Homework 11

Due Nov 16th, 2020

Fill in your name

In [1]:
```
first_name = "Scott"
last_name = "Urista"

assert(len(first_name) != 0)
assert(len(last_name)  != 0)
```

1) Sudoku

In a Sudoku puzzle, the player tries to fill a square with a number from 1 to 9. The number must not already appear in the same row, column, or 3x3 square.

You are given 3 lists, holding the values already seen in a row, column, and square. Produce a list of the legal remaining values.

Your solution should use one or more List Comprehensions for full credit.

In [2]:
```
def legal_values(row, col, square):
    full_moves = [1,2,3,4,5,6,7,8,9]


    # we can concantenate lists
    moves_list = row + col + square


    # return items not in the full list
    return [item for item in full_moves if item not in moves_list]
```

Unit tests for Sudoku

In [3]:
```
def test_sudoku():
    assert(legal_values([1], [4], [7]) == [2, 3, 5, 6, 8, 9])
    assert(legal_values([1, 2, 3], [4, 5, 6], [7, 8, 9]) == [])
    assert(legal_values([1, 2, 3], [1, 2, 3], [7, 8, 9]) == [4, 5, 6])
    assert(legal_values([1, 3, 5], [1, 4, 8], [7, 8, 9]) == [2, 6])
    assert(legal_values([1, 3, 5, 7, 9], [2, 4, 6, 8], [7, 9]) == [])
    assert(legal_values([1, 5, 7, 9], [2, 4, 8], [7, 9]) == [3, 6])
    print('Success!')

test_sudoku()
```

```
Success!
```

2) Graph Global Mile records

Take a look at the data here: you can harvest it from the web, or use a CSV file we'll provide.

https://github.com/KarenWest/FundamentalsOfDataAnalysisInLanguageR/blob/master/WorldRecords.csv
(https://github.com/KarenWest/FundamentalsOfDataAnalysisInLanguageR/blob/master/WorldRecords.csv)

We haven't discussed many of the points below: you will need to explore the documentation on your own.

https://matplotlib.org/3.2.1/contents.html (https://matplotlib.org/3.2.1/contents.html)

The CSV file has many records. Plot the world records for the mile. Let X be the year and Y be the time in seconds.

Map the Mens and Womens records on the same graph in different colors.

Since records for men and women were set in different years, you won't be able to use plot(). Use a scatter plot instead.

Make sure your X and Y values are numbers, rather than the strings in the table.
Include labels for the X and Y axis, and a legend telling us what the colors mean.

Include a horizontal line at y = 240: the 4 minute barrier has been a touchstone and a benchmark for years.

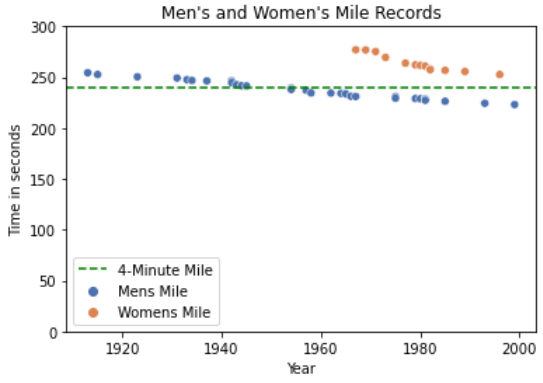
You will want the Y axis to include 0 so that we can judge the times relative to the time it takes to run a mile, but you don't want to include the origin: you don't want to go back to 0 AD. We don't have good records until the 20th century.

Hint: here are some matplotlib calls to investigate

```
plt.scatter()     # Be sure to investigate the optional parameters
plt.legend()
plt.xlabel()
plt.ylabel()
plt.ylim()
```

Your Solution

In [13]:
```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

filename = "WorldRecords.csv"

# read in data
df = pd.read_csv(filename)


# filter to men's and women's mile events
df2 = df[(df['Event'] =='Mens Mile') | (df['Event'] == 'Womens Mile')]

x = df2['Year']
y = df2['Record']


#build chart
sns.scatterplot(x=x, y=y, hue='Event', palette='deep', data=df2)

#title, label & legend
plt.axhline(240, color = 'green', ls='--', label = "4-Minute Mile")
plt.ylim(0,300)
plt.legend(loc='lower left')
plt.title("Men's and Women's Mile Records")
plt.xlabel('Year')
plt.ylabel('Time in seconds')


plt.show()
```



3) People

We define a class Person which describes a citizen with a name. Students and Employees are subclasses of Persons.

You will need to redefine, or override, methods for the subclasses to make the tests below pass.

We have made a start defining a Student, but have done nothing for Employee. Read the Unit Tests to see what you will need to add or change.

In [14]:
```
class Person:

    def __init__(self, first, last):
        self.firstname = first.capitalize()
        self.lastname = last.capitalize()

    def __str__(self):
        return self.firstname + " " + self.lastname

    def __eq__(self, other):
        return (self.firstname == other.firstname) \
            and (self.lastname == other.lastname)

    def is_employed(self):
        return False


class Student(Person):
    "Person who is a student"

    def __init__(self, first, last, school, id):
        # Call Superclass to set common information
        super().__init__(first, last)
        self.school = school
        self.id = id

    def __str__(self):
        # Call Superclass to display common information
        return super().__str__() + ", " + str(self.id) + ' at ' + self.school

    def __eq__(self, other):
        if not isinstance(other, Student):
            return False
        return super().__eq__(other) and (self.id == other.id) and (self.school == other.school)

class Employee(Person):
    "Person who is employed"
    def __init__(self, first, last, company, id):
        super().__init__(first, last)
        self.id = id
        self.company = company

    def __str__(self):
        # Call Superclass to display common information
        return super().__str__() + ", " + str(self.id) + ' at ' + self.company

    def __eq__(self, other):
        if not isinstance(other, Employee):
            return False
        return super().__eq__(other) and (self.id == other.id) and (self.company == other.company)

    def is_employed(self):
        return True
```

Unit Tests for Person

In [15]:
```
def test_person():

    # People
    man1 = Person("Homer", "Simpson")
    man2 = Person("homer", "simpson")
    assert man1 == man2
    assert not man1.is_employed()
    assert man1.__str__() == 'Homer Simpson'
    assert man2.__str__() == 'Homer Simpson'

    # Students
    woman1 = Student("Marge", "Simpson", 'Simmons', 107)
    woman2 = Student("Marge", "Simpson", 'Wheelock', 153)
    assert woman1.__str__() == "Marge Simpson, 107 at Simmons"
    assert woman2.__str__() == "Marge Simpson, 153 at Wheelock"
    assert not woman1 == woman2

    # Employees
    moe1 = Employee("Moe", "Szyslak", 'Tavern', 153)
    assert moe1.__str__() == "Moe Szyslak, 153 at Tavern"
    assert not moe1 == woman2

    moe = Employee("Moe", "Szyslak", 'Tavern', 153)
    assert moe.__str__() == "Moe Szyslak, 153 at Tavern"
    assert not moe == woman2

    waylon = Employee("Waylon", "Smithers", "Springfield Power", 2)
    assert not moe == waylon

    # Cross Check
    moe2 = Student("Moe", "Szyslak", 'BC', 153)
    assert moe2.__str__() == "Moe Szyslak, 153 at BC"
    assert not moe == moe2
    assert not moe2 == moe

    print('Success!')

test_person()
```

```
Success!
```

4) Anagram

Each word in words.txt belongs to a set of words that are anagrams. Some sets have a single elment - no word is an anagram for 'aa'. Some words have a single anagram (veins and vines). Some words have many anagrams. We are looking for the largest set of anagrams. Take a filename and return a list of the sets of anagrams, sorted by size. The unit tests look for the largest sets.

Your function should return a list of tuples: each tuple holds the length of the set, and then a list of members, such as

```
[ (3,['anergias','angaries','arginase']), (3,['amain','amnia','anima']), (3,['alien','aline','anile']) ...]
```

For the file shorterwords.txt, which holds 5K words, these are the three largest sets of anagrams.

There are two challenges to this problem: finding the right answer, and finding the right answer quickly. We will give extra credit if you can find the sets for words.txt in under 2 seconds. It takes my solution a fraction of a second on an 8 year old laptop.

We provide Unit Tests to call your function three times: once to examine a file 5K items shorter.txt, once for 10K items, short.txt, and once to examine the full list, words.txt. You only need to define find_anagrams once, in the first cell.

```
from typing import List

def find_anagrams(path: str) -> List:
    "Find the largest set of anagrams, return as sorted list"
    pass
```

A dictionary would help: but what mapping should it represent?

Your solution

In [7]:
```
from typing import List
from collections import defaultdict

def find_anagrams(path: str) -> List:
    """
    "Find the largest set of anagrams, return as sorted list"
    """

    try:
        with open(path, 'r') as f:
            words_list = [word.strip() for word in f]

        # create list of all words sorted alphabetically
        sorted_words = [(''.join(sorted(word))) for word in words_list]

    except FileNotFoundError:
        print(f"Houston, we have a problem, could not find file {path}")

    # pair each word to its sorted self
    pairs = [(sorted_words[x], words_list[x]) for x in range(len(words_list))]

    # defaultdict - sidesteps KeyErrors!
    agrams =defaultdict(list)
    for x, y in pairs:
        agrams[x].append(y)

    #item of length 1 means not an anagram
    sol = []
    for item in agrams:
        if len(agrams[item]) > 1:
            sol.append((len(agrams[item]),agrams[item]))

    return sorted(sol, reverse=True)
```

Search the file of 5K words

```
lst = find_anagrams('shorter.txt')
for anagrams in lst[:3]:
    print(anagrams)
```

Should yield, in some order. The times might be aspirational.

```
(3, ['anergias', 'angaries', 'arginase'])
(3, ['amain', 'amnia', 'anima'])
(3, ['alien', 'aline', 'anile'])
CPU times: user 11.9 ms, sys: 4.13 ms, total: 16 ms
Wall time: 49.1 ms
```

In [8]:
```
%%time
lst = find_anagrams('shorter.txt')
for anagrams in lst[:3]:
    print(anagrams)
```

```
(3, ['anergias', 'angaries', 'arginase'])
(3, ['amain', 'amnia', 'anima'])
(3, ['alien', 'aline', 'anile'])
CPU times: user 5.6 ms, sys: 98 µs, total: 5.7 ms
Wall time: 5.13 ms
```

Unit test on set of 10K Words

You will need a copy of the 10K word file, short.txt.

The test below should show the 6 largest sets of anagrams

The first line of my output is

```
(6, ['abets', 'baste', 'bates', 'beast', 'beats', 'betas'])
...
CPU times: user 15.5 ms, sys: 1.73 ms, total: 17.2 ms
Wall time: 35.4 ms
```

In [9]:
```
%%time
lst = find_anagrams('short.txt')
for anagrams in lst[:6]:
    print(anagrams)
```

```
(6, ['abets', 'baste', 'bates', 'beast', 'beats', 'betas'])
(5, ['albas', 'baals', 'balas', 'balsa', 'basal'])
(4, ['bestir', 'bister', 'bistre', 'biters'])
(4, ['basest', 'basset', 'bastes', 'beasts'])
(4, ['ardeb', 'barde', 'bared', 'beard'])
(4, ['abet', 'bate', 'beat', 'beta'])
CPU times: user 8.82 ms, sys: 4.08 ms, total: 12.9 ms
Wall time: 11.9 ms
```

Extra credit: find the top 5 sets in full words.txt in less than 2 seconds

You will need to process words.txt, the file of 114K words.

The call '%time' will report how long your run took.

My output starts like this:

```
(11, [...
(11, [...
(10, [...
...
CPU times: user 226 ms, sys: 8.06 ms, total: 234 ms
Wall time: 233 ms
```

In [10]:
```
%%time
lst = find_anagrams('words.txt')
for anagrams in lst[:5]:
    print(anagrams)
```

```
(11, ['apers', 'asper', 'pares', 'parse', 'pears', 'prase', 'presa', 'rapes', 'reaps', 'spare', 'spea
r'])
(11, ['alerts', 'alters', 'artels', 'estral', 'laster', 'ratels', 'salter', 'slater', 'staler', 'stela
r', 'talers'])
(10, ['least', 'setal', 'slate', 'stale', 'steal', 'stela', 'taels', 'tales', 'teals', 'tesla'])
(9, ['estrin', 'inerts', 'insert', 'inters', 'niters', 'nitres', 'sinter', 'triens', 'trines'])
(9, ['capers', 'crapes', 'escarp', 'pacers', 'parsec', 'recaps', 'scrape', 'secpar', 'spacer'])
CPU times: user 144 ms, sys: 3.93 ms, total: 148 ms
Wall time: 147 ms
```

Post Mortem

How long did it take you to solve this problem set?

Did anything confuse you or cause difficulty?

In [ ]:
```
# Your thoughts
# Sudoku was straight-forward. Prob 2 (chart) straight-forward; took about 30-45 minutes of reading stuff


# Prob 3, People class - I was able to figure out how to pass the problem, but still have
# no confidence that I actually understand what the heck I'm doing. I understand the concept
# of classes - basically templates - and instances of classes - objects made from that template.
# But I'm not fully understanding the logic behind the syntax and usage. Basically a lot of
# trial and error involved and I don't understand enough to have any sort of structured approach to
# the trial and error. I mean, sometimes we use 'self' othertimes we don't. And I don't yet see why.


# This is quite different with the other problems we've been doing, where I feel like I have a good grasp
# of how to manipulate various data types and structures. Of course sometimes I don't fully understand
# how something works, but that's a details problem, not a concept problem.


# Anagrams - I spent quite a while on this, full-on hacker mode.
# I ultimately spent an hour or so sketching things out on my iPad.
# As Prof. Parker noted, time away from the keyboard can be quite helpful. I can't stress enough how
# valuable this time was - it was time very well spent. Doodling various things on my iPad and kind of
# 'seeing' what I was trying to do was a tremendous help.

# I of course first grabbed last week's anagram code and tweaked that..and saw straight away
# that it wasn't going to work. Not just because checking each number vs every other number
# is 100,000 x 100,000 which is a Bigly Number, but because in last week's anagram I
# was doing the sort of both words, right at the end on the return.
# So I'd be doing that sort a few billion times, which seems...inefficient.


# I wasted a lot of time trying to get some dictiontary solution to work.

# This is when I went and spent some time on the iPad, and the first idea that worked was realizing
# I could simply create a list of all the words in alphabetical order at the start.

# The next idea was realizing I could pair each word to its sorted self.

# And the last idea was realizing that I was stupid to try to use the word as the dictionary key,
# since the answer requires us to list the original words...so the *sorted word* has to be the key!
```