EPPS/GISC 4317: Python programming for social science

Lab4: Functions & Classes

Download the streets.csv & parcels.csv files to your personal directory.

Part 1 – Creating Functions

In this exercise, you will create a custom function that can be called from within the same script or from another script.

1. Start PyCharm and create a new file in your Lab 4 directory and save it as **mylist.py**
1. Enter the following code, replacing "<mylab4directory>" with your actual lab 4 directory:

```
streetsCSV = r"<yourlab4directory>\streets.csv"

with open(streetsCSV, 'r') as csvFile:
   reader = csv.reader(csvFile)
   print(reader)
```

2. Running the script prints the Python reference information for each field object in the streets csv file. The code does not print any field names. To print the names of the field objects, you can read the first row. A for loop can be used to iterate over the list of fields.

3. Modify the code as follows:

```
with open(streetsCSV, 'r') as csvFile:
   reader = csv.reader(csvFile)
   print(reader)
   headers = reader.__next__()
#Create a list to save headers
headerlist = []
for header in headers:
   headerlist.append(header)
#Print the headers list
print(headerlist)
```

4. Run the script. Running the script should now print the list of field names, as follows:

```
['OBJECTID', 'PRE_TYPE', 'ST_NAME', 'STREET_NAM',
'STREET_TYP', 'SUF_DIR', 'FULLNAME', 'LOW', 'HIGH', ...
'F_ZLEV', 'T_ZLEV', 'FEET', 'Shape_len']
```

Once you have the script to create a list of header names, you may want to use it again. You can do this by creating a custom function.

5. Modify the code as follows:

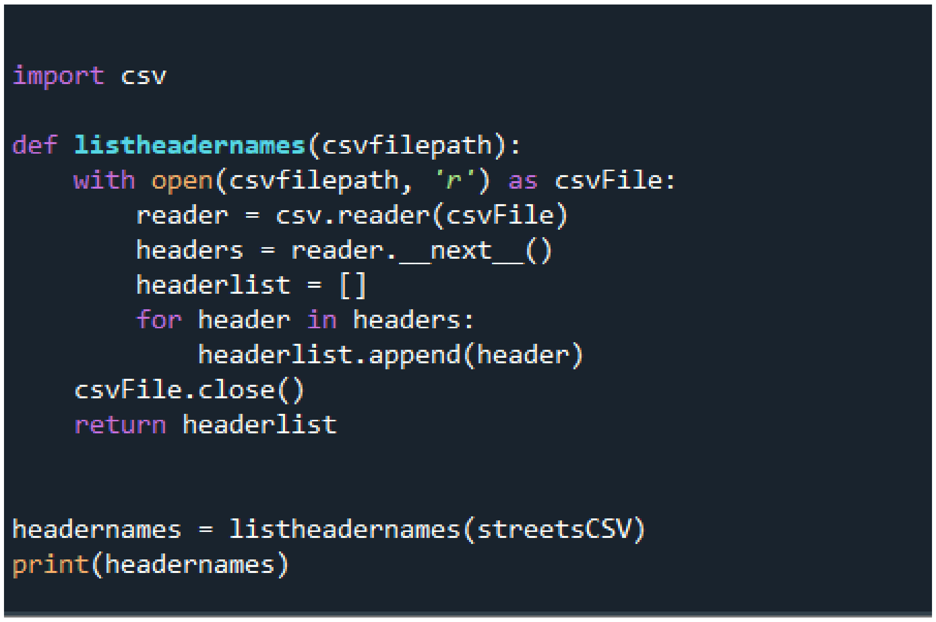
```
import csv
```

```
def listheadernames(csvfilepath):
    with open(csvfilepath, 'r') as csvFile:
        reader = csv.reader(csvFile)
        headers = reader.__next__()
        headerlist = []
        for header in headers:
            headerlist.append(header)
    return headerlist
```

The block of code that creates the list of headers is now defined as a function called `listheadernames`. This function can now be called, for example, from within the same script.

6. Add the following lines of code to the very bottom, outside the function:

```
headernames = listheadernames(streetsCSV)
print(headernames)
```



```
import csv

def listheadernames(csvfilepath):
    with open(csvfilepath, 'r') as csvFile:
        reader = csv.reader(csvFile)
        headers = reader.__next__()
        headerlist = []
        for header in headers:
            headerlist.append(header)
    csvFile.close()
    return headerlist


headernames = listheadernames(streetsCSV)
print(headernames)
```

7. Run the script. Running the script prints the list of header names to the Interactive Window.

   Initially, it does not appear to do anything different from the earlier version of the script that did not define a custom function. However, once the function is created, it can also be called from another script.

Part 2 – Call functions from other scripts

1. Create a new Python script and save as myscript.py in your lab 10 folder.

2. Enter the following code, replacing "<mylab4directory>" with your actual lab 4 directory

   ```
   import csv
   import mylist

   streetsCSV = r"<yourlab4directory>\streets.csv"
   headernames = mylist.listheadernames(streetsCSV)
   print(headernames)
   ```

   This script imports the mylist module, calls the `listheadernames` function, and passes the name of a csv file path as an argument to this function.

3. Run the script. The result is that the list of headers is printed out twice to the Interactive Window.

   What happened? When the myscript.py script called the `listheadernames` function, it ran the mylist.py script. This script creates and then prints the list of header names. The function also returns the list of header names, and it is printed by the myscript.py script. To avoid double printing, some additional code is needed.

4. Open the mylist.py script and add the following line of code just before the line that starts with " headernames...":

   ```
   if __name__ == "__main__":
   ```

   (Note: There are two underscores in each spot.)

5. Indent the last two lines of code.

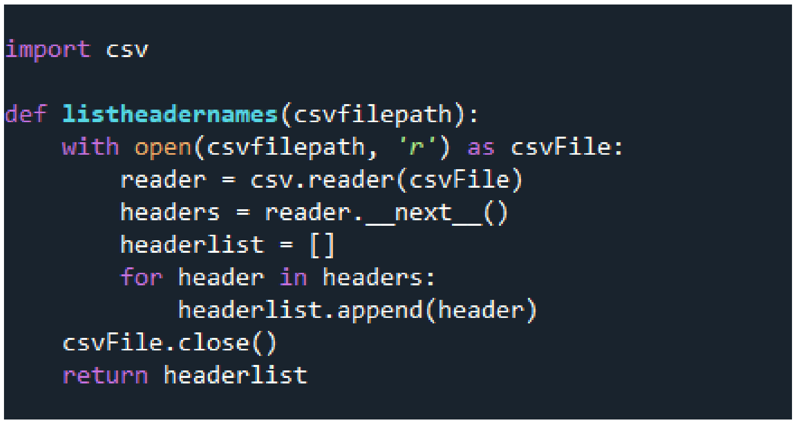
6. Open myscript.py.

7. Without making any changes, run the script. It should now only print the fields once.

   When the myscript.py script runs and calls the `listheadernames` function, the `if __name__ == "__main__":` statement in the mylist.py script ensures that the next block of code is run only if the mylist.py script is run by itself.

   The block of code following the `if __name__ == "__main__":` statement can be considered a "test." When the mylist.py script is run by itself, this test code allows you to check whether the custom function works correctly. However, it may not be necessary to keep this code if the script is being used to store a custom function that will only be called from other scripts.

8. Reopen mylist.py and modify it by removing the last three lines of code. It should look like this:



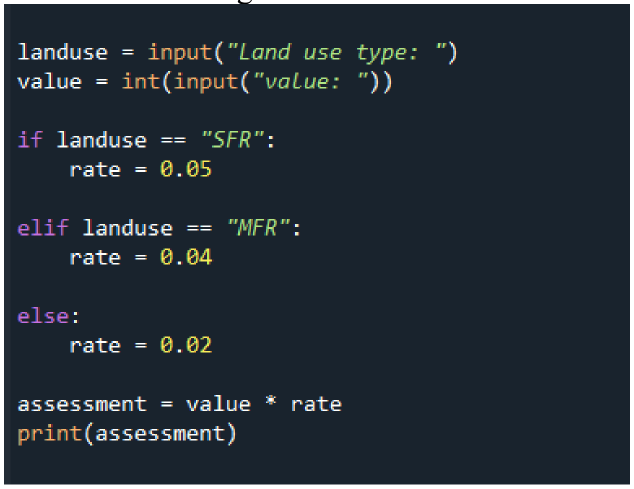
```
import csv

def listheadernames(csvfilepath):
    with open(csvfilepath, 'r') as csvFile:
        reader = csv.reader(csvFile)
        headers = reader.__next__()
        headerlist = []
        for header in headers:
            headerlist.append(header)
    csvFile.close()
    return headerlist
```

Part 3 – Working with Classes

Classes allow you to group functions and variables together. Once created, classes make it possible to create objects that have specific properties as defined by these functions and variables. Next, you will first consider a script to calculate property taxes, and then you will create a class to make this calculation more versatile.

1. In PyCharm, create a new Python script and save as **assessment.py** to your lab 4 folder.

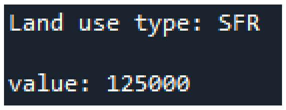
2. Enter the following code:



```
landuse = input("Land use type: ")
value = int(input("value: "))

if landuse == "SFR":
    rate = 0.05

elif landuse == "MFR":
    rate = 0.04

else:
    rate = 0.02

assessment = value * rate
print(assessment)
```

This script calculates the property tax assessment based on variables for land use and the property value.

3. Run the script and enter the arguments as shown below:



```
Land use type: SFR

value: 125000
```

Running the script prints the result of 6250.0 to the Interactive Window.

To automate this calculation for many different entries, you would read the values from a file and iterate over these values. Then you would have a few options to carry out the tax calculation. First, you can place the code within the iteration — that is, within the `for` loop or the `while` loop. Second, you can create a custom function in a separate script that does the calculation; when the function is called, the necessary arguments are passed, and the function returns a value. Third, you can create a class that contains the calculation as a method.

4. In PyCharm, create a new Python script and save as tax.py to your lab 10 folder.

5. Enter the following code:

```
def taxcalc(landuse, value):
   if landuse == "SFR":
      rate = 0.05
   elif landuse == "MFR":
      rate = 0.04
   else:
      rate = 0.02
   assessment = value * rate
   return assessment
```

6. Create a new Python script and save as parcelCalc.py to your lab 04 folder.

7. Enter the following code:

```
import tax
mytax = tax.taxcalc("SFR", 125000)
print(mytax)
```

8. Run the script. Running the script prints the result of 6250.0 to the Interactive Window.

9. Create a new Python script and save as parcelclass.py in your lab 4 folder.

10. Enter the following code:

```
class Parcel:
   def __init__(self, landuse, value):
      self.landuse = landuse
      self.value = value
   def assessment(self):
      if self.landuse == "SFR":
            rate = 0.05
        elif self.landuse == "MFR":
            rate = 0.04
        else:
            rate = 0.02
        assessment = self.value * rate
        return assessment
```

11. Create a new Python script and save as parcelTax.py in your lab 04 folder.

12. Enter the following code:

```
import parcelclass
myparcel = parcelclass.Parcel("SFR", 125000)
print("Land use: ", myparcel.landuse)
print("Value: ", myparcel.value)
mytax = myparcel.assessment()
print("Tax assessment: ", mytax)
```

13. Running the script prints the following tax information to the Interactive Window:

```
Land use:  SFR
Value:  125000
Tax assessment:  6250.0
```

    Although the use of the class accomplishes the same calculation as the custom function, a class is more versatile because it allows you to combine properties and functions. It would be relatively easy, for example, to expand the class with additional methods for different calculations, which could all be part of the same class.

HOMEWORK

(a) Custom Function

Create a custom function called `countnullfields` that counts the number of empty fields in a header of a csv file. If there is not any empty header, then return zero. Csv file is going to be the parameter of the function in a script (named as mycount.py), that you call from another script (named as callingscript.py). You may use the streets csv file in the Lab 4 folder for testing purposes.

Deliverable: 2 separated Python files (mycount.py & callingscript.py)

(b) Classes

You are given a csv file called parcels.csv located in Lab 4 folder that contains the

following fields: FID, Landuse, and Value. Modify the parceltax.py script so that it determines the property tax for each parcel and stores these values in a list. You should use the class created in the parcelclass.py script above (remain the class unchanged). Print the values of the final list by FID.
Sample output:

```
FID 1    has a tax value of    19000.0
FID 2    has a tax value of    27000.0
FID 3    has a tax value of    9200.0
FID 4    has a tax value of    6250.0
FID 5    has a tax value of    13000.0
FID 6    has a tax value of    24800.0
FID 7    has a tax value of    6250.0
```

Deliverable: The modified *parceltax.py* file (if you modify the *parcelclass.py* file, include that as well)