User Documentation

Fundamentals of Programming (COMP 1007)

Connor Kuljis 19459138, Submission Date: 05/06/2020

Table of Contents

Quick Start - How to use the programs

Investigation 1 - Power Usage

```
python3 usage.py

python3 usagecomparison.py
```

Investigation 2 - Power Model

```
jupyter notebook

open Inves2.ipynb
```

Investigation 3 - Power Simulation

```
housesimulation.py: takes 6 command line arguments to construct a
                    suburb with appropriate name and amount of houses

python3 housesimulation.py [Suburb Name] [Postcode]
                           [Mansion Qty] [Family Qty] [Flat Qty] [Studio Qty]

eg: python3 housesimulation.py Dianella 6059 2 8 5 3
```

Parameter Sweep

```
bash ./sweepHouses.sh [Suburb Name] [Postcode]
                      [Mansions start] [Mansions stop] [Mansions step]
                      [Family start] [Family stop] [Family step]
                      [Flat start] [Flat stop] [Flat step]
                      [Studio start] [Studio stop] [Studio step]

eg: bash ./sweepHouses5.sh Dianella 6059 1 2 1 1 2 1 1 2 1 1 2 1
```

Overview

Investigation 1: Power Usage

The purpose of the first investigation is to explore the results of a power meter at a residence, by recording the data every 24 hours and documenting the results in a csv file. The data will also be compared to the daily weather maximums for each day, also stored in csv format.

The program can parse in both csv files and construct a pandas data frame from it. Once the data is available, a list of date objects are extracted by using the `pandas.to_datetime()` method, with appropriate formatting. Each column in the dataframes are separated into list variables, and sliced to match the same length.

Currently our powerusage only shows the cumulative usage, and not the difference between each day. To solve this, the pandas function `.diff()` is used to calculate the difference of each element compared with the last. This is stored in a new dataframe ready for visualisation.

`usage.py` creates 3 subplots in the figure. It presents a broad view of the daily temperature, daily power usage and cumulative power usage.

`usagecomparison.py` further explores this data and overlays the temperature against power usage, combining two seperate y axes on a shared x axis. This also provides some statistical data by summarizing the tendency, shape and dispersion of values. Results are stored by piping the output to a text file `python3 usagecomparison.py >> results.txt`

Investigation 2: Power Model

This investigation is presented in a jupyter notebook.

The purpose of investigation two is to model appliance usage. An appliance CSV file can be created and analysed to reflect on the results of investigation one.

By reading in the csv file and performing some list operations, a chart of what the daily usage, hour by hour looks like.

According to the power model file format, the third element in each row is a string of 24 numbers, representing the state of each appliance at that hour. eg 1 is maximum usage and 0 is off. It could also be in decimal format. To create a power model, every value in this string is converted to a floating point number and multiplied by the wattage for that appliances (which is the second element in each row). An example is: [2000:0,0.5...] becomes [2000:0,1000]. By using a for loop each row is sliced, performed on and appended to a new list of values which can be plotted.

Investiagtion 3: Power Simulation

The goal of this investigation is to simulate the daily power usage of a suburb. The investigation has 2 class model files; house and suburb. The suburb is constructed by appending house objects to a class field of the current suburb object.A suburb can be simulated the driver program `housesimulation.py` in which the number of houses are passed in thorugh command line prompts.

The house objects includes 4 sub-classes. Mansion, Family, Flat and Studio. The house.py class model file uses the principle of object orientation; inheritence to re-use fields and methods from the base class. Each subclass has its own appliance csv file eg: a list of appliances.

By adapting programs from the previous investigations, some visualisations can be made and information about power usage can be drawn.

This investigation also includes a bash script to perform a parameter sweep to simulate various different layouts of house types.

File Structure Overview

Each investigation is separated into individual directories; **Inves1, Inves2** and **Inves3** The purpose of each file follows:

Inves1

```
* Figure_1.png
* Figure_2.png
* usagecomparison.py  : overlays power usage and weather in a plot using matplotlib
* usage.py    : reads may_max.csv and powerusage.csv, and explores data using matplotlib
* may_max.csv         : weather model file
* powerusage.csv      : power usage file
* usage_analysis.txt  : saving results of usagecomparison.py in a text file
```

Inves2

```
* Inves2.ipynb        : jupyter notebook visually exploring daily usage over time
* ben.csv             : sample files
* jia.csv             : sample files
* katrina.csv         : sample files
* myhouse.csv         : sample files
* powermodel.csv      : sample files
* valerie.csv         : sample files
```

Inves3

```
* house.py            : house model class
```

```
* suburb.py           : suburb model class
* housesimulation.py  : run a simulation of a suburb from command line arguments
* sweepHouses.sh      : parameter sweep bash script
* family.csv          : appliance file for family house object
* flat.csv            : appliance file for flat house object
* mansion.csv         : appliance file for mansion house object
* studio.csv          : appliance file for studio house object
* testscript.txt      : examples how to run the parameter sweep
```

Discussion

Investigation 1

- Creating dateobjects using pandas, note the different formats. example of lines from csv

`3 5 2020,187937 (example powerusage.csv)`
`1-5-2020,24.1 (example from may_max.csv)`

Converting to date time objects allows for easy plotting of a time-series. `weather_df['date'] = pd.to_datetime(weather_df['date'], format='%d-%m-%Y') power_df['date'] = pd.to_datetime(power_df['date'], format='%d %m %Y')`

- To find the difference of power usage of each day

Creating a new dataframe, which is how much power is used each day, instead of cumulative using the `difference_df = power_df['usage'].diff()`

```
RESULTS
0     187937
1     187968
2     187993
3     188032
4     188100
5     188117
6     188188
7     188219
8     188261
9     188303
10    188349
11    188381
12    188427
13    188467
14    188508
15    188532
16    188621
17    188691
```

```
18    188760
19    188828

.. to become

0      NaN
1     31.0
2     25.0
3     39.0
4     68.0
5     17.0
6     71.0
7     31.0
8     42.0
9     42.0
10    46.0
11    32.0
12    46.0
13    40.0
14    41.0
15    24.0
16    89.0
17    70.0
18    69.0
19    68.0

notice it is one element shorter.
```

Investigation 2

Data uses the delimiter ":", giving three parameters, appliance name, wattage and power usage.

Power usage becomes a string further delimited on ','

This is completed by a list comprehension, spliting the long string of numbers into floats and multiplying by current wattage **usagelist = [float(i) * appliancewattage for i in p_usage.split(',')]**

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import csv

appliance = []
wattage = []
```

```
usage = []
times=range(1,25)

# importing the csv file
with open('myhouse.csv', newline='') as csvfile:
    modelreader = csv.reader(csvfile, delimiter=':') # seperate on ':'
    for row in modelreader:
        # the format follows; ['appliance name','wattage','power usage']
        # currently each row is stored as a String
        # going to split each element into separate lists and convert datatype if needed

        appliancenames = row[0]
        appliancewattage = float(row[1])
        p_usage = row[2]

        # appending appliance names
        appliance.append(appliancenames)

        # appending appliance wattage
        wattage.append(appliancewattage)

        # list comprehension splitting the long string of numbers into floats and multiplyin
        usagelist = [float(i) * appliancewattage for i in p_usage.split(',')]
        # appending usage
        usage.append(usagelist)

plt.figure(1)
plt.title("Stacked Daily Power Usage")
plt.xlabel("24hr Time (Hours)")
plt.ylabel("Power Usage (Watts)")
plt.stackplot(times, usage, labels=appliance)
plt.legend(loc='upper right')
plt.show()

(Example) Fridge:200:1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1

... to become

appliance  = [Fridge]
wattage    = [200] (convert to float)
usage      = [200,200,200,200,200,200,200,200,200,200,200,200,200,200,200,200,200,200,200,20
```

Investigation 3

- House base class attributes

```
class House():
    def __init__(self, address, housetype, postcode, numResidents, numBeds, numBaths, applia
        self.housetype = housetype
        self.address = address
        self.postcode = postcode
        self.numResidents = numResidents
        self.numBeds = numBeds
        self.numBaths = numBaths
        self.appliances = self.readAppliance(appliancefile) # list of appliances include nam
```

Includes the functions: + readAppliance(filename) + getApplianceNames() + getUsage() + getNumResidents() + calcTotalUsage() + calcDailyUsage() + calcUsageAtTime(hour) + printit()

By inheritance, sub classes can be created: eg Mansion. Here you can see that default attributes can be assigned and passed to the base class, allowing the above functions to be reused.

```
class Mansion(House):
    housetype = "Mansion"
    numResidents = 3
    numBeds = 8
    numBaths = 4
    appliancefile = "mansion.csv"

    def __init__(self, address, postcode):
        super().__init__(address, self.housetype, postcode, self.numResidents, self.numBeds,
```

- Suburb class uses an if - elif statement to check the type of house, and construct that subclass accordingly. This allows our driver code housesimulation.py to automate and control suburb creation. The suburb can then tally the total usage of each house.

```
class Suburb():
    def newHouse(self, houseType, address):
        temp = None
        if houseType == 'Mansion':
            temp = Mansion(address, self.postcode)
        elif houseType == "Flat":
            temp = Flat(address, self.postcode)
        elif houseType == "Family":
            temp = Family(address, self.postcode)
        elif houseType == "Studio":
            temp = Studio(address, self.postcode)
        else:
            print("Error processing house -> '" + houseType + "' (please double check values
        if temp:
```

```
            self.houses.append(temp)
            print("Added " + houseType + ": " + address + " to Suburb - " + self.name + " ('
```

- Extra functionality. Parameter sweep explanation.

```
sName=$1
sPostcode=$2

# mansions
lowMM=$3
hiMM=$4
stepMM=$5

# family
lowFF=$6
hiFF=$7
stepFF=$8

# flats
lowFL=$9
hiFL=${10}
stepFL=${11}

# studios
lowST=${12}
hiST=${13}
stepST=${14}
```

There are four house types. We can run paramater sweeps for each house type with the arguments [start], [stop], [step]; giving 4 * 3 = 12 arguments + 2 for suburb name and postcode.