CODE REVIEW

Criteria of evaluation

- Naming - Convention used and choosing appropriate variables
- Commented code - Are the comments sufficient and clear?
- Simplicity and clarity - Levels of nesting ,Lines per function
- Ease of reuse - Portable? Modular?

Naming

- Consistent Convention

Two words in a compound variable name always separated with an _

```
def getStateFilesAsDict(all_dirs,format_of_file):
    all_files = [sorted(glob.glob(x+"/"+'*.'+format_of_file)) for x in all_dirs]
    #print(rainfall_files[0][0].split('\\'))
    state_files = defaultdict(lambda:[])
    for x in all_files:
        for y in x:
            res  = y.split('/')
            state_files[res[1]].append(y)
    return state_files

def getAverageAndAppend(city_file_path,name_of_new_column):
    data = pd.read_csv(city_file_path)
    #fill_zero = [0] * len(data['Year'])
    if name_of_new_column not in data.columns:
        avg = list()
        for i in (range(len(data['Year']))):
            sum_n = 0
            for j in range(1,13):
                sum_n += data.loc[i][j]
            avg.append(sum_n/12)
        data[name_of_new_column] = pd.Series(avg, index=data.index)
        #print(data.head(10))
```

Naming

- Appropriate variable names

While variables are named appropriately in most places, in some parts of the code it is not easy to figure out what the names stand for without reading comments

```
seasons={"Rabi":["Oct","Nov","Dec","Jan","Feb","Mar"],"Kharif":["Jul","Aug","Sep","Oct"],"Whole Year":["Oct","Nov","Dec","Jan","Feb","Ma

#prediction using linear regression
def predict(params,year):
        slope, intercept, r_value, p_value, std_err = stats.linregress(params)
        return (slope*year+intercept)

#this predicts the weather type for the years mentioned in the list
def get_params_wrapper_predict(params):
        slope, intercept, r_value, p_value, std_err = stats.linregress(params)
        for i in ["2002","2003","2004","2005","2006","2007","2008","2009"]:
                params.append([int(i),(slope*int(i)+intercept)])
        return params

#wrapper function which populates the list params by reading a specific CSV file
def get_params_wrapper(param_type,state,district,season):       #param_type=Rainfall/temparature
        params=[]
        with open("CSV/"+param_type+"/"+state+"/"+district+".csv") as csvfile:
                reader=csv.DictReader(csvfile)
                s=0;
```

Comments in code

- Sufficient and clear

The comments clearly state what each function is doing and they are sufficient to understand the summary of each part of the code

```
seasons={"Rabi":["Oct","Nov","Dec","Jan","Feb","Mar"],"Kharif":["Jul","Aug","Sep","Oct"],"Whole Year":["Oct","Nov","Dec","Jan","Feb","Ma

#prediction using linear regression
def predict(params,year):
        slope, intercept, r_value, p_value, std_err = stats.linregress(params)
        return (slope*year+intercept)

#this predicts the weather type for the years mentioned in the list
def get_params_wrapper_predict(params):
        slope, intercept, r_value, p_value, std_err = stats.linregress(params)
        for i in ["2002","2003","2004","2005","2006","2007","2008","2009"]:
                params.append([int(i),(slope*int(i)+intercept)])
        return params

#wrapper function which populates the list params by reading a specific CSV file
def get_params_wrapper(param_type,state,district,season):       #param_type=Rainfall/temparature
        params=[]
        with open("CSV/"+param_type+"/"+state+"/"+district+".csv") as csvfile:
                reader=csv.DictReader(csvfile)
                s=0;
```

Simplicity and clarity

- Levels of nesting

There aren't multiple levels of nesting making the code easy to understand

```
if(len(month_season) ==0):
    expected = calculate_avg_per_year_and_plot(every_city_file_path,"avg_rainfall",year)
    print('avg rainfall for Year %d: %f cm for region : %s , state : %s' % (year, expected,every_city,every_state))
    return expected
elif( month_season[0] == "month"):
    mydict = {'January':"Jan","Febraury":"Feb","March":'Mar',"April":"Apr","May":"May","June":"June","July":"July","August":"Aug","September":"Sept","Oc
    month = mydict[month_season[1]]
    expected = calculate_per_month_and_plot(every_city_file_path,month,year)
    print('avg rainfall in the month of %s for Year %d: %f cm for region : %s , state : %s' % (month,year, expected,every_city,every_state))
    return expected
elif(month_season[0] == "season"):
    season_name = "avg_"+month_season[1]
    expected = calculate_avg_per_season_and_plot(every_city_file_path,season_name,year)
    print('avg rainfall in %s season for Year %d: %f cm for region : %s , state : %s' % (month_season[1],year, expected,every_city,every_state))
    return expected
else:
    raise "Invalid Argument Exception"
```

Simplicity and clarity

- Lines per function

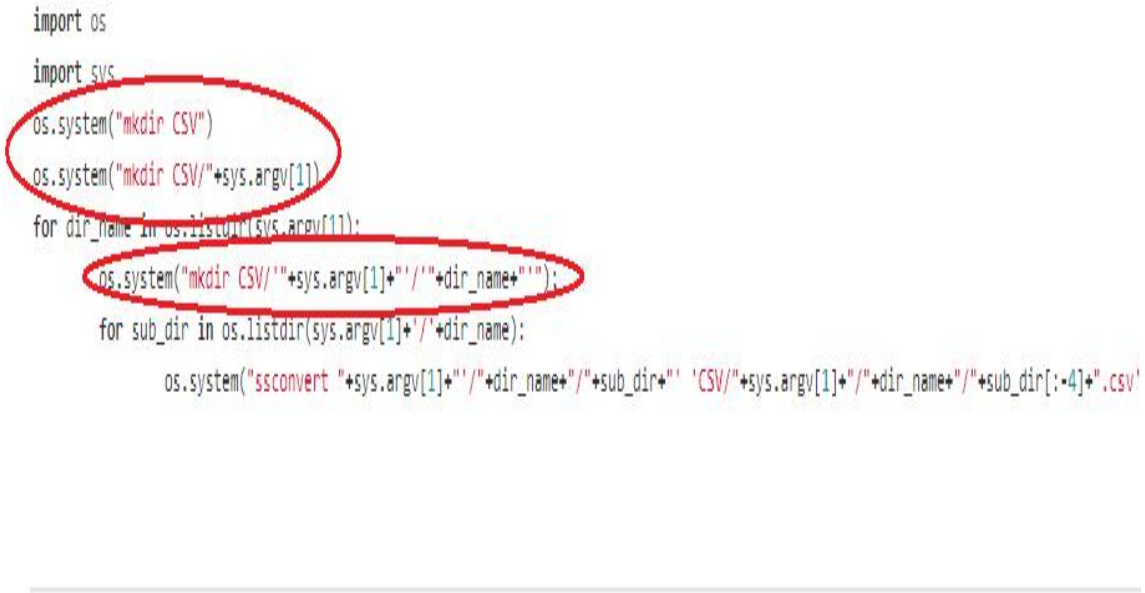
There are not too many lines in every function making the code easy to manage

```
def getDataForMonth(city_file_path,month):
    data = pd.read_csv(city_file_path)
    return pd.DataFrame(list(map(list, zip(data['Year'],data[month]))),columns = ['Year',month])

def calculate_avg_per_year_and_plot(every_city_file_path,new_attribute,year):
    df = getAverageAndAppend(every_city_file_path,new_attribute)
    series = pd.Series(data = df[new_attribute].values,index =df['Year'])
    expected,to_plot = predict_for_year(series,year)
    plot_predicted(to_plot)
    return expected

def calculate_avg_per_season_and_plot_(every_city_file_path,wanted_,df,year):
    series = pd.Series(data = df[wanted_].values,index =df['Year'] )
    expected,to_plot = predict_for_year(series,year)
    plot_predicted(to_plot)
    return expected

def calculate_avg_per_season_and_plot(every_city_file_path,wanted_,year):
    df = getAverageBySeasonsAndAppend(every_city_file_path,"avg_summer","avg_rainy","avg_winter"
    return calculate_avg_per_season_and_plot_(every_city_file_path,wanted_,df,year)
```

Ease of reuse

- Portablity

The code is not portable and seems to only work on linux based systems. Code can use system independent functions such as os.mkdir(path) rather than os.system(mkdir path)



Portability

- Modular

The code is modular - not evrything is dumped in one file and every file is split up into functions to perform different tasks.

```
#this is the module function which client can access for yield
def get_yield(state,district,crop,season):
        get_yield_wrapper();           # this ensures that dictionary 'd' is populated
        p_a=[]
        for i in d[state.upper()][district.upper()][crop][season]:
                p_a.append([i[0],(int(i[2])/int(i[1]))])
        return p_a

#this is the module function which client can access
def get_rainfall(state,district,season):
        return get_params_wrapper("Rainfall",state,district,season)

#this is the module function which client can access
def get_temperature(state,district,season):
        return get_params_wrapper("Temperature",state,district,season)
```