('Fergus McDuck', duck_tree))
['Hortense McDuck', 'Scrooge McDuck']
>>> children('Donald Mallard', duck_tree)
[]
```

Task D: Grandchildren (2 marks)

Implement a function `grandchildren(person, family)` with the following specification.

Input: A string containing a person's name, `person`, and a family tree database `family` as specified above.

Output: A list containing only the names of the grandchildren of `person` that are stored in the database, or an empty list if no such information exists in the database (including if the person themselves is not in the database).

For example, `grandchildren` should behave as follows:

```
>>> duck_tree = [['Fergus McDuck', 'Dingus McDuck', 'Molly Mallard'],
...              ['Downy ODrake', None, None],
...              ['Quackmore Duck', None, None],
...              ['Donald Duck','Quackmore Duck','Hortense McDuck'],
...              ['Della Duck', 'Quackmore Duck', 'Hortense McDuck'],
...              ['Hortense McDuck', 'Fergus McDuck', 'Downy ODrake'],
...              ['Scrooge McDuck', 'Fergus McDuck', 'Downy ODrake'],
...              ['Huey Duck', None, 'Della Duck'],
...              ['Dewey Duck', None, 'Della Duck'],
...              ['Louie Duck', None, 'Della Duck']]
>>> sorted(grandchildren('Quackmore Duck', duck_tree))
['Dewey Duck', 'Huey Duck', 'Louie Duck']
>>> sorted(grandchildren('Downy ODrake', duck_tree))
['Della Duck', 'Donald Duck']
>>> grandchildren('Della Duck', duck_tree)
[]
```

Task E: Cousins (4 marks)

Implement a function `cousins(person, family)` with the following specification.

Input: A string containing a person's name, `person`, and a family tree database `family` as specified above.

Output: A list containing the names of all cousins of `person` that are stored in the database, or an empty list if no such information exists in the database (including if the person themselves is not in the database).

For example, `cousins` should behave as follows, after reading in the provided `hobbit-family.txt` file:

```
>>> hobbits = read_family('hobbit-family.txt')
>>> sorted(cousins('Frodo Baggins', hobbits))
['Daisy Baggins', 'Merimac Brandybuck', 'Milo Burrows', 'Saradoc Brandybuck', 'Seredic Brandybuck']
```

Part 2: Complex Relationship Lookups (12%, due in Week 11)

In this part of the assignment, you will write two functions for returning more complex relationship information from a family tree database.

Task A: Direct Ancestor (6 Marks)

Implement a function `direct_ancestor(p1, p2, family)` with the following specification.

Input: Two strings representing two names, `p1` and `p2`, and a family tree database `family`.

Output: One of the following three string outputs as appropriate (where `p1` and `p2` are the given input strings, and `n` is a string representation of a non-negative integer):

- "`p1` is a direct ancestor of `p2`, `n` generations apart."
- "`p2` is a direct ancestor of `p1`, `n` generations apart."
- "`p1` is not a direct ancestor or descendant of `p2`."

For example, `direct_ancestor(p1, p2, family)` should behave as follows:

```
>>> hobbits = read_family('hobbit-family.txt')
>>> direct_ancestor('Frodo Baggins', 'Frodo Baggins', hobbits)
'Frodo Baggins is a direct ancestor of Frodo Baggins, 0 generations apart.'
>>> direct_ancestor('Frodo Baggins', 'Gormadoc Brandybuck', hobbits)
'Gormadoc Brandybuck is a direct ancestor of Frodo Baggins, 5 generations apart.'
```

Task B: Cousin Degree (6 Marks)

Implement a function `cousin_degree(p1, p2, family)` with the following specification.

Input: Two strings representing two names, `p1` and `p2`, and a family tree database `family`.

Output: An integer representing the minimum distance cousin relationship between `p1` and `p2`, as follows:

- 0, if `p1` and `p2` are siblings;
- 1, if `p1` and `p2` are cousins;
- some positive integer `n`, if `p1` and `p2` are `n`th cousins[2]; or
- -1 if `p1` and `p2` have no cousin or sibling relationship.

For example, `cousin_degree(p1, p2, family)` should behave as follows:

```
>>> hobbits = read_family("hobbit-family.txt")
>>> cousin_degree('Minto Burrows', 'Myrtle Burrows', hobbits)
0
>>> cousin_degree('Estella Bolger', 'Meriadoc Brandybuck', hobbits)
3
>>> cousin_degree('Frodo Baggins', 'Bilbo Baggins', hobbits)
-1
```

[2]You should use the definition of `n`th cousin given at `https://www.familysearch.org/blog/en/cousin-chart/`.