# Metadata

```
Course:  DS 5100
Module:  04 Functions HW
Title:   Fighting Forest Fires with Functions
Author:  R.C. Alvarado (adapted)
Datae:   7 July 2023
```

# Student Info

- Name: Jacqui Unciano
- Net ID: jdu5sq
- URL of this file in GitHub https://github.com/jacquiunciano/DS5100-jdu5sq/blob/main/lessons/M04/hw04.ipynb:

# Instructions

In your **private course repo on Rivanna**, write a Jupyter notebook running Python that performs the numbered tasks below.

For each task, create one or more code cells to perform the task.

Save your notebook in the `M04` directory as `hw04.ipynb`.

Add and commit these files to your repo.

Then push your commits to your repo on GitHib.

Be sure to fill out the **Student Info** block above.

To submit your homework, save the notebook as a PDF and upload it to GradeScope, following the instructions.

**TOTAL POINTS: 14**

# Overview

In this homework, you will work with the Forest Fires Data Set from UCI.

There is a local copy of these data as a CSV file in the `HW` directory for this module in the course repo.

You will create a group of related functions to process these data.

This notebook will set the table for you by importing and structuring the data first.

# Setting Up

First, we read in our local copy of the dataset and save it as a list of lines.

In [2]:
```
import os
os.chdir("C:/Users/jacqu/OneDrive/Documents/MSDS/datasets")
```

In [3]:
```
data_file = open('uci_mldb_forestfires.csv', 'r').readlines()
```

Then, we inspect first ten lines, replacing commas with tabs for readability.

In [4]:
```
for row in data_file[:10]:
    row = row.replace(',', '\t')
    print(row, end='')
```

```
X       Y       month   day     FFMC    DMC     DC      ISI     temp    RH      wind
rain    area
7       5       mar     fri     86.2    26.2    94.3    5.1     8.2     51      6.7
0.0     0.0
7       4       oct     tue     90.6    35.4    669.1   6.7     18.0    33      0.9
0.0     0.0
7       4       oct     sat     90.6    43.7    686.9   6.7     14.6    33      1.3
0.0     0.0
8       6       mar     fri     91.7    33.3    77.5    9.0     8.3     97      4.0
0.2     0.0
8       6       mar     sun     89.3    51.3    102.2   9.6     11.4    99      1.8
0.0     0.0
8       6       aug     sun     92.3    85.3    488.0   14.7    22.2    29      5.4
0.0     0.0
8       6       aug     mon     92.3    88.9    495.6   8.5     24.1    27      3.1
0.0     0.0
8       6       aug     mon     91.5    145.4   608.2   10.7    8.0     86      2.2
0.0     0.0
8       6       sep     tue     91.0    129.5   692.6   7.0     13.1    63      5.4
0.0     0.0
```

## Convert CSV into Datafame-like Data Structure

We use a helper function to convert the data into the form of a dataframe-like dictionary.

That is, we convert a list of rows into a dictionary of columns, each cast to the appropriate data type.

Later, we will use Pandas and R dataframes to do this work.

First, we define the data types by inspecting the data and creating a dictionary of lambda functions to do our casting.

In [5]:
```
dtypes = ['i', 'i', 's', 's', 'f', 'f', 'f', 'f', 'f', 'i', 'f', 'f', 'f']
# dtypes = list("iissffffifff") # We could have done it this way, too

caster = {
    'i': lambda x: int(x),
    's': lambda x: str(x),
    'f': lambda x: float(x)
}
```

Next, we grab the column names from the first row or list.

Note that `.strip()` is a string function that removes extra whitespace from before and after a string.

In [6]:
```
cols = data_file[0].strip().split(',')
cols
```

Out[6]:
```
['X',
 'Y',
 'month',
 'day',
 'FFMC',
 'DMC',
 'DC',
 'ISI',
 'temp',
 'RH',
 'wind',
 'rain',
 'area']
```

Finally, we iterate through the list of rows and flip them into a dictionary of columns.

The key of each dictionary element is the columns name, and the value is a list of values with a common data type.

In [7]:
```
# Get the rows, but not the first, and convert them into lists
rows = [line.strip().split(',') for line in data_file[1:]]

# Initialize the dataframe by defining a dictionary of lists, with each column name as
firedata = {col:[] for col in cols}

# Iterate through the rows and convert them to columns
for row in rows:
    for j, col in enumerate(row):
        firedata[cols[j]].append(caster[dtypes[j]](col))
```

Test to see if it worked ...

In [8]:
```
firedata['Y'][:5]
```

Out[8]:
```
[5, 4, 4, 6, 6]
```

# Working with spatial coordinates `X`, `Y`

For the first tasks, we grab the first two columns of our table, which define the spatial coordinates within the Monteshino park map.

In [9]:
```
X, Y = firedata['X'], firedata['Y']
```

In [10]:
```
X[:10], Y[:10]
```

Out[10]:
```
([7, 7, 7, 8, 8, 8, 8, 8, 8, 7], [5, 4, 4, 6, 6, 6, 6, 6, 6, 5])
```

# Task 1

(2 points)

Write a function called `coord_builder()` with these requirements:

- Takes two lists, X and Y, as inputs. X and Y must be of equal length.
- Returns a list of tuples `[(x1,y1), (x2,y2), ..., (xn,yn)]` where `(xi,yi)` are the ordered pairs from X and Y.
- Uses the `zip()` function to create the returned list.
- Use a list comprehension to actually build the returned list.
- Contains a docstring with short description of the function.

In [46]:
```
# CODE HERE
def coord_builder(list1, list2):
    '''
    This function takes two lists of equal length and returns a list of
    tuples for the ordered pairs from list1 and list2.

    INPUTS
    list1       list of ints
    list2       list of ints

    OUTPUT
    list3       list of tuples
    '''
    if len(list1)==len(list2):
        list3 = [(xi, yi) for xi, yi in zip(list1, list2)]
        print(list3)
    else:
        print("ERROR: Please use lists of equal length!")
    return
```

# Task 2

(1 PT)

Call your `coord_builder()` function, passing in `X` and `Y`.

Then print the first ten tuples.

In [12]:
```
# CODE HERE
coord_builder(X, Y)[:5]
```

Out[12]:
```
[(7, 5), (7, 4), (7, 4), (8, 6), (8, 6)]
```

# Working with AREA

Next, we work the area column of our data.

In [13]:
```
area = firedata['area']
```

In [14]:
```
area[-10:]
```

Out[14]:
```
[0.0, 0.0, 2.17, 0.43, 0.0, 6.44, 54.29, 11.16, 0.0, 0.0]
```

## Task 3

(1 PT)

Write code to print the minimum area and maximum area in a tuple `(min_value, max_value)`.

Save `min_value` and `max_value` as floats.

In [15]:
```
# CODE HERE
def min_max(list1):
    '''
    This function takes a list of ints or floats and returns the minimum
    value in the list and the maximum value in the list as a tuple.

    INPUT
    list1    list of ints or floats

    OUTPUT
    mm       tuple as floats
    '''
    mm = (float(min(list1)), float(max(list1)))
    return mm
```

## Task 4

(2 PTS)

Write a lambda function that applies the following function to $x$:

> $log_{10}(1 + x)$

Return the rounded value to $2$ decimals.

Assign the function to the variable `mylog10`.

Then call the lambda function on `area` and print the last 10 values.

Hints:

- Use the `log10` function from Python's `math` module. You'll need to import it.
- Use a list comprehension to make the lambda function a one-liner.
- To get the last members of a list, used negative offset slicing. See the Python documentation on lists for a refresher on slicing.

In [47]:
```
# CODE HERE
import math
log10 = lambda x: round(math.log10(1+x), 2)
```

In [48]:
```
[log10(ea_num) for ea_num in area][-10:]
```

Out[48]:
```
[0.0, 0.0, 0.5, 0.16, 0.0, 0.87, 1.74, 1.08, 0.0, 0.0]
```

# Working with MONTH

The month column contains months of the year in abbreviated form — `jan` to `dec`.

In [18]:
```
month = firedata['month']
```

In [19]:
```
month[:10]
```

Out[19]:
```
['mar', 'oct', 'oct', 'mar', 'mar', 'aug', 'aug', 'aug', 'sep', 'sep']
```

## Task 5

(1 PT)

Create a function called `get_uniques()` that extracts the unique values from a list.

- Do not use `set()` but instead use a **dictionary comprehension** to capture the unique names.
- Hint: They keys in a dictionary are unique.
- Hint: You do not need to count how many times a name appears in the source list.

Then function should optionally return the list as sorted in ascending order.

Then apply it to the `month` column of our data with sorting turned on.

Then print the unique months.

In [24]:
```
# CODE HERE
def get_uniques(list1):
```

```
    '''
    This function takes a list of str and returns a list of all unique
    values in alphabetical order.

    INPUT
    list1         list of str

    OUTPUT
    uniques       list of str
    '''
    uniques = sorted(list(dict.fromkeys(list1)))
    return uniques
```

In [29]:
```
unique_months = get_uniques(month)
print(unique_months)
```

```
['apr', 'aug', 'dec', 'feb', 'jan', 'jul', 'jun', 'mar', 'may', 'nov', 'oct', 'sep']
```

## Task 6

(1 PT)

Write a lambda function called `get_month_for_letter` that uses a list comprehension to select all months starting with a given letter from the list of unique month names you just crreated.

The function should assume that the list of unique month names exists in the global context.

The returned list should contain uppercase strings.

Run and print the result with `a` as the paramter.

In [42]:
```
# CODE HERE
get_month_for_letter = lambda a_letter: [month.upper() for month in unique_months
                                        if a_letter.upper()==month[0].upper()]
```

In [43]:
```
get_month_for_letter("a")
```

Out[43]:
```
['APR', 'AUG']
```

# Working with DMC

DMC - DMC index from the FWI system: 1.1 to 291.3

In [44]:
```
dmc = firedata['DMC']
```

In [45]:
```
dmc[:10]
```

Out[45]:
```
[26.2, 35.4, 43.7, 33.3, 51.3, 85.3, 88.9, 145.4, 129.5, 88.0]
```

## Task 7

(2 PTS)

Write a function called `bandpass_filter()` with these requirements:

- Takes three inputs:
  - A list of numbers `num_list`.
  - An integer serving as a lower bound `lower_bound`.
  - An integer serving as an upper bound `upper_bound`.
- Returns a new array containing only the values from the original array which are greater than `lower_bound` and less than `upper_bound`.

In [49]:
```
# CODE HERE
def bandpass_filter(num_list, lower_bound, upper_bound):
    '''
    This function takes in a list of ints or floats and checks to see if each number i
    within a specified bounds. Returns a list of numbers that are within the bounds.

    INPUT
    num_list       a list of ints and/or floats
    lower_bound    int or float
    upper_bound    int or float

    OUTPUT
    within_bounds  a list of ints or floats
    '''
    within_bounds = [num for num in num_list if (num>lower_bound and num<upper_bound)]
    return within_bounds
```

# Task 8

(1 PT)

Call `bandpass_filter()` passing `dmc` as the list, with `lower_bound=25` and `upper_bound=35`.

Then print the result.

In [50]:
```
# CODE HERE
bandpass_filter(dmc, 25, 35)
```

Out[50]:
```
[26.2,
 33.3,
 32.8,
 27.9,
 27.4,
 25.7,
 33.3,
 33.3,
 30.7,
 33.3,
 25.7,
 25.7,
 25.7,
 32.8,
 27.2,
 27.8,
 26.4,
 25.4,
 25.4,
 25.4,
 25.4,
 26.7,
 25.4,
 27.5,
 28.0,
 25.4]
```

# Working with FFMC

FFMC - FFMC index from the FWI system: 18.7 to 96.20

In [51]:
```
ffmc = firedata['FFMC']
```

In [52]:
```
ffmc[:10]
```

Out[52]:
```
[86.2, 90.6, 90.6, 91.7, 89.3, 92.3, 92.3, 91.5, 91.0, 92.5]
```

## Task 9

(2 PTS)

Write a lambda function `get_mean` that computes the mean $\mu$ of a list of numbers.

- The mean is jus the sum of a list of numeric values divided by the length of that list.

Write another lambda function `get_ssd` that computes the squared deviation of a number.

- The function takes two arguments, a number from a given list and the mean of the numbers in that list.
- The function is meant to be used in a for-loop that iterates through a list.
- The squared deviation of a list element $x_i$ is $(x_i - \mu)^2$.

Then write `get_sum_sq_err()` with these requirements:

- Takes a numeric list as input.
- Computes the mean $\mu$ of the list using `get_mean`.
- Computes the sum of squared deviations for the list using a list comprehension that applies `get_ssd`.
- Returns the sum of squared deviations.

In [53]:
```
# CODE HERE
get_mean = lambda list1: sum(list1)/len(list1)
```

In [54]:
```
get_ssd = lambda num1, mean_list: (num1-mean_list)**2
```

In [55]:
```
def get_sum_sq_err(list1):
    the_mean = get_mean(list1)
    the_ssd = [get_ssd(num, the_mean) for num in list1]
    return sum(the_ssd)
```

## Task 10

(1 PT)

Call `sum_sq_err()` passing `ffmc` as the list and print the result.

In [56]:
```
# CODE HERE
get_sum_sq_err(ffmc)
```

Out[56]:
```
15723.357872340412
```

In [ ]:
```
```