Lab 6

Objectives

This week's lab presents you with some data analysis problems. These problems are a bit different from most of the exercises in the labs so far, in that:

- The questions are not clearly stated, and there is no right answer.
- The problems will need a (slightly) bigger python program to solve, not just a single function. It is up to you to decide what is the best way to break up the problem and what functions (if any!) to define.

This can also be a good time to review the material that we have covered in the first half of the course, perhaps to go back to previous labs if you have some unfinished exercises, and to ask your tutors for help with any unsolved problems.

The following section (Preliminaries) describes some techniques that will be useful for solving the data analysis problems. You can skim it, then refer back to it while you are working on the problems. After this comes the description of two problems, looking at Canberra weather data and house market data. Read through both and decide which one you find more interesting to attack first. The last exercise in the lab looks at floating point error analysis.

Preliminaries

Data analysis problems will typically require you to:

1. read data from one or more files;
2. organise the data into a table and convert it to the correct type; and
3. deal with entries in the table where some data is missing.

Ways to do these steps were covered in the lecture in week 6. Here is a quick recap:

CSV files

The file format we will be using here is simple *Comma-Separated Values*, or CSV, files. To read in a CSV file, the following steps suffice:

```
import csv
with open("filename.csv") as csvfile:
    reader = csv.reader(csvfile)
    table = [ row for row in reader ]
```

After this, the variable `table` is a list of lists, with one entry per row in the file. Two things to remember are:

- After reading the file, all entries in all rows will be strings.
- If the file contains a header (that is, the entries in the first row are the names or descriptions of the columns rather than values), the first row in `table` will contain this header. You can use slicing to get a table without the header.

List comprehension

List comprehension is a mechanism in python that allows you to create new lists from a sequence in a very convenient way. You can find an overview of list comprehension in lab 4.

In this lab, you can use list comprehensions to

- Convert selected entries in all rows to a type other than string (such as a number).
- Extract a single column from the table, as a list.
- Select only rows from the table (or entries from a column) that satisfy some criterion, for example not having any blank fields.

For example, the expression

```
col3 = [ row[3] for row in table ]
```

creates a list with the entries in column 3 (i.e., the fourth column, since columns are also zero-indexed in this representation), and the expression

```
table_non_empty_col3 = [ row for row in table if row[3] != '' ]
```

creates a new table containing only the rows in `table` that have a non-empty value in column 3.

The built-in `range` function creates an iterable collection of integers, which can be very useful together with list comprehension (and also `for` loops). For example,

```
ind_non_empty_col3 = [ i for i in range(len(table)) if table[i][3] != '' ]
```

creates a list of the *indicies* of rows in `table` that have a non-empty value in column 3.

The `range` function can take both a start and stop value. For example,

```
years = range(2008, 2019)
```

creates a range of the integers from 2008 to 2018 (note that the range is up to, but not including, the stop value). For more details, read the documentation with `help(range)`.

The selection condition used in a list comprehension can be any expression that evaluates to a truth value. For example, if you have written a function `is_prime(n)` that returns `True` if the number `n` is a prime number (as you may have done in lab 3), then you can create a list of all the prime numbers between 1 and 100 with the expression

```
list_of_primes = [ i for i in range(1,101) if is_prime(i) ]
```

The expression that creates the values in the new list can also be of any kind (as long as it is an expression). Thus, similar to the example shown in the lecture,

```
[ [ row[0], row[1], float(row[2]), int(row[3]) ] for row in table ]
```

creates a new table where all entries in column 2 have been converted to floating point numbers and all entries in column 3 have been converted to integers, while columns 0 and 1 have been left as they are (strings). Remember that trying to convert and empty string into a number (`int` or `float`) will cause a runtime error. One way to solve this is to filter out rows with empty entries first (as shown above) and then convert those that remain. However, you can also use a function that you have defined. For example,

```
[ my_fun(row[3]) for row in table ]
```

returns a list with the result of applying `my_fun` to every entry in column 3, where `my_fun` can be any function (of one argument).

Visualising data

As we have seen a few times in the course so far (for example, in Lab 2), the matplotlib module can be used for plotting data. Start with importing the module and giving it a shorthand name:

```
import matplotlib.pyplot as mpl
```

A basic x-vs-y plot is done with the function

```
mpl.plot(xs, ys, linestyle="none", marker="x")
mpl.show()
```

to create a plot with a point marker (here, an "x") for each xy pair, or

```
mpl.plot(xs, ys, linestyle="solid")
mpl.show()
```

to draw a line instead. The two parameters, `xs` and `ys`, must be sequences of numbers, and must have the same length. (Again, list comprehensions and ranges are useful to create them.) There are also other plotting functions. For example,

```
mpl.hist(values, bins=5)
mpl.show()
```

will create a histogram of `values` (which should be a sequence of numbers) with 5 bins, and plot it as a barchart. To find functions for generating other types of plots and to see more options for customising your plots, read the matplotlib documentation using the built-in `help` function or on-line.

Although data analysis is an exploratory activity, in which you will be trying different things, it is strongly recommended that you write your analysis program into a file that you can run, edit and run again. You should also add comments to your file as you develop it, as a reminder to yourself and others what you were thinking when you wrote it.

For this lab, you may also want to work together with other students, in a pair or a small group. Discsussing how to attack the question, and how to interpret the results you come up with, with others can help you avoid mistakes and improve your understanding.

Weather data

Is it just me, or has the 2018 winter in Canberra been unusually long and cold? (Freezing nights right up to the end of August!)

Fortunately, the Bureau of Meteorology provides free access to historical weather observations, including daily min/max temperatures and rainfall, from its many measuring stations around Australia. Here are the records of daily minimum and maximum temperature from the Canberra Airport station:

- daily-min-temp-CBR.csv
- daily-max-temp-CBR.csv

Data is not always complete. Records from the Canberra airport station only go back to September of 2008, and also between then and now there are some missing entries.

What is in the files

The files are in CSV format. The columns are:

- Column 0: Product code. This is a "magic" string that describes what kind of data is in this file ("IDCJAC0011" for the daily minimum temperature and "IDCJAC0010" for the daily maximum). It is the same for all rows in a file. You can ignore this column.
- Column 1: Station number. This is where the measurement was taken (070351 is Canberra airport). This is the same for all rows in both files, so you can ignore this too.
- Column 2: Year (an integer).
- Column 3: Month (an integer, 01 to 12).

- Column 4: Day (an integer, 01 to number of days in the month)
- Column 5: Recorded min/max temperature (depending on which file). This is a decimal number. Note that the temperature may be missing, in which case this field is empty.
- Column 6: Number of consecutive days over which the value was recorded. This number is 1 for all (non-missing) entries in both files, so you don't need to consider it.
- Column 7: Has this entry been quality checked? 'Y' or 'N'. Entries that have not been checked are not necessarily wrong, it just means they haven't been quality checked yet.

Questions

The question that we are trying to answer is whether the 2018 winter was unusually long and/or cold. To arrive at any kind of answer, you first need to make the question more precise:

- What is "winter" in Canberra? Traditionally, winter is said to be the months of June, July and August.
- How "cold" has the winter been? You could count the number of nights with sub-zero temperature (in each month or over the whole winter), or you could compute an average (of minimum, of maximum, or of the two combined somehow).
- How "long" has the winter been? You could compare the aggregate values (however you define them) between the three months. For example, is the August average minimum temperature significantly higher or lower than what it was in July?
- Is this "unusual"? With 10 years of data (almost), you can see how the 2018 values (however you define them) rank compared to other years. Is it the longest or coldest winter among the last ten?

Testing

Although there may not be a single, right answer to the questions above, there are many answers that are clearly wrong. How do you convince yourself, and others, that your code is correct when the lecturer has not given you a solution to check against? There are a number of ways:

- First, define precisely what each part of your code should be doing. For example, if you write a function to count the number of nights with sub-zero temperature in a given month of a given year, then the value that this function should return is unambiguous, and you can check if it works correctly by counting a few cases by hand.
- Second, sanity check! The count returned by the function in the example above should never be negative and never greater than the number of days in the month.
- Third, cross-check. You can implement the same function in several different ways, or different functions that have a known relationship. For example, if you write a function to count the number of nights with sub-zero temperature in a given month of a given year, and another function to count the number of nights with a lowest temperature that is above zero in a given month of a given year, then the sum of the two counts should equal the number of days in that month. (Remember to adjust for leap years if the month is February!)

- Fourth, ask someone else to look critically at your code. Ask them to try to find cases where it could fail, or give the wrong answer. After practicing this on someone else's code, try it with your own (what happens if the first or last night of the month has negative temperature? what if there is no record for some nights?)

House market data

What is the state and trends of Canberra's house market? Here is a file with past house sales data (scraped from one of the property advertising web sites) for a subset of Canberra suburbs:

- house-sales-data.csv

This file lists only "block sales", that is, sales of properties on separate title land. It does not include unit or apartment sales.

What is in the file

The file is in CSV format. The columns are:

- Column 0: Address (usually, number and street name).
- Column 1: Suburb name.
- Column 2: Date of sale: Day (an integer).
- Column 3: Date of sale: Month (an integer).
- Column 4: Date of sale: Year (an integer).
- Column 5: Price (in thousands of dollars). The price is a decimal number. Note that the price is sometimes missing, in which case this field is an empty string.
- Column 6: Block area in square meters. This should be an integer.

The data is not complete. It's unclear how far back in history it goes (can you find out what are the earliest sales date for each suburb, and how they correspond with when that suburb was first built?), and it's not likely to contain all sales. The sale price is sometimes missing, or given as `0.0`, which is probably not the true price.

The entries in the file are ordered, first by suburb, then by address, and finally by date.

Questions

- What filtering do you need to do to remove incomplete or unreliable entries? How much (in absolute and as a percentage) of the data remains after filtering?
- What is the range of the data? What are the oldest and newest recorded sales? What is the average (and distribution of) number of sales records per address? (The last question is a bit complicated to answer in general. However, you can take advantage of the fact that the rows in the table are ordered by address; that is, all records for the same address appear consecutively in the table.)
- What is the relation between block area and price? Is it approximately linear? If we calculate an average price per square meter, for example averaging over a suburb or a year, and use that to measure differences or trends, would we be making any kind of systematic error?

- What does the price trend over time look like? We expect that, on average, it should be increasing, but is it linear, sublinear or superlinear? How much does the rate of increase vary over time?
- How much does the suburb matter? If you compare the changes in price over a given time horizon (say, the last 5 or 10 years) in different suburbs, how big is the difference?
- How uniform is the change in price within each suburb? Are there some streets that have gone up or down more than others?
- What is the distribution of block sizes within different suburbs, and how does that compare to the distribution over the whole data set?
- This data set does not distinguish between sales of vacant blocks (land without building) and sales of houses. Is there any way you can (programmatically) guess which entries are which? (for example, we might expect houses to sell for more than the same block if it was empty).

Testing

Same as for the problem above

Using other modules (optional)

There are many python modules that can be useful for programming data analysis. For example, the NumPy module lets you represent tabular data as a 2-dimensional array (as long as all data in the table is of the same type, such as numbers) and provides many functions to operate on such arrays. If you want to explore it, have a look at the documentation.

The Pandas module provides a specialised data type for tabular data (`pandas.DataFrame`) and several functions for operating on this type of data. If you want to explore it, read the documentation.

Floating point error analysis

In the lecture on functions (in week 2) we mentioned a way of approximating the derivative of a function f at a point x, by the slope of a straight line through *f(x - d)* and *f(x + d)*. More precisely, the formula is *(f(x + d) - f(x - d)) / (2 * d)*.

As the distance d tends to zero, we expect this approximation to grow closer to the real derivative *f'(x)*. However, this fails to take into account the floating point round-off error in the calculation of *f(x + d)* and *f(x - d)*, which may become larger relative to the size of *2 * d*.

To measure this effect, we can compare the approximation with the true derivative, for cases where the latter is known. For example, if $f(x) = e^x$ (which is available in python as `math.exp`), we know that the derivative is *f'(x) = f(x)*.

(a) Write a function that computes the approximation of *f'(x)*, with a parameter for the distance d. As python allows you to pass functions as arguments, you can write a (simple) function that does this calculation for any function f and point x.

(b) Write another function that calculates the error as the absolute difference between the approximate and true derivative, for given values of x and d, using the exponential function as f.

Generate a series of diminishing values for d, from, say, 0.1 down to 10^{-15}. You can do this with list comprehension and the `range` function:

```
ds = [10 ** -i for i in range(1,16)]
```

Calculate and plot the error for each d-value in this range. What can you observe?

(c) Try the same exercise with some other functions. $f(x) = x^2$ is an interesting case to test because its derivative is a linear function.