Tick 2

- Python warmup exercises — not assessed
- NumPy warmup exercises — not assessed
- Tick 2a — worth 1 mark
- Tick 2b — worth 1 mark
- Tick 2* — not assessed

Python warmup exercises (not assessed)

> These are optional warmup exercises, to get you used to Python coding.

Use the following autograder settings:

```
import ucamcl
GRADER = ucamcl.autograder('https://markmy.solutions', course='scicomp').subsection('notes1')
```

Each question has a label (in brackets). Call `fetch_question` to see a model answer.

```
q = GRADER.fetch_question('ex1')
# view the answer
print(q)
```

Exercise (ex1) from section 1.2.1. In Python, how do you ...

- calculate the base 10 logarithm of 1200
- calculate the tangent of 60 degrees
- calculate the square root of -20

Exercise (ex2) from section 1.3.1. What is the difference between the two commented lines? Do they give the same result?

```
a = [1, 2, 'buckle my shoe']
b = (a, 3, 4, 'knock at the door')
# b[0].append('then')
# b[0] = a + ['then']
print(a, b)
```

Exercise (ex3) from section 1.3.4. If you go overboard with list comprehensions, your code becomes unreadable. What does the following code do? Here `rooms` is from the code in section 1.3.4.

```
'\n'.join(['Room {r} has {', '.join(occs)}}'
          for r,occs in rooms.items() if r is not None])
```

Exercise (ex4) from section 1.3.5. Write a single line of code to sort the names in this list

```
names = ['adrian', 'chloe', 'guarav', 'shay', 'alexis', 'rebecca', 'zubin']
```

by length, breaking ties alphabetically, using list comprehension.

Hint: make a list of `(len(name), name)` then sort it, where `len(s)` gives the length of a string. When Python sorts a list of tuples, it uses lexicographic ordering (https://en.wikipedia.org/wiki/Lexicographical_order).

Exercise (ex5) from section 1.3.5. Let x be a list of numbers. Give a one-line expression to find the number of unique elements in x. *Hint: use a dictionary comprehension to create a dictionary whose keys are elements of x.*

Exercise (ex6) from section 1.4.1. A simple queue can be simulated by the following equations. Let q_t be the queue size just before timestep t, let the service rate be C, and let a_t be the amount of work arriving at timestep t. Then

$$q_{t+1} = \max(q_t + a_t - C, 0).$$

This is called Lindley's Recursion. Write a function `sim(q0,C,a)` to compute the queue sizes. It should accept an initial queue size `q0` and a list `a` consisting of $[a_0, a_1, \ldots, a_{t-1}]$, and it should return a list $[q_1, \ldots, q_t]$. For example,

```
sim(1, 3, [4, 1, 2, 8, 2, 3, 1])
```

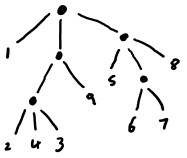
should produce the answer

```
[2, 0, 0, 5, 4, 4, 2]
```

Exercise (ex7) from sections 1.4.1 and 1.4.4. We can represent a tree as a nested list, for example

```
x = [1,[[2,4,3],9],[5,[6,7],8]]
```

Define a function `maptree(f, x)` which applies a function `f` to every leaf of the tree.



NumPy warmup exercises (not assessed)

> These are optional warmup exercises, to get you used to numpy. When you submit an answer, you'll also be shown a model answer. As with the Python warmup exercises, each question has a code (in brackets) which you should use to fetch to see the model answer.

Use the following autograder settings:

```
import ucamcl
GRADER = ucamcl.autograder('https://markmy.solutions', course='scicomp').subsection('notes2')
```

Exercise (ex1) from section 2.2.2. Here is some standard Python code:

```
import math, random
x = random.uniform(-1, 1)
y = random.uniform(-1, 1)
d = math.sqrt(x**2 + y**2)
```

We'd like to repeat this a million times, and find the mean and standard deviation of the `d` values. Implement this using numpy vectorized code.

Exercise (ex2) from section 2.3. For a numpy matrix `a`, what is the relationship between `a.shape` and `len(a)`?

Exercise (ex3) from section 2.3. Look up the numpy help for `np.arange` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.arange.html) and `reshape` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.reshape.html), and use these functions to produce the 3×5 matrix

$$b = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{pmatrix}$$

Look up the help for `np.sum` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.sum.html), and compute the length-5 vector of column sums and the length-3 vector of row sums.

Exercise (ex4) from section 2.3. Find two different ways to use numpy to create the column vector `[[1],[2],...,[n]]`.

Exercise (ex5) from section 2.3. A permutation matrix (https://en.wikipedia.org/wiki/Permutation_matrix) is a square matrix of 0s and 1s, where each row contains exactly one 1, and each column likewise. (The code snippet in section 2.3 of notes, for 'advanced indexing', creates a 3×3 permutation matrix.)

Write code to generate a random $n \times n$ permutation matrix.

Exercise (ex6) from section 2.2.2. In a previous exercise you wrote a Pythonic simulator for a queue, based on the recursion

$$q_{t+1} = \max(q_t + a_t - C, 0).$$

It can be proved that another way to get the same answer is with the formula

$$q_t = q_0 + x_t - \min(0, y_t)$$

where

$$x_t = \sum_{u=0}^{t-1} (a_u - C) \quad \text{and} \quad y_t = \min_{1 \le u \le t} (q_0 + x_u).$$

Given a vector $a = [a_0, a_1, \ldots, a_{t-1}]$,

- compute $x = [x_1, x_2, \ldots, x_t]$ using `np.cumsum` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.cumsum.html)
- compute $y = [y_1, y_2, \ldots, y_t]$ by accumulating (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ufunc.accumulate.html) the function `np.minimum`
- compute $q = [q_1, q_2, \ldots, q_t]$, and check your answer against your Pythonic code.

Exercise (ex7) from section 2.4

When we used numerical optimization in section 2.4, to find the minimum of

$$f(x) = x - 3x^2 + x^4,$$

we used the initial guess $x_0 = 0.5$ and found the local minimum at $x = 1.13$. Now, we'll look for a global minimum. It's good practice to search the parameter space randomly, to avoid Moiré effects (https://en.wikipedia.org/wiki/Moir%C3%A9_pattern).

- Create a vector `x` consisting of values randomly chosen in the interval $[-2, 2]$
- Create a vector `optx` containing the result of running `scipy.optimize.fmin` starting at each of the `x` values.
- Plot your answers with `plt.plot(x, optx, marker='o')`.

You can turn off the diagnostic output with the option `fmin(disp=False)`. You'll need to sort the points before plotting, else the line will go back and forth across the plot.

Tick 2a. Econophysics simulator

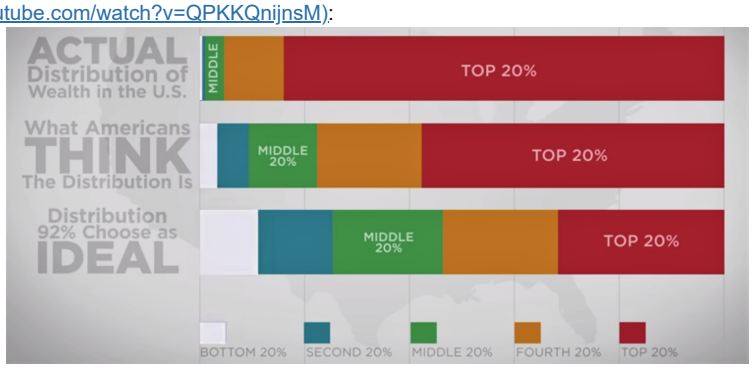
This section is worth 1 mark. Use these autograder settings:

```
import ucamcl
GRADER = ucamcl.autograder('https://markmy.solutions', course='scicomp').subsection('tick2a')
```

This assignment tests your vectorized thinking. You will be asked to run simulations on a population of hundreds of thousands of individuals, over many timesteps. **YOUR CODE MUST USE NUMPY VECTORIZED OPERATIONS** rather than iterating over the population. You may use Python iteration over timesteps.

Introduction

Economic inequality is one of the defining social issues of our age. Yet we have a poor grasp of the scale of inequality, as described in Scientific American (https://www.scientificamerican.com/article/economic-inequality-it-s-far-worse-than-you-think/) and nicely shown in this video (https://www.youtube.com/watch?v=QPKKQnijnsM):



(https://www.youtube.com/watch?v=QPKKQnijnsM)

How does inequality arise? Is it an inevitable outcome of liberal economics, and if so how can it be mitigated by economic policy? These questions have been studied by economists (https://link.springer.com/article/10.1140/epjst/e2016-60162-3) and more recently by (https://phys.org/news/2007-04-world-economies-similarities-economic-inequality.html) physicists (https://arxiv.org/abs/1606.06051). In this assignment you will investigate a simple "econophysics" model of inequality.

Here is a simple model. There are N individuals in the population, each with an initial wealth of £1. Every timestep, we randomly group them into $N/2$ pairs. (Assume N is even.) For every pair, we simulate an economic exchange, as follows. Let the two paired individuals have wealth v and w, and update their wealth according to

$$v_{\text{new}} = R(v + w), \qquad w_{\text{new}} = (1 - R)(v + w)$$

where R is a random number in $[0, 1]$, chosen independently for every pair and at every timestep. This model is loosely inspired by the physics of gases, in which two gas molecules exchange a random amount of energy whenever they collide.

We can measure inequality with the Gini coefficient (https://en.wikipedia.org/wiki/Gini_coefficient),

$$G = 2\frac{\sum_{i=1}^{N} i\, w_{(i)}}{N \sum_i w_{(i)}} - \left(1 + \frac{1}{N}\right)$$

where $w_{(1)}$ is the smallest value, $w_{(2)}$ the second smallest etc. If everyone has the same wealth then $G = 0$; if one person has all the wealth then $G = 1 - 1/N$.

Questions

Question 1. The model needs us to randomly group the population into $N/2$ pairs. We can do this by randomly permuting the vector $[0, \ldots, N-1]$, letting the vector `m1` consist of the first $N/2$ integers and `m2` consist of the rest, and interpreting it as " `m1[i]` is paired with `m2[i]` ".

Write a function `pairs(N)` that returns a tuple `(m1,m2)` where `m1` and `m2` are both vectors of length $N/2$ as described above. For example, if you run `pairs(6)`, you might get the output

```
(array[(3, 0, 1]), array([2, 4, 5]))
```

To submit your answer,

```
q = GRADER.fetch_question('q1')
m1,m2 = pairs(q.n)
ans = {'n': len(np.unique(np.concatenate([m1,m2]))), 's': np.std(np.abs(m1-m2))}
GRADER.submit_answer(q, ans)
```

Question 2. Write a function `kinetic_exchange(v,w)` which takes two wealth vectors `v` and `w`, each of length $N/2$, and returns a tuple `(vnew, wnew)` with two new vectors, according to the kinetic exchange model. To submit your answer,

```
q = GRADER.fetch_question('q2')
v,w = np.linspace(1,5,q.n), np.linspace(1,2,q.n)**q.p
vnew,wnew = kinetic_exchange(v,w)
ans = {'m1': np.mean(vnew), 's2': np.std(wnew)}
GRADER.submit_answer(q, ans)
```

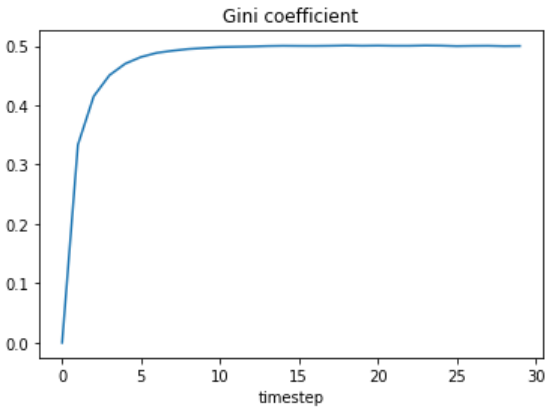
Question 3. Write a function `gini(w)` which takes a vector `w` and returns the Gini coefficient. To submit your answer,

```
q = GRADER.fetch_question('q3')
w = np.linspace(0,1,q.n)**q.p
g = gini(w)
GRADER.submit_answer(q, {'g': g})
```

Question 4. Write a function `sim(N, T)` which runs the kinetic exchange model on a population of N individuals for T timesteps. It should return a pair `(w, gs)` where `w` is the wealth vector after T timesteps, and `gs` is a length T vector where `gs[i]` is the Gini coefficient at timestep i. To submit your answer,

```
q = GRADER.fetch_question('q4')
w,gs = sim(q.n, q.t)
ans = {'gm': np.mean(gs[int(q.t/2):]), 'gs': np.std(gs[int(q.t/2):]), 'ws': np.std(w)}
GRADER.submit_answer(q, ans)
```

Question 5. Simulate a population of 500,000 over 30 iterations. Plot the Gini coefficient as a function of timestep. To be precise, if w_t is the wealth vector after t timesteps then you should plot $\text{gini}(w_t)$ on the y-axis and t on the x-axis. You don't have to submit your plot, but it may be assessed in the ticking session. *Your plot should look something like this:*



Tick 2b. Economic mobility

This section is worth 1 mark. Use these autograder settings:

```
import ucamcl
GRADER = ucamcl.autograder('https://markmy.solutions', course='scicomp').subsection('tick2b')
```

This assignment tests your vectorized thinking. You will be asked to run simulations on a population of hundreds of thousands of individuals, over many timesteps. **YOUR CODE MUST USE NUMPY VECTORIZED OPERATIONS** rather than iterating over the population. You may use Python iteration over timesteps.

Introduction

Some degree of inequality might be acceptable if economic mobility were high, i.e. if everyone had similar chances of reaching either end of the wealth distribution. Economic mobility is often measured by splitting the population into five equal brackets, and measuring the chance of moving between brackets. From the Wikipedia article on economic mobility (https://en.wikipedia.org/wiki/Economic_mobility):

> in terms of relative mobility a report (https://www.brookings.edu/research/economic-mobility-of-families-across-generations/) stated: "contrary to American beliefs about equality of opportunity, a child's economic position is heavily influenced by that of his or her parents." 42% of children born to parents in the bottom fifth of the income distribution ("quintile") remain in the bottom, while 39% born to parents in the top fifth remain at the top.

Let's measure economic mobility by recording the wealth distribution at one timepoint, and again some number of timesteps later, splitting the two distributions into quintiles, and counting what fraction of the population moved by more than one quintile from beginning to end. (In each timestep a median individual might find their wealth increasing or decreasing by around 50%, so one timestep corresponds roughly to several years of human life.) For example, if we have a population of 5000 and we draw up a matrix A where A_{ij} is the number of people who start in quintile i and end up in quintile j, we might get

$$A = \begin{pmatrix} 344 & 313 & 243 & 100 & 0 \\ 266 & 261 & 302 & 167 & 4 \\ 212 & 260 & 225 & 272 & 31 \\ 147 & 143 & 183 & 331 & 196 \\ 31 & 23 & 47 & 130 & 769 \end{pmatrix}$$

(A quick check: the row sums and column sums are all 1000.) The number who moved by more than one quintile is 1148, which is 23% of the population.

Questions

Question 6. In a perfectly mobile economy, where everyone has equal chance of reaching any quintile, what fraction of people are expected to move by more than one quintile?

```
q = GRADER.fetch_question('q6')
GRADER.submit_answer(q, your_answer)
```

Question 7. Write a function `mobility(v,w)` that returns the proportion of people who moved by more than one quintile, where `v[i]` and `w[i]` measure respectively the wealth of individual i at the beginning and end of a time period. *Hint: look up* ***`np.percentile`*** *(https://docs.scipy.org/doc/numpy/reference/generated/numpy.percentile.html#numpy.percentile) and* ***`np.digitize`*** *(https://docs.scipy.org/doc/numpy/reference/generated/numpy.digitize.html).*

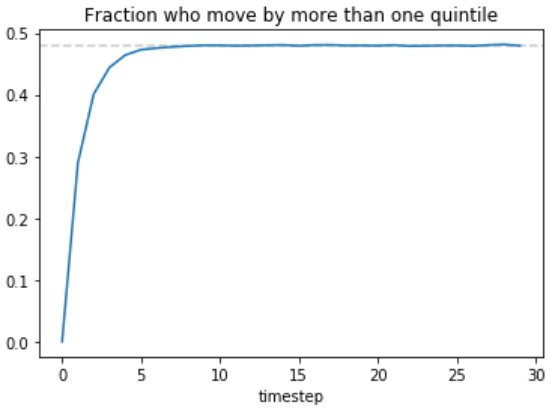
```
# Submitting your answer:
q = GRADER.fetch_question('q7')
v,w = np.arange(q.n)**q.a, np.arange(q.n)**q.a * np.random.random(q.n)
GRADER.submit_answer(q, mobility(v,w))
```

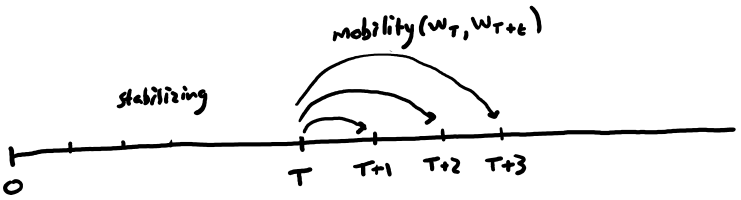
Question 8. Simulate the kinetic exchange model from Assignment 2a long enough for it to stabilize; call this time T, and let the wealth vector be w_T. Run the model t timesteps further to time $T+t$, find the wealth vector w_{T+t}, and compute mobility(w_T, w_{T+t}). It's up to you to decide how to judge stabilization; you don't have to explain your method but you do have to submit a correct answer.

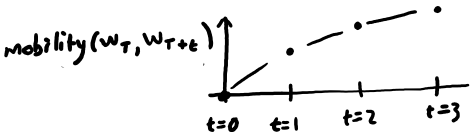
```
# Submitting your answer:
q = GRADER.fetch_question('q8')
# For a population size q.n, measure mobility over from time T to time T+q.t
GRADER.submit_answer(q, your_answer)
```

Question 9. Plot the social mobility for a population of 500,000 as in Question 8, over a sequence of timesteps. To be precise, plot mobility(w_T, w_{T+t}) on the y-axis and t on the x-axis. You don't have to submit your plot, but it may be assessed in the ticking

session. *Your plot should look something like this:*







Tick 2* (not asssessed)

> These are optional further investigations into tax policy, using out econophysics simulator.

Use the following autograder settings:

```
import ucamcl
GRADER = ucamcl.autograder('https://markmy.solutions', course='scicomp').subsection('tick2star')
```

We will investigate variations on the kinetic exchange model, and alternative metrics. The first step is to rewrite the simulator to be more modular. Rewrite the simulator as a function

```
def sim(w0, T, update, metrics):
    ...
    return (w, res)
```

where

- `w0` is either an integer or an initial wealth vector; if `w0` is integer then use initial wealth vector `np.ones(w0)`
- `T` is the number of timesteps to simulate
- `update` is a function to update wealth, with the same signature as `kinetic_exchange`
- `metrics` is a list of functions, each with the same signature as `gini`
- `w` is the wealth vector after `T` timesteps
- `res` is a `T x len(metrics)` matrix, recording the value of each of the metrics at each timestep

Thus the answer to Tick 2a question 5 would be computed by

```
sim(500000, 30, kinetic_exchange, [gini])
```

and the answer to Tick 2b question 9 would be computed by

```
T = ...
w0,_ = sim(500000, T, kinetic_exchange, [])
_,r = sim(w0, 30, kinetic_exchange, [lambda w: mobility(w0,w)])
```

Exercise (ex1). In the kinetic exchange model, the poorest and the richest might swap places after just one transaction, which isn't very likely. Consider a different model for exchange. As before, suppose that two individuals with wealth v and w respectively are paired, but now let their wealth be updated by

$$v_{\text{new}} = v + R\min(v, w), \quad w_{\text{new}} = w - R\min(v, w)$$

where R is now a random number in $[-1, 1]$, chosen independently for every pair at every timestep. The idea is that each party to the exchange puts up a certain amount of money, but no more than they can afford. Implement an update function to model this.

```
# Submitting your answer:
q = GRADER.fetch_question('ex1')
# your_ans = Gini coefficient after q.t timesteps for a popn of size q.n
GRADER.submit_answer(q, your_ans)
```

Exercise (ex2). The Gini coefficient is unfamiliar to many people, and it's easier to communicate "The richest 1% of the population own x% of the wealth." Implement a function

```
topk(w, p)
```

which computes what fraction of the total wealth is owned by the top p of the population.

```
# Submitting your answer
q = GRADER.fetch_question('ex2')
# your_ans = fraction of wealth owned by top 1%, after q.t timesteps for a popn of size q.n
GRADER.submit_answer(q, your_ans)
```

Exercise (ex3). The government can intervene to reduce inequality. Suppose it levies a tax of say 40% on every exchange, collects all the tax revenue every timestep, and distributes it evenly to the entire population. Here's a concrete example, for a population of size 6.

1. Initial wealth values are $[0, 2, 5, 3, 1, 2]$
2. We pair individuals randomly: $(0, 2), (5, 3), (1, 2)$
3. Random exchange amounts pre-tax are $(0, 0), (2.6, -2.6), (-0.4, 0.4)$
4. Exchange amounts post-tax are $(0, 0), (2.08, -2.6), (-0.4, 0.32)$
5. Government revenue is $(2.6 - 2.08) + (0.4 - 0.32) = 0.6$
6. Government redistributes $0.6/6 = 0.1$ to each person
7. Change in wealth is $[0.1, 0.1, 2.18, -2.5, -0.3, 0.42]$
8. New wealth vector is $[0.1, 2.1, 7.18, 0.5, 0.7, 2.42]$

Implement an update function to model this.

```
# Submitting your answer
q = GRADER.fetch_question('ex3')
# your_ans = fraction of wealth owned by top 1%, at tax rate q.taxrate,
# after q.t timesteps for a popn of size q.n
GRADER.submit_answer(q, your_ans)
```

Further investigations.

- Is there a tradeoff between inequality and social mobility? Try different tax rates, measure the inequality and the social mobility, and plot your results.
- The economist Thomas Piketty argues (https://en.wikipedia.org/wiki/Capital_in_the_Twenty-First_Century) that we have entered an age where the return on capital is greater than the growth due to income, and that this leads to higher inequality. We could incorporate income into the model by assigning each individual i a per-timestep income g_i, where the g_i are randomly chosen *a priori*. We could incorporate return on capital into the model, by multiplying wealth by a growth factor every timestep (and rescaling income to account for inflation). Investigate what happens when we combine these two extensions. How well correlated are income and wealth? How does the relationship depend on capital growth rate? Do you agree with Piketty? Does taxation alter the relationship?

Tick 3

- Pandas warmup exercises — not assessed
- Tick 3a — worth 1 mark
- Tick 3b — worth 1 mark
- Tick 3* — not assessed

Pandas warmup exercises (not assessed) ¶

These are optional warmup exercises, to get you used to pandas.

Use the following autograder settings:

```
import ucamcl
GRADER = ucamcl.autograder('https://markmy.solutions', course='scicomp').subsection('notes3')
```

Each question has a label (in brackets). Call `fetch_question` to see a model answer.

```
q = GRADER.fetch_question('ex1')
# view the answer
print(q)
```

Exercise (ex1) from section 3.5. In pandas, to find the most frequent value (the _mode_) of `age_range` in the `stopsearch` dataset, we can use

```
stopsearch['age_range'].mode().values[0]
```

(The `mode()` call returns an object, and `.values[0]` extracts just the value itself.)

How would you generate a table showing the most frequent age range for each combination of ethnicity and gender?

Exercise (ex2) from section 3.4 and 3.5. Given the dataframe

```
df = pandas.DataFrame({'A': [0,0,0,1,1,1], 'B': [0,1,2,0,1,2], 'X': range(6)})
```

how do you produce a table that has rows for A, columns for B, and shows values of X?

Exercise (ex3) from section 3.6. This code produces a dataframe with columns for ethnicity, outcome, and n:

```
sscam = stopsearch.loc[stopsearch.force=='cambridgeshire'].copy()
sscam['outcome'] = np.where(sscam.outcome == 'False', 'nothing', 'find')
df = sscam.groupby(['officer_defined_ethnicity','outcome']).apply(len).reset_index(name='n')
```

Take the subtable with `outcome=='nothing'` , and the subtable with `outcome='find'` , and merge them; then use the merged table to compute the `percent_find` column from section 3.6 of notes.

Exercise (ex4) from section 3.5. As an alternative to the method in exercise ex3, take the indexed array

```
sscam.groupby(['officer_defined_ethnicity','outcome']).apply(len)
```

and convert it to a wide-form dataframe, then use this to compute `percent_find` .

Tick 3a. Analysis of flood data

This section is worth 1 mark. Use these autograder settings:

```
import ucamcl
GRADER = ucamcl.autograder('https://markmy.solutions', course='scicomp').subsection('tick3a')
```

This assignment tests your skill at manipulating dataframes and indexed arrays. **YOUR CODE MUST USE PANDAS AND NUMPY OPERATIONS FOR DATA MANIPULATION** rather than for loops, wherever possible.

Introduction

This assignment asks you to analyse data provided by the UK Environment Agency concerning flooding. The agency offers an API for near real-time data (http://environment.data.gov.uk/flood-monitoring/doc/reference) covering:

- flood warnings and flood alerts
- flood areas which to which warnings or alerts apply
- measurements of water levels and flows
- information on the monitoring stations providing those measurements

In this assignment we will be working with historical data of water level measurements, at several monitoring stations in Cambridge and on the Cam. The dataset is available as a CSV file at https://teachingfiles.blob.core.windows.net/datasets/flood_20191125.csv (https://teachingfiles.blob.core.windows.net/datasets/flood_20191125.csv).

Image by N. Chadwick (http://www.geograph.org.uk/photo/4800494).

Questions

Question 1. Import the CSV file and print out a few lines, choosing the lines at random using `np.random.choice`. The file mistakenly includes records from a River Cam in Gloucestershire, and another River Cam in Somerset. Remove these rows, and store what's left as the data frame `flood`. How many rows are left?

```
# Submit your answer:
GRADER.submit_answer(GRADER.fetch_question('q1'), num_rows)
```

Hint. `DataFrame.drop_duplicates` (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop_duplicates.html) may be useful for seeing what's in the file.

Question 2. Complete this table of the number of entries in this dataset for each town and river.

.	Cambridge	Great Chesterford	Great Shelford	Milton
Bin Brook	2628	0		
River Cam				

```
# Submit your answer
your_ans = ...  # an unstacked indexed array
GRADER.submit_answer(GRADER.fetch_question('q2'), your_ans.values)
```

Question 3. Each water measuring station has a distinct `measure_id` and `label`. Complete this dataframe of the number of measurement stations for each town and river. Use only the Pandas operations for split-apply-combine, don't use any numpy operations or Python `for` loops or list comprehensions.

town	num_stations
Milton	1
Great Shelford	1

```
# Submit your answer. Row order doesn't matter.
GRADER.submit_answer(GRADER.fetch_question('q3'), your_dataframe)
```

Question 4. Each measurement station has low and high reference levels, in columns `low` and `high`. In this dataset, the reference levels are stored for every measurement, but we can verify that every `measure_id` has a unique pair `(low,high)` with

```
assert all(flood.groupby(['measure_id','low','high']).apply(len).groupby('measure_id').apply(len) ==
1), "Reference levels non-constant"
```

Add a column `norm_value`, by rescaling `value` linearly so that `value=low` correponds to `norm_value=0` and `value=high` corresponds to `norm_value=1`. Use `np.nanquantile` to find the *tercile points* (https://en.wiktionary.org/wiki/tercile), the two values that split the entire `norm_value` column into three roughly equal parts.

```
# Submit your answer:
GRADER.submit_answer(GRADER.fetch_question('q4'), [tercile1, tercile2])
```

Question 5. Complete the following dataframe, listing the number of observations in each tercile and the total. (When there are repeated values, it's arbitrary how we assign observations into terciles. To answer this question, use the convention that `med` means `tercile1 <= value < tercile2`.)

label	norm_value_tercile	n	ntot
Bin Brook	med	2466	2628
Bin Brook	high	162	2628
Cambridge Baits Bite	low	2	2813

```
# Submit your answer. Row order doesn't matter.
GRADER.submit_answer(GRADER.fetch_question('q5'), your_dataframe)
```

Question 6. Complete this dataframe, listing the fraction of observations in each tercile per station:

label	low	med	high
Bin Brook	0.000	0.938	0.062
Cambridge Jesus Lock	0.032	0.356	0.612

```
# Submit your answer. Row order doesn't matter. Don't round.
GRADER.submit_answer(GRADER.fetch_question('q6'), your_dataframe)
```

Question 7. Fill in the rest of this indexed array, giving the `low` and `high` values for each measurement station.

label	ref	
Bin Brook	high	1.000
	low	0.057
Cambridge Baits Bite	high	0.294
	low	0.218

```
# Submit your answer. Let your_ans be an indexed array.
GRADER.submit_answer(GRADER.fetch_question('q7'), your_ans.reset_index(name='val'))
```

Hint. There are many ways to approach this. The cleanest is to use `melt` (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.melt.html#pandas.DataFrame.melt)

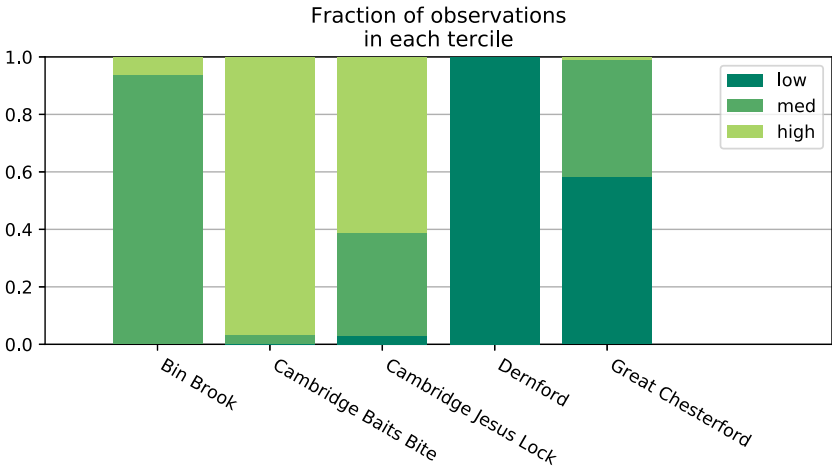
Tick 3b. Plotting

This section is worth 1 mark. There is no automated testing of your answers here, but your code may be assessed in the ticking session.

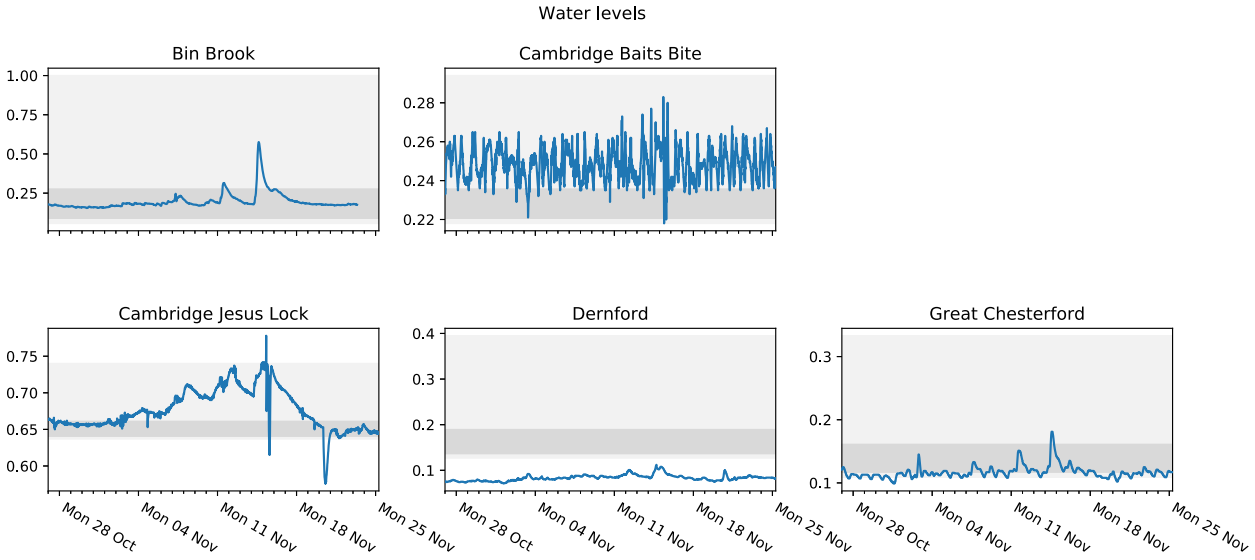
For this assignment you will need to figure out matplotlib's myriad options. You will need to spend time searching (https://stackoverflow.com/questions/tagged/matplotlib) for help. You don't need to be pixel perfect, but you do need to demonstrate that you can control

- aspect ratio
- subplots layout and spacing
- x and y axis ranges, and colour palette
- display of tick labels and gridlines
- legend placement
- titles

Question 8. Reproduce this plot:



Question 9. Reproduce this plot:

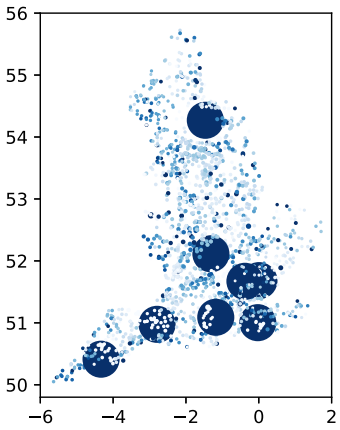


The light shaded area shows the range from `low` to `high` for each station. The dark shaded area shows the inter-tercile range, `low+tercile1*(high-low)` to `low+tercile2*(high-low)` where `tercile1` and `tercile2` are your answers to Question 4. They can be plotted with `ax.axhspan` (https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.axhspan.html#matplotlib.axes.Axes.axhspan). Here are some code snippets that may be useful, for formatting tick labels with dates.

```
import matplotlib
# Date-axis control, taken from http://matplotlib.org/examples/api/date_demo.html
ax.xaxis.set_major_locator(matplotlib.dates.WeekdayLocator(byweekday=matplotlib.dates.MO, tz=pytz.UT
C))
ax.xaxis.set_minor_locator(matplotlib.dates.DayLocator(tz=pytz.UTC))
ax.xaxis.set_major_formatter(matplotlib.dates.DateFormatter('%a %d %b'))
```

Tick 3* (not assessed)

Question 10. The real story in this dataset is about the Yorkshire floods of 2019 (https://en.wikipedia.org/wiki/2019_Yorkshire_floods). Using the full dataset of all readings, plot a map showing the peak water level at each measuring station. Here's an illustration.



You can download the full dataset in three files,

- https://teachingfiles.blob.core.windows.net/datasets/stations_20191125.csv (https://teachingfiles.blob.core.windows.net/datasets/stations_20191125.csv)
- https://teachingfiles.blob.core.windows.net/datasets/measures_20191125.csv (https://teachingfiles.blob.core.windows.net/datasets/measures_20191125.csv)
- https://teachingfiles.blob.core.windows.net/datasets/readings_20191125.csv (https://teachingfiles.blob.core.windows.net/datasets/readings_20191125.csv)

You will have to merge these files: `readings.measure_id` refers to `measures.measure_id`, and `measures.station_uri` refers to `station.uri`. You will also have to filter the readings, and only keep those with `parameter=='Water Level'`.

Question 11. The dataset also includes rainfall measurements. Plot rainfall and water level, for the worst-affected rivers.