CSE 163 **Name:** _____
Spring 2019
Exam 1
5/10/2019 **Student Number:** _____
Time Limit: 50 Minutes

Do not open the exam before the exam begins and close the booklet when time is called. Starting early or working after time is called will lead to a -10 deduction. You may write your name and Net ID on the front of the exam before the exam starts.

This exam contains 13 pages (including this cover page) and 5 questions. Some questions have sub-parts. There are 90 possible for this exam with a point breakdown listed below.

You are allowed to have one sheet of paper (both sides) with you as your cheat sheet. All other materials besides writing utensils should be put away before the exam starts. This includes all electronic devices like phones, calculators, and smart watches.

This exam is not, in general, graded on style and you do not need to include comments or imports for your code. Specific questions may specify restrictions about the style of your code that you must follow to receive full credit.

You may not abbreviate any code, such as "ditto" marks or "..." marks. You may write code to the side and indicate where it should be inserted. These markings must be unambiguous and any ambiguity when grading may result in a deduction if your code is not readable. All code and answers should remain within the provided boxes if possible.

You are allowed to ask for scratch paper after the exam starts to use as additional space when writing answers, but you must indicate on the original page for the problem that part of the answer is on scratch paper. Scratch paper must be stapled to the **END** of your exam after you finish the test. Failure to do so may result in your work on scratch paper not being accepted while grading.

Initials: _____

Initial above to indicate you have read and agreed to the rules above.
Failure to initial may result in your exam not being accepted for credit.

Question	Points	Score
1	10	
2	25	
3	25	
4	20	
5	10	
Total:	90	

(This page is left intentionally blank)

1. (10 points) Write a function called `reverse_file` that takes the name of a file as a parameter and that prints out the file line-by-line, but the lines appearing in the reverse order that they appear in the file.

   For example, if we had a file called `poem.txt` and called `reverse_file('poem.txt')`:

   `poem.txt`

   She sells
   sea shells
   by

   the sea shore

   `reverse_file('poem.txt')`

   the sea shore

   by
   sea shells
   She sells

   Hint: Be careful with new-lines. You may assume every line in the original file ends with a single-newline and does not begin with any white space characters. The output should not contain any blank lines unless it's a corresponding blank line from the input.

Solution:

```
def reverse_file(file_name):
  with open(file_name) as f:
    lines = f.readlines()
    for i in range(len(lines) - 1, -1, -1):
      print(lines[i].strip())
```

2. (25 points total) For the following problems, we will be working with the carbon emissions dataset shown below. There are two main parts to this problem, 2.a represents the data as a list of dictionaries and 2.b represents it as a pandas `DataFrame`.

city	country	emissions	population
New York	USA	200	1500
Paris	France	48	42
Beijing	China	300	2000
Nice	France	40	60
Seattle	USA	100	1000

   (a) (10 points) Write a function called `max_epc_manual` that takes two parameters, a list of dictionaries representing the data above and a name of a country, and returns the name of the city in the given country that has the highest carbon emissions per capita (unit of emissions per person).

   For example, if the data described above is stored in a variable called `data`, then the call `max_epc_manual(data, 'USA')` would return `'New York'` because it has an emissions per capita of 0.133 while Seattle's is only 0.1.

   You may assume no population is 0, that all emissions are non-negative, and there are no missing values. If there is a tie in the carbon emissions per capita, you should return the name that appears earlier in the dataset. If there are no rows in the dataset for the given country, your function should return `None`. For full credit, your solution should run in $\mathcal{O}(n)$ time where n is the number of rows in the dataset.

Solution:

```
def max_epc_manual(data, country):
  max_epc = -1
  city_name = None
  for row in data:
    if row['country'] == country:
        epc = row['emissions'] / row['population']
        if epc > max_epc:
          max_epc = epc
          city_name = row['city']
    return city_name
```

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

i. (5 points) Consider the following piece of code (newlines added for readability).

```
dict(data[data['population] >= 50]
     .groupby('country')['emissions'].max())
```

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

- ◯ `DataFrame`
- ◯ `Series`
- √ **`dict` (dictonary)**
- ◯ `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 in at most two sentences.

Solution:

```
{'China': 300, 'France': 40, 'USA': 200}
```

ii. (10 points) Write a function called `max_epc_pandas` behaves exactly the same as 2.a except it takes a pandas `DataFrame` as a parameter instead of the list of dictionaries.

Like in the homeworks, for full credit your solution must not use any loops or comprehensions. Your solution should run in $\mathcal{O}(n)$ time where n is the number of rows in the dataset.

Hint: Remember, you are able to arithmetic between columns of `DataFrame`s. For example if you had a `DataFrame` named `df` and wrote `df['a'] + df['b']`, it would return a new `Series` that, at each index, stores the sum of the values at the same index from column `a` and `b`.

Space is provided on the next page.

Solution:

```
def max_epc_pandas(data, country):
  data = data[data['country'] == country]
  if len(data) == 0:
    return None

  max_idx = (data['emissions'] / data['population']).idxmax()
  return data.loc[max_idx, 'city']
```

3. (25 points total) For this problem, we will use a modified dataset from Problem 2 that has an additional column for if the city is environmentally friendly. 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. For each problem, we will assume the data has been parsed as a `DataFrame` in a variable called `data`.

city	country	emissions	population	is_green
New York	USA	200	1500	no
Paris	France	48	42	no
Beijing	China	300	2000	no
Nice	France	40	60	yes
Seattle	USA	100	1000	yes

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

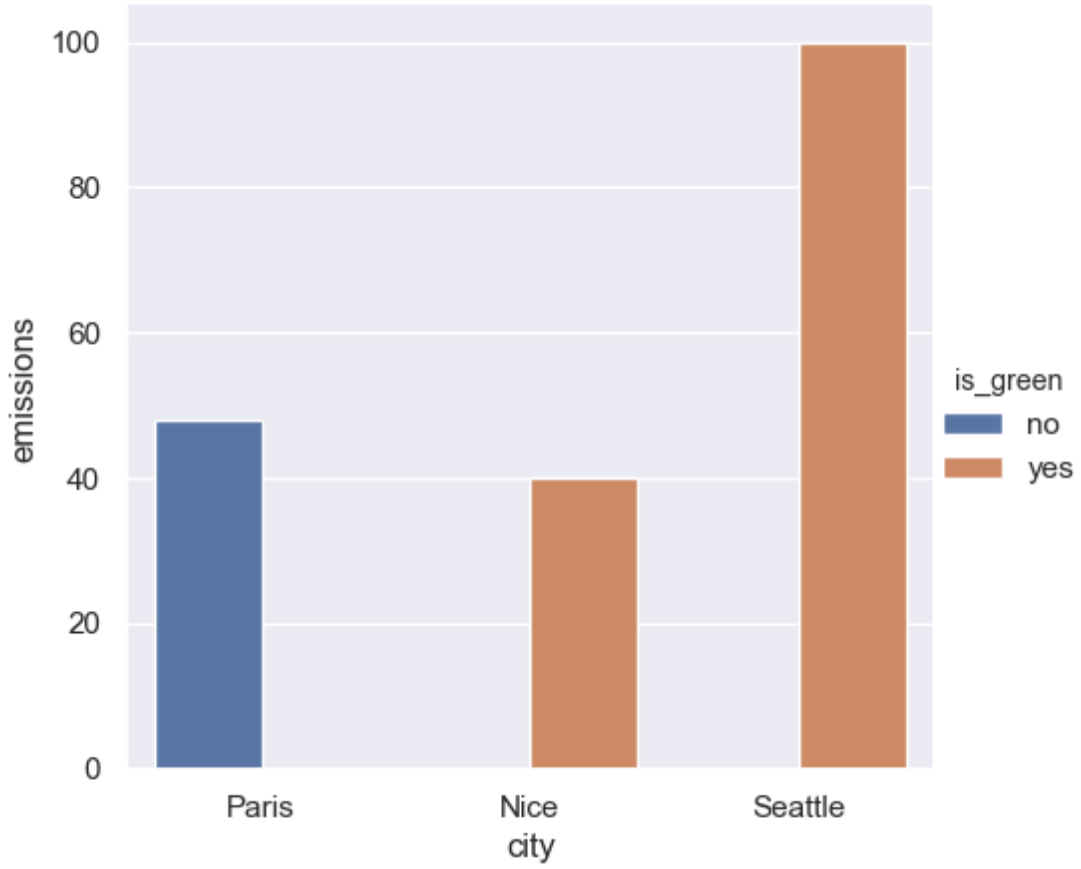
      i. (9 points) In the space below, write the code to draw a bar plot of that shows the emissions of each city that has less than or equal to 1000 people for a population. The plot should color each city to indicate whether or not the city is "green". Assume we already ran `import seaborn as sns` and `sns.set()`.

Solution:

```
sns.catplot(x='city', y='emissions', kind='bar',
            data=data[data['population'] <= 1000],
            hue='is_green')
```

ii. (6 points) Sketch the bar plot you created in the last problem. It does not need to be pixel perfect, but it should accurately convey the data. The graph should have labeled axes and a legend explaining the colors. For drawing purposes, you should shade in the bars that represent "green" cities and leave the other bars white.

Solution: (yours would have "yes" shaded instead of colors)



(b) This problem involves training a machine learning model to predict whether or not a city is "green".

We will make the following assumptions for each problem:

- We are working with a larger 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 `data`.

Below is the standard code for training a machine learning model on this data. The following questions will ask you about why we need each step in this process. For the following questions, your answer should be a one or two sentence description. A valid answer for any of the questions is "it's not necessary", in which your answer should describe why the step is not necessary.

Remember that our goal in machine learning is to predict the value for future, unlabelled data and to get a good estimate of how our model will perform on that future data.

```
1 X = data[data.columns != 'is_green']
2 X = pd.get_dummies(X)  # transforms categorical data
3 y = data['is_green']
4 X_train, X_test, y_train, y_test = \
5     train_test_split(X, y, test_size=0.2)
6 model = DecisionTreeClassifier()
7 model.fit(X_train, y_train)
8 y_test_predict = model.predict(X_test)
9 print(accuracy_score(y_test, y_test_predict))
```

i. (3 points) In Line 1, why is it necessary to remove the `'is_green'` column from the data?

Solution: If we didn't remove the `'is_green'` column, the decision tree would use that value as a feature. This would not work when predicting future data since it would require we already know the value we are predicting as a feature.

ii. (3 points) In Line 2, why is it necessary to call the `get_dummies` function for this dataset?

Solution: All of the rules in the `DecisionTree` are based on if some feature is less than some value. This does not work with categorical data like the `'city'` column so we must transform the data to make this data numeric.

iii. (4 points) Why is it necessary to split the data into a train set and a test set?

Solution: In order to get a good estimate of how we will do on future data, we need to evaluate our model on data it has not yet seen. Splitting into a train and test set lets us test our model on data it has not seen.

4. (a) (15 points) Write a class called `Election` that represents an election where people vote for a candidate. An `Election` has a fixed set of candidates that may be voted for, allows people to vote for their candidate, and can determine the winner of the election. The winner of the election is the person with the most votes.

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

Method	Description
`__init__(self, candidates)`	Creates an `Election` with the given list of candidates. The order of the candidates does not matter. `Election`s start with no votes for any candidate.
`vote(self, candidate)`	Casts a single vote for the given candidate. If the vote is for a candidate not in this election, the vote is ignored.
`winner(self)`	If there is a unique winner for the election (i.e. more votes than any other candidate), this method returns the name of the winner. If there is no unique winner this function should return `None`.
`__eq__(self, other)`	Returns `True` if this election has the exact same set of candidates as the other and each candidate has the same number of votes in both `Election`s. Returns `False` if not. You may assume `other` is an `Election`.

   The `None` return for `winner` is worth relatively few points for the problem. If you can't figure that part out, just consider the case where there is a candidate with more votes than any other. **Write your response in the box on the next page.**

   (b) (5 points) **In the box below**, write a short program that constructs an `Election` between Pedro and Summer, casts 3 votes for Pedro and 2 for Summer, and then prints the name of the winner by calling the `winner` method. You should not access the fields of the `Election` object in this program, you should only call methods.

   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:

```
election = Election(['Pedro', 'Summer'])
for i in range(3):
    election.vote('Pedro')

for i in range(2):
    election.vote('Summer')

print(election.winner())
```

Solution:

```
class Election:
  def __init__(self, candidates):
    self._votes = {c: 0 for c in candidates}

  def vote(self, c):
    if c in self._votes:
      self._votes[c] += 1

  def winner(self):
    top_name = None
    top_count = -1
    for c in self._votes.keys():
        votes = self._votes[c]
        if votes > top_count:
            top_count = votes
            top_name = c
        elif votes == top_count:
            top_name = None
    return top_name

  def __eq__(self, other):
    return self._votes == other._votes
```

5. (10 points total) 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) (2½ points)

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

  for i in range(len(nums)):
    t = t + i
  return t
```

Solution: $\mathcal{O}(n)$

   (b) (2½ points)

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

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

   (c) (2½ points)

```
def fun3(nums):
  t = 0
  for i in range(3):
    for j in range(len(nums)):
      for k in range(10):
        t += i * j * k
  return t
```

Solution: $\mathcal{O}(n)$

   (d) (2½ points)

```
def fun4(nums):
  return max(nums) / min(nums)
```

Solution: $\mathcal{O}(n)$

- **Built-in Python functions**
  - `print(*strings)`
  - `range(end)`
  - `range(start, end[, step])`
  - `abs(v)`
  - `min(v1, v2) / max(v1, v2)`
  - `sum(v1, v2)`
  - `open(fname)`
  - Types: `int(v)`,`float(v)`,`str(v)`,`bool(v)`
- **String methods**
  - `upper()`, `lower()`
  - `find(s)`
  - `strip()`
  - `split()`
- **List methods**
  - Construct: `list()` or `[]`
  - `append(val)`
  - `extend(lst)`
  - `insert(idx, val)`
  - `remove(val)`
  - `pop(idx)`
  - `index(val)`
  - `reverse()`
  - `sort(key=None)`
- **Set methods**
  - Construct: `set()`
  - `add(val)`
  - `remove(val)`
- **Dictionary methods**
  - Construct: `dict()` or `{}`
  - `keys()`
  - `values()`
  - `items()`
- **File methods**
  - `readlines()`
  - `read()`
- **Pandas methods**
  - Parse: `pd.read_csv(file_name)`
  - `mean()`
  - `min() / max()`
  - `idxmin() / idxmax()`
  - `count()`
  - `unique()`
  - `groupby(col)`
  - `apply(fun)`
  - `isnull() / notnull()`
  - `dropna() / fillna(v)`
  - `sort_values(col)`
  - `sort_index()`
  - `nlargest(n, col)`
- **Pandas fields**
  - `index`
  - `loc[row, col]`
- **Seaborn methods**
  - `sns.catplot(x, y, data, kind[, hue])`
    kind: [“count”, “bar”, “violin”]
  - `sns.relplot(x, y, data, kind[, hue[, size]])`
    kind: [“scatter”, “line”]
  - `sns.regplot(x, y, data)`
- **sklearn methods**
  - `sklearn.metrics.accuracy_score(y_true, y_pred)`
  - `sklearn.metrics.mean_square_error(y_true, y_pred)`
  - `sklearn.model_selection.train_test_split(X, y, test_size)`
- **sklearn model classes**
  - `sklearn.tree.DecisionTreeClassifier()`
  - `sklearn.tree.DecisionTreeRegressor()`
- **sklearn model methods**
  - `fit(X, y)`
  - `predict(X)`
- **Special object methods**
  - `__init__`
  - `__repr__`
  - `__eq__`