US Social Security data on given names

Eric Martin, CSE, UNSW

COMP9021 Principles of Programming

[1]:
```
from pathlib import Path
import os
import csv
from collections import defaultdict
```

Downloaded from https://www.ssa.gov/OACT/babynames/limits.html, the `names` directory contains, besides `NationalReadMe.pdf`, files whose names are of the form `yob****.txt` with “yob” standing for “year of birth” and `****` ranging from `1880` to `2018`. These are csv files, with “csv” standing for “comma separated values”: each line consists of 3 fields: a first name, `F` or `M` for female or male, respectively, and a strictly positive integer for the count of newborns who have been given that name in the year whose value is embedded in the file name. All female names are listed before all male names. For a given gender, data are listed in decreasing order of count. For a given gender and count, names are listed in lexicographic order. For instance, for the oldest year, here are the first 10 lines:

[2]:
```
!head names/yob1880.txt
```

```
Mary,F,7065
Anna,F,2604
Emma,F,2003
Elizabeth,F,1939
Minnie,F,1746
Margaret,F,1578
Ida,F,1472
Alice,F,1414
Bertha,F,1320
Sarah,F,1288
```

And here are the last 10 lines:

[3]:
```
!tail names/yob1880.txt
```

```
Unknown,M,5
Vann,M,5
Wes,M,5
Winston,M,5
Wood,M,5
Woodie,M,5
Worthy,M,5
```

```
Wright,M,5
York,M,5
Zachariah,M,5
```

Our first task is to reorganise the data: create a directory `names_per_gender`, create two subdirectories, `female` and `male`, of `names_per_gender`, and in each of both subdirectories and for each `.txt` file F in `names`, create a copy of F such that:

- the copy of F in the `female` subdirectory will consist of the lines for all female names in F with only 2 fields, namely, first name and count, so without `F`, the second field;
- the copy of F in the `male` subdirectory will consist of the lines for all male names in F with only 2 fields, namely, first name and count, so without `M`, the second field.

To work with directories and files in a platform independent manner, the `Path` class from the `pathlib` module is appropriate. One can create `Path` objects from directory and file names and check whether they exist with `Path`'s `exists()` method. Given a `Path` object P for a directory D, `Path` objects for subdirectories of D or for files in D can be created with the `/` operator, with as first and second operands, P and the subdirectory or file name, respectively; `/` will produce path names with a separator that is appropriate for the operating system on which code is executed:

[4]:
```
Path('names'), Path('names').exists()
Path('names') / 'yob1880.txt', (Path('names') / 'yob1880.txt').exists()
Path('nonexisting'), Path('nonexisting').exists()
Path('names') / 'yob1800.txt', (Path('names') / 'yob1800.txt').exists()
```

[4]:
```
(PosixPath('names'), True)
```

[4]:
```
(PosixPath('names/yob1880.txt'), True)
```

[4]:
```
(PosixPath('nonexisting'), False)
```

[4]:
```
(PosixPath('names/yob1800.txt'), False)
```

We first create a `Path` object for the existing `names` directory, for the to be created `names_per_gender` directory, and for the to be created `female` and `male` subdirectories of `names_per_gender`:

[5]:
```
names_dirname = Path('names')
names_per_gender_dirname = Path('names_per_gender')
female_subdirname = names_per_gender_dirname / 'female'
male_subdirname = names_per_gender_dirname / 'male'
```

The `exists()` function from the `path` module of the `os` module also allows one to check whether a directory or file exists. That module has other useful functions, in particular:

- `rmdir()`, to remove an empty directory;
- `mkdir()`, to create (make) a directory that does not already exist.

For instance, if the `names_per_gender` directory existed, contained `female` and `male` and no other subdirectories, and both `female` and `male` were empty directories, then the following code fragment would successfully

- remove the `female` directory,
- remove the `male` directory, and
- remove the then empty `names_per_gender` directory.

That would allow the next three calls to `os.mkdir()` to execute successfully, without a `FileExistsError` error to be raised:

```
[6]: if os.path.exists(names_per_gender_dirname):
         os.rmdir(female_subdirname)
         os.rmdir(male_subdirname)
         os.rmdir(names_per_gender_dirname)
     os.mkdir(names_per_gender_dirname)
     os.mkdir(female_subdirname)
     os.mkdir(male_subdirname)
```

We need to process all files in `names` except for `NationalReadMe.pdf`. We could use the `listdir()` function from the `os` module to list all files in `names` and ignore files not ending in `.txt`:

```
[7]: for file in os.listdir(names_dirname):
         if not file.endswith('.txt'):
             print(file)
```

```
NationalReadMe.pdf
```

Thanks to the `glob()` method of the `Path` class, we can instead generate only `Path` objects for the files of interest. This method uses Unix syntax to create patterns and match file and directory names:

- `*` to match a (possibly empty) sequence of characters
- `?` to mach a single character
- square brackets to enclose the characters to match.

The following statements illustrate:

```
[8]: list(names_dirname.glob('*17*'))
     list(names_dirname.glob('*2??7*'))
     list(names_dirname.glob('*2??[357]*'))
```

```
[8]: [PosixPath('names/yob2017.txt'), PosixPath('names/yob1917.txt')]

[8]: [PosixPath('names/yob2017.txt'), PosixPath('names/yob2007.txt')]

[8]: [PosixPath('names/yob2015.txt'),
      PosixPath('names/yob2017.txt'),
      PosixPath('names/yob2003.txt'),
      PosixPath('names/yob2007.txt'),
      PosixPath('names/yob2013.txt'),
      PosixPath('names/yob2005.txt')]
```

To extract the values of a csv file, one can of course open the file, read it line by line, and split each line using the comma as separator, but it is cleaner and more robust to instead, let the object returned by `open()` be the argument of the `reader()` function of the `csv` module; that function returns an iterator to generate for each line in the file, the tuple of values on that line. The following code fragment illustrates, printing out all lines in `yob1880.txt` for counts of female or male names greater than 2000:

[9]:
```
with open(names_dirname / 'yob1880.txt') as file:
    csv_file = csv.reader(file)
    for name, gender, tally in csv_file:
        if int(tally) > 2_000:
            print(name, gender, tally)
```

```
Mary F 7065
Anna F 2604
Emma F 2003
John M 9655
William M 9532
James M 5927
Charles M 5348
George M 5126
Frank M 3242
Joseph M 2632
Thomas M 2534
Henry M 2444
Robert M 2415
Edward M 2364
Harry M 2152
```

A file such as `yob1880.txt` is to be processed as one of the files in `names` whose paths are generated by `glob()` applied to `names_dirname`. Names and counts extracted from the rows in `yob1880.txt` are to be written to one of both files with the name `yob1880.txt` located in the `female` and `male` subdirectories of `names_per_gender`. Thanks to the `name` attribute of a `Path` object, the paths to both files are conveniently created from the path to `yob1880.txt` in `names`:

[10]:
```
filename = next(names_dirname.glob('*1880*'))

filename
filename.parent
filename.name
female_subdirname / filename.name
male_subdirname / filename.name
```

[10]:
```
PosixPath('names/yob1880.txt')
```

[10]:
```
PosixPath('names')
```

[10]:
```
'yob1880.txt'
```

[10]:
```
PosixPath('names_per_gender/female/yob1880.txt')
```

[10]:

```
PosixPath('names_per_gender/male/yob1880.txt')
```

For each `.txt` file F in `names`, we open, with a single `with` statement, F for reading purposes, and two files F_F and F_M with the same name as F in the subdirectories `female` and `male` of `names_per_gender`, respectively, for writing purposes, with the paths to F_F and F_M created as just described. In parallel to making use of `csv.reader()`, we make use of `csv.writer()` to write rows of data in a csv file, with successive values properly separated with commas. The dictionary `csv_file_per_gender` allows one to choose which one of F_F or F_M should be written to. In the last line of the following code fragment, the only purpose of the assignment to `_` is to suppress Jupyter output:

[11]:

```
for filename in names_dirname.glob('*.txt'):
    with open(filename) as file,\
          open(female_subdirname / filename.name, 'w') as female_file,\
          open(male_subdirname / filename.name, 'w') as male_file:
         csv_file = csv.reader(file)
         female_csv_file = csv.writer(female_file)
         male_csv_file = csv.writer(male_file)
         csv_file_per_gender = {'F': female_csv_file, 'M': male_csv_file}
         for name, gender, tally in csv_file:
             _ = csv_file_per_gender[gender].writerow((name, tally))
```

Our second task is to find out the longest intervals of time that separate the years Y_1 and Y_2 when a name was given (as a male or female name) in both Y_1 and Y_2, but not in-between. We would like to output the top 10 longest intervals together with the years that start and end the interval, and together with the name that was "forgotten and revived" in that time interval.

To this aim, it is convenient to create a dictionary that maps a given first name to the list of years, from oldest to most recent, when the name was given once at least. For instance, here are the years when `Franc` was given as a name:

[12]:

```
!grep Franc, names/*
```

```
names/yob1882.txt:Franc,F,5
names/yob1883.txt:Franc,F,5
names/yob2001.txt:Franc,M,5
names/yob2002.txt:Franc,M,6
names/yob2013.txt:Franc,M,5
```

So `'Franc'` should be one of the keys, with as value `[1882, 1883, 2001, 2002, 2013]`.

Years will be added one by one to the lists of values as files are processed one by one. Using a simple dictionary, one has to distinguish between creating a key and a value, that should be a list with a single year, and adding a new year to the list that is the value of an existing key:

[13]:

```
name = 'Franc'
years_per_name = {}
for year in 1882, 1883, 2001, 2002, 2003:
    if not name in years_per_name:
        years_per_name[name] = [year]
        print(f'Processing year {year}: '
              f'creating key "{name}" and value [{year}]'
             )
    else:
        years_per_name[name].append(year)
        print(f'Processing year {year}: '
              f'appending {year} to value for key "{name}"'
             )

years_per_name
```

```
Processing year 1882: creating key "Franc" and value [1882]
Processing year 1883: appending 1883 to value for key "Franc"
Processing year 2001: appending 2001 to value for key "Franc"
Processing year 2002: appending 2002 to value for key "Franc"
Processing year 2003: appending 2003 to value for key "Franc"
```

```
[13]: {'Franc': [1882, 1883, 2001, 2002, 2003]}
```

A `KeyError` error is generated when trying to access a nonexisting key:

```
[14]: name = 'Franc'
      years_per_name = {}
      years_per_name[name]
```

```
      ␣
↪---------------------------------------------------------------------------

        KeyError                                  Traceback (most recent call␣
↪last)

        <ipython-input-14-dce68830c349> in <module>
          1 name = 'Franc'
          2 years_per_name = {}
    ----> 3 years_per_name[name]


        KeyError: 'Franc'
```

When using a `defaultdict` from the `collections` module, trying to access a nonexisting key creates the key, together with the default value for the class provided as argument to `defaultdict`:

```
[15]: name = 'Franc'

      years_per_name = defaultdict(int)
```

```
print('Creating a key with 0 as default value:')
years_per_name[name]
years_per_name
print('Creating a key with 0 as default value, immediately modified:')
years_per_name = defaultdict(int)
years_per_name[name] += 2; years_per_name

years_per_name = defaultdict(list)
print('Creating a key with [] as default value:')
years_per_name[name]
years_per_name
print('Creating a key with [] as default value, immediately modified:')
years_per_name[name].append(1882); years_per_name
```

```
Creating a key with 0 as default value:
```

```
[15]: 0
```

```
[15]: defaultdict(int, {'Franc': 0})
```

```
Creating a key with 0 as default value, immediately modified:
```

```
[15]: defaultdict(int, {'Franc': 2})
```

```
Creating a key with [] as default value:
```

```
[15]: []
```

```
[15]: defaultdict(list, {'Franc': []})
```

```
Creating a key with [] as default value, immediately modified:
```

```
[15]: defaultdict(list, {'Franc': [1882]})
```

Thanks to default dictionaries, the key `'Franc'` can be created and years incrementally added to the value list as follows:

```
[16]: name = 'Franc'
      years_per_name = defaultdict(list)
      for year in 1882, 1883, 2001, 2002, 2003:
          years_per_name[name].append(year)

      years_per_name
```

```
[16]: defaultdict(list, {'Franc': [1882, 1883, 2001, 2002, 2003]})
```

Extracting years from filenames is easy:

```
[17]: int('yob1880.txt'[3 : 7])
```

[17]: 1880

So creating the full dictionary can be done as follows; we only have to beware that `glob()` does not return the file names in lexicographic order, so we use `sorted()` as it is essential that the years that make up the value of a given key of `years_per_name` are sorted from oldest to most recent:

[18]:
```
years_per_name = defaultdict(list)
for filename in sorted(names_dirname.glob('*.txt')):
    year = int(filename.name[3 : 7])
    with open(filename) as file:
        csv_file = csv.reader(file)
        for name, _, _ in csv_file:
            years_per_name[name].append(year)

years_per_name['Franc']
```

[18]: [1882, 1883, 2001, 2002, 2013]

From `years_per_name`, we can create a list of triples of the form (D, Y, N) where D is a year difference, Y is a year that starts a year difference of D (to which D can be added and yield the year that ends the year difference), and N is a name that was given in year Y and only D years later:

[19]:
```
revivals = [[years_per_name[name][i + 1] - years_per_name[name][i],
             years_per_name[name][i], name
            ] for name in years_per_name
                  for i in range(len(years_per_name[name]) - 1)
           ]

[revival for revival in revivals if revival[0] == 2001 - 1883]
```

[19]: [[118, 1883, 'Franc']]

Sorting `revivals` in reversed order results in a list where:

- year differences are ordered from largest to smallest;
- for a given year difference, years that start the year difference are ordered from most recent to oldest;
- for a given year difference and year that starts the year difference, names are ordered in anti-lexicographic order:

[20]:
```
revivals.sort(reverse=True)
for i in range(10):
    print(f'{revivals[i][2]} was last used in {revivals[i][1]} '
          f'and then again in {revivals[i][1] + revivals[i][0]}, '
          f'{revivals[i][0]} years later.'
         )
```

```
Franc was last used in 1883 and then again in 2001, 118 years later.
```

```
Izzie was last used in 1891 and then again in 2006, 115 years later.
Rasmus was last used in 1888 and then again in 2003, 115 years later.
Izma was last used in 1899 and then again in 2007, 108 years later.
Leannah was last used in 1889 and then again in 1996, 107 years later.
Almar was last used in 1915 and then again in 2017, 102 years later.
Addiemae was last used in 1915 and then again in 2017, 102 years later.
Saidee was last used in 1893 and then again in 1995, 102 years later.
Olwen was last used in 1917 and then again in 2018, 101 years later.
Onah was last used in 1916 and then again in 2017, 101 years later.
```

It could be better to have:

- year differences ordered from largest to smallest;
- for a given year difference, years that start the year difference ordered from oldest to most recent;
- for a given year difference and year that starts the year difference, names ordered in lexicographic order.

It suffices to negate the year differences to reverse their order:

[21]:
```python
revivals.sort(key=lambda x: (-x[0], x[1], x[2]))
for i in range(10):
    print(f'{revivals[i][2]} was last used in {revivals[i][1]} '
          f'and then again in {revivals[i][1] + revivals[i][0]},',
          f'{revivals[i][0]} years later.'
         )
```

```
Franc was last used in 1883 and then again in 2001, 118 years later.
Rasmus was last used in 1888 and then again in 2003, 115 years later.
Izzie was last used in 1891 and then again in 2006, 115 years later.
Izma was last used in 1899 and then again in 2007, 108 years later.
Leannah was last used in 1889 and then again in 1996, 107 years later.
Saidee was last used in 1893 and then again in 1995, 102 years later.
Addiemae was last used in 1915 and then again in 2017, 102 years later.
Almar was last used in 1915 and then again in 2017, 102 years later.
Caledonia was last used in 1900 and then again in 2001, 101 years later.
Tabea was last used in 1915 and then again in 2016, 101 years later.
```