CSE 163 **Name:** ______________________

Spring 2019

Practice Exam 1

5/10/2019

Time Limit: 50 Minutes

This is a practice exam for Exam 1. This should be a good indicator of what the actual exam will look like in terms of formatting and the types of things we want you to show us that you know. There are a couple ways this exam is different:

1. Some of the programming questions give you a little less space than necessary to write your answer. The actual exam will have plenty of space for you to write answers.
2. We were not very careful about making sure this practice exam was 50 minutes long. We thought it would be better to air on the side of too long for practice so you can get a bit more breadth in the types of things we are looking for. We will be very particular on the actual exam that it is the right length for 50 minutes.

Remember, you get a cheat sheet that you can use both sides for, and the list of methods found on the course website will be printed with your exam for you to use as a reference.

---

1. Your English professor sent back your essay, and unfortunately, you didn't do very well. However, your professor is giving you a chance to regain some points. All you have to do is go through your essay and remove all occurrences of a set of words given by your professor. As a clever programmer, you know you can do this quickly with Python.

   Write a function called `remove_words` that takes two parameters, the first is the name of a file containing your essay and the second is a set of words to remove. Your function should print out the new essay with all instances of the words to be removed left out and return the number of words removed.

   The printed essay should have one space between every word and the words should appear in the same line number that they appear in the original essay; it is okay if the printed essay has trailing whitespace on the lines since that is difficult to remove this. Don't worry about any special cases, such as if the word is followed by a comma, or is capitalized.

   For example, if we had a file called essay.txt with the contents:

   > *The Gettysburg address was like cool and all lol but have you heard*
   > *"Now that I Found You" by Carly Rae Jepsen? She is like totally the best.*

   Then the call `remove_words('essay.txt', {'lol', 'like'})` would return 3 and print:

   > *The Gettysburg address was cool and all but have you heard*
   > *"Now that I Found You" by Carly Rae Jepsen? She is totally the best.*

   Hint: There are two ways you can handle printing the words on a line

   - `print` takes an optional named-parameter called `end` that you can specify to be the empty string. Example: `print('hello', end='')` would print hello without a newline afterwards
   - Build up each line as a string and print once the line is complete.

Solution:

```
def remove_words(essay_file, words_to_remove):
    with open(essay_file) as f:
        count = 0
        for line in f.readlines():
            for word in line.split():
                if word in words_to_remove:
                    count += 1
                else:
                    print(word, end='␣')
            print()
        return count
```

2. For the following problems, we will be working with the following dataset of food in a grocery store. There are two main parts to this problem, where the data is represented differently in each part

- 2.a: The data is stored as a list of dictionaries.
- 2.b: The data is stored as a pandas DataFrame.

name	**color**	**price**	**food_group**
broccoli	green	1.5	vegetable
chicken	pink	6	meat
cheddar	yellow	4	dairy
mango	yellow	1	fruit
carrot	orange	5.2	vegetable

(a) Write a function called `color_max_price_manual` that takes a list of dictionaries that represents the data above and returns a dictionary that indicates the most expensive price for each color of food. The returned dictionary should have colors as keys and the largest price for that color as values. For the dataset above, `color_max_price_manual(data)` would return:

{`'green': 1.5, 'pink': 6, 'yellow': 4, 'orange': 5.2`}

The order of the keys in the dictionary does not matter. For full credit, your solution should run in O(N) time where N is the number of rows in the dataset.

Solution:

```
def color_max_price_manual(data):
  result = {}
  for row in data:
    color = row['color']
    if color in result:
      result[color] = max(result[color],
                          row['price'])
    else:
      result[color] = row['price']
  return result
```

(b) For the following parts, we will assume we have parsed the above dataset in a `DataFrame` named `data`.

i. Consider the following piece of code (newlines added for readability).

```
data[
    (data['price'] < 3) & (
        (data['color'] == 'yellow') |
        (data['food_group'] == 'vegetable')
    )
]['price'].max()
```

First, what is the type of the value this expression produces (**select one**)

◯ `DataFrame`

◯ `Series`

[✓ **`float`**]

◯ `None`

◯ This code causes an error

Second, write the value this expression evaluates to. If you write out a `DataFrame` or a `Series`, you do NOT need to write out the index but if you write a `DataFrame`, you must indicate the column names. If you answered "error" to the last question, explain why.

Solution: 1.5

ii. Write a function called `color_max_price_pandas` behaves exactly the same as 2.a except it takes a pandas `DataFrame` as a parameter instead of the list of dictionaries. The return type should still be a dictionary where the order of the keys does not matter.

Like in the homeworks, for full credit your solution must not use any loops or comprehensions, except to translate the data to a dictionary at the end (this loop should not have any computation in it though).

Translating to a dictionary is worth relatively few points in this problem, so if you don't know how to do this, just write code to compute values described.

Space is provided on the next page.

Solution:

```
def color_max_price_pandas(data):
  return dict(data.groupby('color')['price'].max())
```

3. For this problem, we will use the Iris dataset we talked about in class, a small sample is shown below. For the 3.a, we will assume the small sample is the whole dataset for simplicity purposes, but for the second part about machine learning, we will assume we have a much larger dataset that is well representative of what we would see in the wild.

sepal_length	**sepal_width**	**petal_length**	**petal_width**	**species**
1	2	3	4	setosa
2	3	4	1	versicolor
3	4	1	2	versicolor
4	1	2	3	virginica

(a) For this part, we will focus on data visualization using the `seaborn` library.

i. In the space below, write the code to draw a scatter plot of petal width on the x-axis and petal length on the y-axis. You should also make each point indicate the species as the plant by changing the point's color. Assume that we have already imported `seaborn` and renamed it to be `sns` like normal.

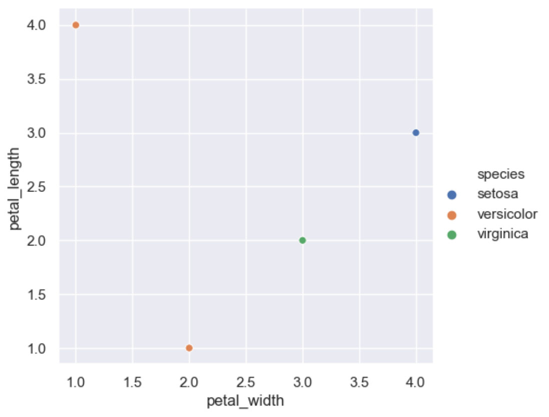
Solution:

```
sns.relplot(x='petal_width', y='petal_length',
            data=data, hue='species')
```

ii. Sketch the scatter plot you created in the last problem. It does not need to be pixel perfect, but it should be clear which points correspond to which samples. The graph should be well labeled so the reader can understand the data being presented. We expect most people to not have colored writing utensils with them, so you should actually draw the points as different shapes to indicate color. You should use the following shapes for each species:

- Setosa: square
- Versicolor: circle
- Virginica: star

Solution: (yours would have shapes instead of colors)



(b) This problem involves training a machine learning model to predict the species of the flowers.

We will make the following assumptions for each problem:

- We are working with the full iris dataset that representative of data in the wild.
- We have already run the code for the proper imports and stored the data in a variable named iris.

```
X = iris[iris.columns != 'species']
y = iris['species']
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.2)
model = DecisionTreeClassifier()
model.fit(X_train, y_train)
y_test_predict = model.predict(X_test)
print(accuracy_score(y_test, y_test_predict))
```

For each problem below, we will describe changes to this code above and you will be asked if this would be an okay change to train the model correctly. Remember, our goal when training a model is to learn a model that will perform well on future data and to do so, we generally want to make a good prediction for if we think this model will do well in the future.

i. Consider if we were to change the code above in the following way

```
Line 1:  X = iris
```

Is this setup appropriate such that we train a model that we can use on future data that has not been labelled yet and we have a confident estimate of its accuracy on the future data? You should explain your answer in a few sentences in the box below.

◯ True
[✓ **False**]

Solution: This will not work because the input features will contain the label we are trying to predict. This will not work on future data we are trying to predict labels for since we would require a label as one of the features.

ii. Consider if we were to change the code above in the following way

```
Line 5:  model.fit(X, y)
```

Is this setup appropriate such that we train a model that we can use on future data that has not been labelled yet and we have a confident estimate of its accuracy on the future data? You should explain your answer in a few sentences in the box below.

◯ True
[✓ **False**]

Solution: With this cahnge, we are now training on all of our data, including the test set. This will prevent us from getting a good estimate of future error since our test set is no longer unseen data.

4. (a) Write a class called `Clock` that represents a clock that tracks time. When constructed, a `Clock` class should always start at midnight. The `Clock` should keep track of time at the granularity of seconds, but when looking at the time, it should only show hours and minutes.

   When looking at the time, there are two main options which are standard time or military time. Standard time (or 12-hour time) is what most watches will read like "12:01 am" or "1:04 pm". Military time, on the other hand, tries to disambiguate times by using a 24-hour numbering system instead for the hours. For example, the two times listed before are "0:01" and "13:04" respectively in military time.

   The `Clock` class should have the following methods with the described arguments. You should not include any additional arguments for these methods:

Method	**Description**
`__init__(self)`	Constructs an `Clock` at midnight.
`tic(self)`	Advances the `Clock` by one second.
`get_time(self, is_military)`	Returns a string representation of the `Clock` based in either standard or military time (explained above with examples). Notice that seconds are not shown.
`__eq__(self, other)`	Returns true if the other `Clock` represents the same time of day (excluding seconds), and false otherwise. You may assume other is always a `Clock`.

   Write your response in the box on the next page.

(b) In the box below on the next page, write a short program that constructs a `Clock` object, increments the time by 10 minutes by calling the `tic` method, and then prints the time in military time.

   You do not need to write a main method for this problem, you may write the lines of code directly in the space below.

Solution:

```
clock = Clock()
for _ in range(10 * 60):
  clock.tic()
print(clock.get_time(True))
```

Space for the Clock class definition.

Solution:

```
class Clock:
  def __init__(self):
    self._hour = 0
    self._minute = 0
    self._second = 0

  def tic(self):
    self._second += 1

    if self._second == 60:
      self._second = 0
      self._minute += 1

    if self._minute == 60:
      self._minute == 0
      self._hour += 1

    if self._hour == 24
      self._hour = 0

  def get_time(self, is_military):
    if is_military:
      return str(self._hour) + ':' + str(self._minute)
    else:
      hour = self._hour
      mark = 'am'
      if hour >= 12:
        hour -= 12
        mark = 'pm'

      if hour == 0:
        hour = 12

      return str(hour) + ':' + str(self._minute) + \
          '␣' + mark

  def __eq__(self, other):
    return self._hour == other._hour and \
        self._minute == other._minute
```

5. For the following problems, write the run-time of each function using the Big-O notation. For these problems, we will use n as the variable to describe the length of the input structure. Your answer should be the "smallest" Big-O runtime possible (i.e. you may not say $\mathcal{O}(n^{12})$ as an answer if $\mathcal{O}(n)$ is an answer that is closer to the actual run-time).

   (a)

```
def fun1(nums):
    t = 0
    for num in nums:
        t += 1
        t += num
        t += 1
    return t
```

Solution:

$\mathcal{O}(n)$

   (b)

```
def fun2(nums):
    t = 0
    for x in nums:
        for y in nums:
            t += x * y
    for x in nums:
        t -= x * x
    return total
```

Solution:

$\mathcal{O}(n^2)$

   (c)

```
def fun3(nums):
    if len(nums) <= 2:
        return None
    x = 0
    y = 0
    for i in range(len(nums) - 2):
        z = 0
        for j in range(3):
            z += nums[i + j]
        y += (z / 3)
        x += 1
    return y / x
```

Solution:

$\mathcal{O}(n)$