hw04

September 14, 2022

1 hw04

1.1 Metadata

```
Name:    hw04
URL:     https://github.com/tslever/DS5100-2022-08-tsl2b/blob/main/lessons/M04/hw04.ipynb
Course:  DS 5100
Term:    Fall 2022 Online
Module:  M04: Functions
Author:  Tom Lever
Net ID:  tsl2b
Created: 13 September 2022
Updated: 14 September 2022
```

1.2 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 directory `DS5100-2022-08-0/lessons/M04_PythonFunctions/HW`.

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.

1.3 Setting Up

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

[1]:
```
import os
path: str = None
if os.name == 'posix':
    path = "~/Documents/DS5100-2022-08-0/lessons/M04_PythonFunctions/HW/
 ↪uci_mldb_forestfires.csv"
elif os.name == 'nt':
    path = os.environ['USERPROFILE'] + "/Documents/DS5100-2022-08-0/lessons/
 ↪M04_PythonFunctions/HW/uci_mldb_forestfires.csv"
list_of_file_lines: list[str] = None
with open(path, 'r') as file:
    list_of_file_lines = file.readlines()
```

```
print(type(list_of_file_lines))
```

```
<class 'list'>
```

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

[2]:
```
for file_line in list_of_file_lines[:11]:
    file_line = file_line.replace(',', '\t')
    print(file_line, 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
7	5	sep	sat	92.5	88.0	698.6	7.1	22.8	40
4.0	0.0	0.0
```

1.4 Convert Data Set into Dataframe-like Data Structure

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

That is, we convert a list of rows into a dictionary of columns, where each column is 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.

[3]:
```
from typing import Callable, Any
list_of_data_types: list[str] = ['i', 'i', 's', 's', 'f', 'f', 'f', 'f', 'f',␣
 ↪'i', 'f', 'f', 'f']
dictionary_of_data_types_and_casters: dict[str, Callable[Any, Any]] = {
    '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.

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

[4]:
```
list_of_column_names: list[str] = list_of_file_lines[0].strip().split(',')
print(list_of_column_names)
```

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

We iterate through the list of file lines and flip data values into a dictionary of column names and lists of column values.

We test to see if it worked.

[5]:
```
list_of_data_lines: list[str] = [line.strip().split(',') for line in␣
 ↪list_of_file_lines[1:]]
dictionary_of_column_names_and_lists_of_column_values: dict[str, list[Any]] =␣
 ↪{column_name: [] for column_name in list_of_column_names}
for data_line in list_of_data_lines:
    for column_index, column_value in enumerate(data_line):
        column_name: str = list_of_column_names[column_index]
        list_of_data_values: list[str] =␣
 ↪dictionary_of_column_names_and_lists_of_column_values[column_name]
        data_type: str = list_of_data_types[column_index]
        caster: Callable[Any, Any] =␣
 ↪dictionary_of_data_types_and_casters[data_type]
        cast_column_value: Any = caster(column_value)
        list_of_data_values.append(cast_column_value)

for key in dictionary_of_column_names_and_lists_of_column_values.keys():
    print(key + " | " +␣
 ↪str(dictionary_of_column_names_and_lists_of_column_values[key][0:10]))
```

```
X | [7, 7, 7, 8, 8, 8, 8, 8, 8, 7]
Y | [5, 4, 4, 6, 6, 6, 6, 6, 6, 5]
month | ['mar', 'oct', 'oct', 'mar', 'mar', 'aug', 'aug', 'aug', 'sep', 'sep']
day | ['fri', 'tue', 'sat', 'fri', 'sun', 'sun', 'mon', 'mon', 'tue', 'sat']
FFMC | [86.2, 90.6, 90.6, 91.7, 89.3, 92.3, 92.3, 91.5, 91.0, 92.5]
DMC | [26.2, 35.4, 43.7, 33.3, 51.3, 85.3, 88.9, 145.4, 129.5, 88.0]
DC | [94.3, 669.1, 686.9, 77.5, 102.2, 488.0, 495.6, 608.2, 692.6, 698.6]
ISI | [5.1, 6.7, 6.7, 9.0, 9.6, 14.7, 8.5, 10.7, 7.0, 7.1]
temp | [8.2, 18.0, 14.6, 8.3, 11.4, 22.2, 24.1, 8.0, 13.1, 22.8]
RH | [51, 33, 33, 97, 99, 29, 27, 86, 63, 40]
wind | [6.7, 0.9, 1.3, 4.0, 1.8, 5.4, 3.1, 2.2, 5.4, 4.0]
rain | [0.0, 0.0, 0.0, 0.2, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
area | [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

[6]:
```
dictionary_of_column_names_and_lists_of_column_values['Y'][:5]
```

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

1.5 Working with Spatial Coordinates `X` and `Y`

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

[7]:
```
X: list[int] = dictionary_of_column_names_and_lists_of_column_values['X']
Y: list[int] = dictionary_of_column_names_and_lists_of_column_values['Y']
print(X[:10])
print(Y[:10])
```

```
[7, 7, 7, 8, 8, 8, 8, 8, 8, 7]
[5, 4, 4, 6, 6, 6, 6, 6, 6, 5]
```

1.6 Tasks

1.6.1 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 $[(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)]$, where (x_i, y_i) are the ordered pairs from `X` and `Y`.
- Uses the `zip` function to create the returned list.
- Uses a list comprehension to actually build the returned list.
- Contains a docstring with a short description of the function.

[8]:
```
def coord_builder(X: list[int], Y: list[int]) -> list[tuple[int, int]]:
    '''
    PURPOSE: Given a list of X coordinates and a list of Y coordinates,␣
 ↪provides a list of (X, Y) coordinate pairs
    INPUTS:
        X: list of integers
        Y: list of real integers
    OUTPUTS:
        list_of_coordinate_pairs
    '''

    if (len(X) != len(Y)):
        raise AssertionError('Length of list X is not equal to length of list Y.
 ↪')
    zip_of_coordinate_pairs: zip = zip(X, Y)
```

```
    list_of_coordinate_pairs: list[tuple[int, int]] = [coordinate_pair for␣
↪coordinate_pair in zip_of_coordinate_pairs]
    return list_of_coordinate_pairs
```

1.6.2 Task 2

(1 point)

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

Then print the first ten tuples.

[9]:
```
list_of_coordinate_pairs: list[tuple[int, int]] = coord_builder(X, Y)
_ = [print(coordinate_pair) for coordinate_pair in list_of_coordinate_pairs[0:
↪10]]
```

```
(7, 5)
(7, 4)
(7, 4)
(8, 6)
(8, 6)
(8, 6)
(8, 6)
(8, 6)
(8, 6)
(7, 5)
```

1.7 Working with area

Next, we work with the area column of our dictionary of column names and lists of column values.

[10]:
```
list_of_areas: list[float] =␣
↪dictionary_of_column_names_and_lists_of_column_values['area']
list_of_areas[-10:]
```

[10]:
```
[0.0, 0.0, 2.17, 0.43, 0.0, 6.44, 54.29, 11.16, 0.0, 0.0]
```

1.7.1 Task 3

(1 point)

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

[11]:
```
(min_value, max_value) = (min(list_of_areas), max(list_of_areas))
print((min_value, max_value))
print(type((min_value, max_value)))
print(type(min_value))
print(type(max_value))
```

```
(0.0, 1090.84)
<class 'tuple'>
```

```
<class 'float'>
<class 'float'>
```

1.7.2 Task 4

(2 points)

Write a lambda function that applies the following function to each element x in a list:

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

Have the lambda function return a list of logarithms, each rounded to 2 decimal places.

Assign the function to the variable `mylog10`.

Then call the lambda function on the list of areas 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-line function.
- To get the last members of a list, use negative-offset slicing. See the Python documentation on lists for a refresher on slicing.

[12]:
```
from math import log10
mylog10: Callable[list[float], list[float]] = lambda list_: [round(log10(1 +␣
 ↪x), 2) for x in list_]
list_of_logged_areas: list[float] = mylog10(list_of_areas)
print(list_of_logged_areas[-10:])
```

```
[0.0, 0.0, 0.5, 0.16, 0.0, 0.87, 1.74, 1.08, 0.0, 0.0]
```

1.8 Working with month

The month column contains months of the year in `MMM` format (`jan` to `dec`).

[13]:
```
list_of_months: list[str] =␣
 ↪dictionary_of_column_names_and_lists_of_column_values['month']
list_of_months[:10]
```

[13]:
```
['mar', 'oct', 'oct', 'mar', 'mar', 'aug', 'aug', 'aug', 'sep', 'sep']
```

1.8.1 Task 5

(1 point)

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

- Do not use `set`. Instead, use a dictionary comprehension to capture the unique names.
- Hint: The keys in a dictionary are unique.
- Hint: You do not need to count how many times a name appears in the source list.

The function should optionally return the list sorted in ascending order.

Then apply the function to the list of months with sorting turned on.

Then print the unique months.

[14]:
```
def get_uniques(list_: list, sort: bool = True) -> list:
    dictionary_of_unique_elements_as_keys_and_values: dict[Any, Any] = {element:
 ↪ element for element in list_}
    list_of_unique_elements: list[str] =␣
 ↪list(dictionary_of_unique_elements_as_keys_and_values.keys())
    if sort == False:
        return list_of_unique_elements
    else:
        month_order: list[str] = ["jan", "feb", "mar", "apr", "may", "jun",␣
 ↪"jul", "aug", "sep", "oct", "nov", "dec"]
        return sorted(list_of_unique_elements, key = lambda x: month_order.
 ↪index(x))

list_of_unique_months: list[str] = get_uniques(list_of_months, True)
print(list_of_unique_months)
```

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

1.8.2 Task 6

(1 point)

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 created.

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 parameter.

[15]:
```
get_month_for_letter: Callable[str, list[str]] = lambda letter: [month.upper()␣
 ↪for month in list_of_unique_months if month.startswith(letter)]
print(get_month_for_letter('a'))
```

```
['APR', 'AUG']
```

1.9 Working with DMC

Duff Moisture Codes (DMC's) based on the Fire Weather Index (FWI) System vary from 1.1 to 291.3.

[16]:
```
list_of_Duff_Moisture_Codes: list[float] =␣
 ↪dictionary_of_column_names_and_lists_of_column_values['DMC']
list_of_Duff_Moisture_Codes[:10]
```

[16]:
```
[26.2, 35.4, 43.7, 33.3, 51.3, 85.3, 88.9, 145.4, 129.5, 88.0]
```

1.9.1 Task 7

(2 points)

Write a function called `bandpass_filter` with these requirements:

- Takes three inputs:
  - A list of numbers `num_list`

[17]:
```
def bandpass_filter(num_list: list[float], lower_bound: int, upper_bound: int):
    return [number for number in num_list if (number > lower_bound) and (number␣
 ↪< upper_bound)]
```

1.9.2 Task 8

(1 point)

Call `bandpass_filter`, passing column DMC as the list, with `lower_bound = 25` and `upper_bound = 35`.

Then print the result.

[18]:
```
print(bandpass_filter(list_of_Duff_Moisture_Codes, 25, 35))
```

```
[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]
```

1.10 Working with FFMC

Fine Fuel Moisture Codes (FFMC's) based on the Fire Weather Index System vary from 18.7 to 96.2.

[19]:
```
list_of_Fine_Fuel_Moisture_Codes =␣
 ↪dictionary_of_column_names_and_lists_of_column_values['FFMC']
list_of_Fine_Fuel_Moisture_Codes[:10]
```

[19]:
```
[86.2, 90.6, 90.6, 91.7, 89.3, 92.3, 92.3, 91.5, 91.0, 92.5]
```

1.10.1 Task 9

(2 points)

Write a lambda function `get_mean` that computes the mean μ of a list of numbers.

- The mean is just 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 μ 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.

[20]:
```
get_mean: Callable[list[float], float] = lambda list_: sum(list_) / len(list_)
get_ssd: Callable[float, float] = lambda number_from_list, mean_of_list:␣
 ↪(number_from_list - mean_of_list)**2
def get_sum_sq_err(list_: list[float]) -> float:
    mean_of_list = get_mean(list_)
    return sum([get_ssd(number, mean_of_list) for number in list_])
```

1.10.2 Task 10

(1 point)

Call `get_sum_sq_err`, passing the list of Fine Fuel Moisture Codes as the list. Print the result.

[21]:
```
get_sum_sq_err(list_of_Fine_Fuel_Moisture_Codes)
```

[21]:
```
15723.357872340412
```

[ ]:
```

```