1. Using regular Python (no pandas, no csv module), change the comma separated file, scores.csv, to a tab separated file, scores.tsv. See the before and after below.

scores.csv (commas)

```
netid,first,last,h1,h2,h3,h1_tot,h2_tot,h3_tot
abc123,alice,chiu,10,15,9,12,20,10
def456,dan,frick,12,14,8,12,20,10
hij789,helen,jane,,12,,12,20,10
```

scores.tsv (tabs)

```
name  hw_score
alice chiu  0.81
dan frick   0.81
helen jane  0.29
```

a) **Remove** the **netid** column.
b) **Combine first name** and **last name** into a single column, name
c) Take the **total of the homework scores** (h1, h2 and h3) and **divide** them by the **total of the possible points** for each homework (h1_tot, h2_tot, and h3_tot)
d) The values for all of the homework numbers are either numbers or no value (it is not required to handle the case where the value is not numeric)
e) If there is no value for a homework score, count it as 0
f) **You must use at least one list comprehension** in your implementation

```
with open('scores.csv', 'r') as fread:
    with open('scores.tsv', 'w') as fwrite:
        lines = fread.readlines()[1:]
        fwrite.write('name\thw_score\n')
        for line in lines:
            parts = line.split(',')
            hws = parts[3:6]
            hws = [0 if n == '' else int(n) for n in hws]
            tots = [int(n) for n in parts[6:]]
            pct = sum(hws) / sum(tots)

            name = f'{parts[1]} {parts[2]}'
            fwrite.write(f'{name}\t{pct:.2f}\n')
```

2. Write a generator, repeat_val. It should have two arguments, a value and a number to repeat. Example usage:

```
for letter in repeat_val('Y', 3):
    print(letter)
Y
Y
Y

def repeat_val(val, n):
    for i in range(n):
        yield val
```

3. Consider the following variable definition:
   `s = 'pythonisAWESOMEcantWaitFORMidterm'`

   Count the number of uppercase letters in s by writing a one line function named count_caps (a variable set to a lambda is ok too!); do this two ways:

   a) using a list comprehension in some part of the solution
   b) using findall with a regular expression as part of the larger solution

```
def count_caps(s):
  return len([ch for ch in s if ch.isupper()])

def count_caps(s):
  return len(re.findall('[A-Z]', s))
```

4. Now assume you have a list of strings defined as follows:

   `words = ['DataBAE', 'MidtermPrep', 'pYtHoN']`

   a) Use a list comprehension to create a new list from the original list of words above, where the strings in the new list only have 3 or more uppercase letters (any other words are left out): `['DataBAE', 'pYtHoN']` You can reuse the function defined in an earlier question if it's helpful.

```
[w for w in words if count_caps(w) >= 3]
```

   b) Do the same as above, but instead of giving back the original words, give back the word with only the first letter uppercase, and all other letters lowercase:
   `['Databae', 'Python']`

```
[w[0].upper() + w[1:].lower() for w in words if count_caps(w) >= 3]

# or

[w.title() for w in words if count_caps(w) >= 3]
```

   c) Use a list comprehension to give back a list containing the highest unicode code point for each letter in the list element. Remember that ord converts a number to its unicode code point whereas chr converts a code point to its corresponding number:
   # it's t for each word, so result is actually [116, 116, 116].

```
[ord(max(list(w), key=ord)) for w in words]
```

5. Now assume you are given the follow Series

```
s = Series(
    data=[“MidtermPrep”,”postgreSQLisAWESOME”, “pYtHoN”,”Foo, “bar,],
```

```
    index=[“cat”, “dog”, ”fish”, “chicken”, 'fly']
)
```

a) Write two ways to retrieve the first element, `MidtermPrep`

```
s['cat']
s[0]
```

b) Name some differences between a Series and a dictionary.

So many... here are a few:

- **Can have duplicate keys (labels) in Series**
- **A Series is typed**
- **Can be indexed with both a position and key (label)**
- **Can be indexed with a list of booleans!**
- **Supports vectorized operations**

c) In one line, create a new Series with all words in s uppercase

```
# many ways! map, or even easier, use str accessor!
s.map(lambda w: w.upper())
s.map(str.upper)
s.str.upper()

# can fall back to regular python as well
pd.Series([word.upper() for word in s.values], s.index)
```

d) In one line, create a new Series with all words in s with exclamation points added to the end – the number of exclamation points should match the number of characters in the original word:

```
s.map(lambda w: w + len(w) * '!')
```

e) Return the length of a string in the series that has the most uppercase letters.

```
s.apply(count_caps).max()
```

f) Give back the word with the most uppercase letters

```
max(s.values, key=count_caps)
```

6. Consider some data given in table called **exams** below

id	first_nam	last_name	course	midterm	final	pass_fail

	e					
1	Jack	Bar	Algorithms	81	87	N
2	Bill	Foo	Web Dev	90	75	N
3	Jack	Bar	Web Dev	72	90	N
4	Bill	Foo	Algorithms	50	75	Y
5	Jane	Bar	Algorithms	94	89	N
6	Jane	Bar	Compilers	65	70	N

a) What might the create table statement look like to construct a table that would contain these column names and values?

```
CREATE TABLE exams (
    id serial NOT NULL PRIMARY KEY,
    first_name varchar(100),
    last_name varchar(100),
    course varchar(100),
    midterm smallint,
    final smallint,
    pass_fail varchar(1)
);
```

b) Write a SQL query to find the average midterm score for each course in the table **exams.** Sort the results in descending order, such that the course with the highest average midterm score appears first. Show the name of the course... and the average with two decimal places. Assume that the types of the column are appropriate such that whatever functions you use will work.

```
select course, round(avg(midterm), 2) as midterm_avg
from exams
group by course
```

```
order by midterm_avg desc;
```

c) Assume we define a “pass” for a student in a given course as having an average of their midterm and final scores greater than or equal to 70. Write a SQL query to return the first and last names of students who successfully passed Algorithms.
Note: From the given data, the query should return
“Jack, Bar”
“Jane, Bar”
in any order.

```
select first_name, last_name
from exams
where (midterm + final) / 2 > 70
  and course = 'Algorithms';
```

d) It’s cumbersome having to calculate whether or not a student has passed. Using a series of SQL statements, modify your table in such a way that you will not have to manually calculate pass or fail for future queries.

```
alter table exams add column pass boolean;
# ok to use different type, but may have to use multiple
# updates for that...
update exams set pass = (midterm + final / 2 > 70);
```

e) Let’s consider the “hardest” course as the one having the lowest average final exam score of all courses. Write an SQL query that returns the names of the courses from “hardest” to “easiest”.

Note: From the given data, the query should return “Compilers” first, followed by “Web Dev” and Algorithms.

```
select course, avg(final)
from exams group by course
order by avg(final);
```

f) Create a report that displays the name of each course along with the number of students enrolled that aren’t pass fail... only displaying the courses with more than 1 student enrolled, and lastly, sorted by the name of the course in alphabetical order.

```
select course, count(id)
from exams
where pass_fail = 'N'
group by course having count(id) > 1
order by course;
```

7. Since we are working with tabular data, recall our work with pandas (DataFrames, Series, etc.), and a database (for our purposes, we can use database to mean both the actual database and the management software on top of it – the DBMS, whereas we'll restrict DataFrame usage to file data, and data retrieved form the web). Briefly describe the pros using pandas and the pros of using databases (with features being *mostly exclusive* to the particular technology).

   These are just some pros... there can be more!

   Pandas pros:
   - **Uses same language as "application code"**
   - **Program paradigm may be more familiar (procedural – as in how, rather than declarative – as in what) for users with previous programming experience**
   - **Entirety of Python ecosystem available**
   - **Arguably more concise syntax (shorter, more terse)**

   Database pros:
   - **Multi-user support**
   - **Transactions (rollback and commit)**
   - **Concurrent operations**
   - **Declarative paradigm may be easier for non-programmers**
   - **Tooling present for data integrity (constraints)**

8. Given the following two DataFrame objects, answer the questions below

```
df1 = pd.DataFrame([
    ['95.01', 'algorithms'],
    ['50', 'algorithms'],
    ['90.2', 'web dev'],
    ['-70', 'algorithms'],
    ['885.0', 'web dev']],
    columns=['grade', 'course'])

df2 = pd.DataFrame([
    ['55', 'algorithms'],
    ['92', 'web dev'],
    ['70', 'algorithms'],
    ['100', 'web dev'],
    ['75', 'algorithms']],
    columns=['grade', 'course'])
```

   a) Based on the declaration above, what do you think the types of each column are in df1

      object (str or unicode are ok too as they are used in numpy)

   b) What code can you use to see what the types are of each column?

      df1.info()

c) Modify df1 to clean up the “grade” column – by converting to only whole numbers. It’s ok to either truncate or round up or down. It’s also ok to assume that there are no missing values. Make sure to convert it to a more appropriate type. The original column in df1 must be changed.

```
#A few potential solutions (there are more)
pd.to_numeric(df1['grade']).map(int) # note this truncates
pd.to_numeric(df1['grade']).round() # keeps decimal point

# a crazy way (treat as one line)
df1['grade']
    .map(lambda val: re.match('(^[0-9-]*)\.?.*$', val)[1])
    .astype('int')
```

d) Modify df1 to consider erroneous outlier grades that not in range [0, 100] as missing

```
df1[df1['grade'] > 100] = np.nan
df1[df1['grade'] < 0] = np.nan
```

e) Move the first two columns of df1 to the end (that is, assume default index values... the rows with index 0 and 1 should be at the end):

```
df1.reindex(index=list(df1.index[2:]) + list(df1.index[:2]))
# also ok to hardcode index values instead of adding
# index=[2, 3, 4, 0, 1]
```

f) Combine df1 and df2 into a single DataFrame **df**

```
df = pd.concat((df1, df2))
```

9. Imagine that you are collecting route data from taxi car vehicles that consists of location data points. A single location data point is represented by a tuple of 3 numbers: a timestamp and two floating point numbers: longitude and latitude. A single route then, is represented by a list of such locations, i.e. time, longitude, and latitude tuples, that effectively tell you where the taxi has been driving around. The first tuple in the list are the starting coordinates of the trip, and the last tuple is the “drop-off”.

   Create a class called TaxiRoute that can support storing such data. It should be initialized by passing in a list of tuples representing locations of the taxi in any order and should support retrieving the coordinates of a taxi at a particular time, note that for this we only return 2 values, since we index based on the time. It should also support looping over a route so that the long and lat are returned on each iteration. For instance, here is an example of a TaxiRoute that is initialized with 4 locations and from time 0 to 4.

```
route = TaxiRoute(
```

```
       [(0, 120, -230.5), (1,  85.5,  -240.1),  (2,  40,  102.3),
(3,  125,  -250.5)]
)

print(route[1]) # prints (85.5 -240.1)
print(route[0]) # prints (120, -230.5)
for r in route: # prints out the last 2 elements of every tuple
    print(r)
# (120, -230.5)
# (85.5, -240.1)
# (40, 102.3)
# (125, -250.5)

class TaxiRoute:
    def __init__(self, locs):
        self.locs = locs

    def __getitem__(self, i):
        filtered = [t for t in self.locs if t[0] == i]
        if len(filtered) > 0:
            return filtered[0][1:]
        # returns None otherwise...

    def __iter__(self):
        self.cur = 0
        return self

    def __next__(self):
        if self.cur == len(self.locs):
            self.cur = 0
            raise StopIteration
        ret = self.locs[self.cur][1:]
        self.cur += 1
        return ret
```