assignment1-solutions

February 22, 2023

1 Assignment 1

Download auto_mpg.txt and store inside 'tabular' folder created inside of 'data' directory. Add this notebook to your python codebase.

The **purpose** of this assignmnet is to become familiar with core Python (*Part 0*), numpy and Pandas basics (*Part 1*), and handling data (*Part 2*).

NAME DATE

1.1 Part 0

The goal of Part 0 is to - Practice problems based on core Python - Gain better understanding of work-flows controlled by conditional statements

1.1.1 Resources for python

Here are some of the best resources for Python on the web.

Learning resources

- Interactive Python An online book that includes embedded live excercises. Fun!
- Dive Into Python An excellent, thorough book.
- tutorial point A resource that is useful when you want an explanation of one concept, rather than a whole chapter.

Reference resources Typically, if you have a question about python, you can find an answer by using google. The following sites will usually have the best answer.

- Official python documentation
- Quick Reference from Tutorial Point

Sample. Notice the printout below the solution. Notice it is self-documenting, including problem definition and solution (i.e., source code), along with result (i.e., printout).

[1]:
```
val = 2
li = [2, 3, 4, 5]
if val in li:
    print('Found value', val, 'in list')
else:
    print('Value', val, 'not found in list')
```

```python
if 6 in li:
    print('Found value', 6, 'in list')
else:
    print('Value', 6, 'not found in list')
print('List items:', li)
```

```
Found value 2 in list
Value 6 not found in list
List items: [2, 3, 4, 5]
```

0.1) Describe the 4 core Python containers (note the keyword core, i.e., not numpy arrays or other container types that are included in Python Packages).

a) What are characteristics of each?

lists (li=[]), tuples (tu=()), strings (st=" "), dictionaries (dic={})– each are containers with various characteristics.

Lists and tuples can store any type, and can be made up of various different types. Both preserve order (as do strings), with the difference being lists are mutable, while tuples and strings are immutable.

Strings are made up of sequences of characters.

Dictionaries are key-value pairs, where values are accessed via indexing with key. Keys must be unique and are immutable, while values can be of any type and are mutable.

Each container is accessed using square brackets, with indices for tuples, lists, and strings (i.e., ordered) and keys for dictionaries (i.e., unordered).

b) Instantiate each with 0 elements (i.e., empty), and show adding a single element to each.

[2]:
```python
# instantiate empty containers
st=""
li=[]
tu=()
dic={}

# add single elements
st += "a"
li += ["a"] # or li += list("a")
tu += tuple("a")
dic["el1"] = "a"
print(st, li, tu, dic)
```

```
a ['a'] ('a',) {'el1': 'a'}
```

c) Provide 1 or more use cases for each.

ANSWER

0.2) Write a program that takes in a positive number (in some variable, say `i`) and computes the sum of all the number between 0 and that number (inclusive).

a) Do it using a for loop

[3]:
```
i = 10
```

[4]:
```
def sum_forloop(n):
    total = 0
    for x in range(n+1):
        total += x
    return total


print('Sum using list using for loop', sum_forloop(i))
```

```
Sum using list using for loop 55
```

b) Do it in one line using the function `sum` and list comprehension.

[5]:
```
sum_comprehension = lambda n: sum([x for x in range(n+1)])
print('Sum using list comprension', sum_comprehension(i))
```

```
Sum using list comprension 55
```

0.3) Create a lookup table for your class schedule, with the CRN as keys and the name of class as the value. Loop over the dictionary and print out the CRN and course name (single line per class).

[6]:
```
courses = {'cs135': 'Intro to Machine Learning', 'cs136': 'Pattern Recognition'}

[print(key, ':', value) for key, value in courses.items()]
```

```
cs135 : Intro to Machine Learning
cs136 : Pattern Recognition
```

[6]:
```
[None, None]
```

0.4) Create an empty list. Then, copy the for-loop from previous excercise such that the program prompts you to input the time of the day (as type sting, and using military time would allow for AM and PM to be omitted). These times are to be stored in empty dictionay using the same keys (i.e., CRN->time class starts)

[7]:
```
times = {}
for key in courses.keys():
    time = input('time for class ' + str(key) + ': ')
    times[key] = time
```

```
time for class cs135:  10:30 am
time for class cs136:  12:15 pm
```

0.5 Write a Python program to convert temperatures to and from Celsius, Fahrenheit.

$$\frac{c}{5} = \frac{f - 32}{9},$$

where c is the temperature in Celsius and f is the temperature in Fahrenheit.

Test code: 60°C is 140 in Fahrenheit 45°F is 7 in Celsius

[8]:
```
def fahrenheit2celsius(fahrenheit):
    return 5*(fahrenheit - 32) / 9

def celsius2fahrenheit(celsius):
    return 9 * celsius / 5 + 32


temp_c = 60
temp_f = 45

print("{} deg. F is {} deg. C".format(temp_f, fahrenheit2celsius(temp_f)))
print("{} deg. C is {} deg. F".format(temp_c, celsius2fahrenheit(temp_c)))
```

```
45 deg. F is 7.222222222222222 deg. C
60 deg. C is 140.0 deg. F
```

0.6 Write a Python program to construct the following pattern, using a nested for loop.

O

O X

O X O

O X O X

O X O X O

O X O X

O X O

O X

O

[9]:
```
mystring = ""
pattern = 'OXOXO\n'
mystring+=pattern
for i in range(2, len(pattern)):
    mystring = pattern[:-i] + "\n" + mystring + pattern[:-i] + "\n"
print(mystring)
```

```
O
OX
OXO
OXOX
OXOXO
OXOX
OXO
```

```
OX
O
```

0.7 Write a Python program that reads two integers representing a month and day and prints the season for that month and day. Go to the editor Expected Output:

Input the month (e.g. January, February etc.): july
Input the day: 31
Season is summer

[21]:
```python
month = input("Input the month (e.g. January, February etc.): ")
day = int(input("Input the day: "))

# make all lower-case so not case sensitive
month = month.lower()
if month in ('january', 'february', 'march'):
    season = 'winter'
elif month in ('april', 'may', 'june'):
    season = 'spring'
elif month in ('july', 'august', 'september'):
    season = 'summer'
else:
    season = 'autumn'

if (month.lower() == 'march') and (day > 19):
    season = 'spring'
elif (month.lower() == 'june') and (day > 20):
    season = 'summer'
elif (month.lower() == 'september') and (day > 21):
    season = 'autumn'
elif (month.lower() == 'december') and (day > 20):
    season = 'winter'
else:
    pass

print(f"Season is of {day} {month} is {season}")
```

```
Input the month (e.g. January, February etc.):  July
Input the day:  31

Season is of 31 july is summer
```

0.8 Implement repeats(), as specified in doc-string. Then call on variables a and b below. Print True if repeated, else, print False.

[11]:
```python
a = [1, 3, 1, 6, 3, 5, 5, 2]
b = [1, 2, 3, 3, 4, 5, 6, 7, 8, 9]
```

```
def repeated_val(xs, val=5):
    """
    Function to search whether or not 'val' is repeated in sequence.

    :param xs:      List of items to search
    :param val:     Val being searched (default = 5)
    :return:        True if repeated 'val' and neigbhors, i.e., [..., 'val',␣
↪'val', ...] = True; else, False
    """
    last = None
    for x in xs:
        if x == last and x == val:
            return True
        last = x
    return False


print("list 'a' repeats 5:", repeated_val(a))
print("list 'a' repeats 6:", repeated_val(a, val=6))
print("list 'b' repeats 5:", repeated_val(b))
```

```
list 'a' repeats 5: True
list 'a' repeats 6: False
list 'b' repeats 5: False
```

0.9 Implement sum_for_loop() below, and call on variable i.

[12]:
```
i = 10
def sum_for_loop(n):
    """ Computes and returns the sum of all the numbers between 0 and n

    :param n:      positive integer to sum from zero to
    """
    if type(n) is not int or n < 0:
        return None
```

1.2 Part 1

The goal in this part is to - understand basic functionality of numpy and pandas - learn how to use numpy and pandas to solve common coding tasks - understand these packages to process real-world data

As import the libraries such to allow the numpy library to be called by with np and pandas pd

[13]:
```
import numpy as np
import pandas as pd
import os
```

1.2.1 a) Numpy Basics

1.2.2 1)

Create a 10x10 array with random values and find the minimum and maximum values

[14]:
```
np.random.seed(123)
Z = np.random.random((10,10))
Zmin, Zmax = Z.min(), Z.max()
print(Zmin, Zmax)
```

```
0.016129206695016833 0.9953584820340174
```

1.2.3 2)

Extract the integer part of array Z using 5 different numpy methods

[15]:
```
np.random.seed(123)
Z = np.random.uniform(0, 10, 10)
print (Z - Z%1)
print (np.floor(Z))
print (np.ceil(Z)-1)
print (Z.astype(int))
print (np.trunc(Z))
```

```
[6. 2. 2. 5. 7. 4. 9. 6. 4. 3.]
[6. 2. 2. 5. 7. 4. 9. 6. 4. 3.]
[6. 2. 2. 5. 7. 4. 9. 6. 4. 3.]
[6 2 2 5 7 4 9 6 4 3]
[6. 2. 2. 5. 7. 4. 9. 6. 4. 3.]
```

Create a vector of size 20 with values (0, 1), i.e., both excluded

[16]:
```
Z = np.linspace(0, 1, 22, endpoint=True)[1:-1]
print(Z)
```

```
[0.04761905 0.0952381  0.14285714 0.19047619 0.23809524 0.28571429
 0.33333333 0.38095238 0.42857143 0.47619048 0.52380952 0.57142857
 0.61904762 0.66666667 0.71428571 0.76190476 0.80952381 0.85714286
 0.9047619  0.95238095]
```

Create a random vector of size 15 and sort it

[17]:
```
np.random.seed(123)
Z = np.random.random(15)
Z.sort()
print(Z)
```

```
[0.0596779  0.22685145 0.28613933 0.34317802 0.39211752 0.39804426
 0.42310646 0.43857224 0.4809319  0.55131477 0.68482974 0.69646919
 0.71946897 0.72904971 0.9807642 ]
```

Consider two random array A anb B, check if they are equal

[18]:
```
np.random.seed(123)
A = np.random.randint(0, 2, 5)
B = np.random.randint(0, 2, 5)
```

[19]:
```
equal = np.allclose(A, B)
print(equal)
A = B
equal = np.allclose(A, B)
print(equal)
```

```
False
True
```

matplotlib is the plotting library which pandas' plotting functionality is built upon, and it is usually aliased to plt.

%matplotlib inline tells the notebook to show plots inline, instead of creating them in a separate window.

plt.style.use('ggplot') is a style theme that most people find agreeable, based upon the styling of R's ggplot package.

See the documentation https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.plot.html if you get stuck!

Make an array immutable (read-only)

[20]:
```
Z = np.zeros(10)
Z.flags.writeable = False
Z[0] = 1
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[20], line 3
      1 Z = np.zeros(10)
      2 Z.flags.writeable = False
----> 3 Z[0] = 1

ValueError: assignment destination is read-only
```

What if we want to plot multiple things? Pandas allows you to pass in a matplotlib Axis object for plots, and plots will also return an Axis object.

Make a bar plot of monthly revenue with a line plot of monthly advertising spending (numbers in millions)

Create a structured array representing a position (x,y) and a color (r,g,b). Instantiate structured array's values to be all zeros (though same method for other values as well).

```
[ ]: Z = np.zeros(10, [ ('position', [ ('x', float, 1),
                                       ('y', float, 1)]),
                      ('color',    [ ('r', float, 1),
                                       ('g', float, 1),
                                       ('b', float, 1)])])

     # Z['position']['x']
     # Z
```

```
[34]: print(Z)
```

```
[((0., 0.), (0., 0., 0.)) ((0., 0.), (0., 0., 0.))
 ((0., 0.), (0., 0., 0.)) ((0., 0.), (0., 0., 0.))
 ((0., 0.), (0., 0., 0.)) ((0., 0.), (0., 0., 0.))
 ((0., 0.), (0., 0., 0.)) ((0., 0.), (0., 0., 0.))
 ((0., 0.), (0., 0., 0.)) ((0., 0.), (0., 0., 0.))]
```

```
[35]: print(Z['position'])
```

```
[(0., 0.) (0., 0.) (0., 0.) (0., 0.) (0., 0.) (0., 0.) (0., 0.) (0., 0.)
 (0., 0.) (0., 0.)]
```

```
[36]: print(Z['color']['b'])
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

Considering a four dimensions array, how to get sum over the last two axis at once?

```
[37]: np.random.seed(123)
      A = np.random.randint(0, 10, (3, 4, 3, 4))
      summation = A.reshape(A.shape[:-2] + (-1,)).sum(axis=-1)
      print(summation)
```

```
[[40 37 56 43]
 [61 44 54 70]
 [48 53 64 71]]
```

Considering a (w,h,3) image of (dtype=ubyte), compute the number of unique colors

```
[38]: np.random.seed(123)
      w, h = 16, 16
      I = np.random.randint(0, 2, (h, w, 3)).astype(np.ubyte)
      print(I.shape)
      F = I[..., 0] * 2 * 2 + I[..., 1] * 2 + I[..., 2] # convert base 2 --> base 10
      n = len(np.unique(F))
      print(F)
      print(n, 'unique colors')
```

```
(16, 16, 3)
```

```
[[2 0 3 3 2 5 4 3 5 0 3 4 4 5 3 4]
 [1 4 5 1 2 4 1 1 7 0 3 0 2 6 7 3]
 [4 2 3 5 4 6 7 7 5 7 0 6 3 0 5 4]
 [5 7 2 4 6 7 3 6 3 1 0 5 4 2 2 2]
 [2 0 5 4 2 1 0 5 7 3 6 3 7 1 6 4]
 [3 5 1 6 6 1 3 1 7 5 6 6 1 4 4 2]
 [7 3 4 1 3 6 7 7 1 1 3 7 5 6 1 5]
 [1 3 0 3 7 2 7 1 1 0 1 5 0 6 5 3]
 [0 5 4 4 6 3 2 7 3 7 1 7 7 7 7 1]
 [6 2 0 4 1 2 4 1 5 2 7 2 0 7 0 7]
 [0 2 6 4 4 2 0 3 1 7 6 2 6 5 1 6]
 [3 4 7 6 4 2 7 0 5 5 0 0 6 7 6 2]
 [3 1 7 4 3 7 6 1 7 0 6 6 3 7 1 0]
 [7 3 0 0 5 7 3 2 2 0 3 2 7 6 2 2]
 [3 4 5 0 7 5 2 1 3 4 5 3 6 4 2 5]
 [4 4 7 6 3 5 3 0 1 1 0 3 6 1 5 0]]
8 unique colors
```

How to accumulate elements of a vector (X) to an array (F) based on an index list (I)?

[39]:
```
X = [1, 2, 3, 4, 5, 6]
I = [1, 3, 9, 3, 4, 1]
F = np.bincount(I, X)
print(F)
```

```
[0. 7. 0. 6. 5. 0. 0. 0. 0. 3.]
```

[40]:
```
Z = np.ones(10)
I = np.random.randint(0,len(Z),20)
Z += np.bincount(I, minlength=len(Z))
print(I)
print(Z)
```

```
[3 2 4 8 0 6 8 8 5 6 5 6 0 4 9 0 8 0 5 6]
[5. 1. 2. 2. 3. 4. 5. 1. 5. 2.]
```

How to read the following file?

[41]:
```
fpath = os.path.join("data", "tabular", "missing.dat")
Z = np.genfromtxt(fpath, delimiter=",")
print(Z)
```

```
[[ 1.   2.   3.   4.   5.]
 [ 6.  nan  nan   7.   8.]
 [nan  nan   9.  10.  11.]]
```

Convert a vector of ints into a matrix binary representation

[42]:
```
I = np.array([0, 1, 2, 3, 15, 16, 32, 64, 128])
B = ((I.reshape(-1,1) & (2**np.arange(8))) != 0).astype(int)
print("Solution 1:")
print(B[:,::-1])
print("Solution 2:")
print(np.unpackbits(I.astype(np.uint8)[:, np.newaxis], axis=1))
B.shape
print(type(B[0][0]))
```

```
Solution 1:
[[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 1 1]
 [0 0 0 0 1 1 1 1]
 [0 0 0 1 0 0 0 0]
 [0 0 1 0 0 0 0 0]
 [0 1 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0]]
Solution 2:
[[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 1 1]
 [0 0 0 0 1 1 1 1]
 [0 0 0 1 0 0 0 0]
 [0 0 1 0 0 0 0 0]
 [0 1 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0]]
<class 'numpy.int64'>
```

Given a two dimensional array, how to extract unique rows?

[43]:
```
Z = np.random.randint(0, 2, (6, 3))
print(Z)
T = np.ascontiguousarray(Z).view(np.dtype((np.void, Z.dtype.itemsize * Z.
 ↪shape[1])))
_, idx = np.unique(T, return_index=True)
uZ = Z[idx]
print(uZ)
```

```
[[1 0 0]
 [0 0 0]
 [0 1 1]
 [0 1 1]
 [0 1 1]
 [0 1 0]]
[[0 0 0]
```

```
 [0 1 0]
 [0 1 1]
 [1 0 0]]
```

1.3 Pandas

Made-up data representing animals and trips to vet

[44]:
```
data = {'animal': ['cat', 'cat', 'snake', 'dog', 'dog', 'cat', 'snake', 'cat',␣
 ↪'dog', 'dog'],
        'age': [2.5, 3, 0.5, np.nan, 5, 2, 4.5, np.nan, 7, 3],
        'visits': [1, 3, 2, 3, 2, 3, 1, 1, 2, 1],
        'priority': ['yes', 'yes', 'no', 'yes', 'no', 'no', 'no', 'yes', 'no',␣
 ↪'no']}

labels = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

Create a DataFrame df from this dictionary data which has the index labels.

[45]:
```
df = pd.DataFrame(data, index=labels)
print(df)
```

```
  animal  age  visits priority
a    cat  2.5       1      yes
b    cat  3.0       3      yes
c  snake  0.5       2       no
d    dog  NaN       3      yes
e    dog  5.0       2       no
f    cat  2.0       3       no
g  snake  4.5       1       no
h    cat  NaN       1      yes
i    dog  7.0       2       no
j    dog  3.0       1       no
```

Display a summary of the basic information about this DataFrame and its data.

[46]:
```
df.info()

# ...or...

df.describe()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 10 entries, a to j
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   animal    10 non-null     object
 1   age       8 non-null      float64
```

```
 2   visits    10 non-null     int64
 3   priority  10 non-null     object
dtypes: float64(1), int64(1), object(2)
memory usage: 400.0+ bytes
```

[46]:

```
            age     visits
count  8.000000  10.000000
mean   3.437500   1.900000
std    2.007797   0.875595
min    0.500000   1.000000
25%    2.375000   1.000000
50%    3.000000   2.000000
75%    4.625000   2.750000
max    7.000000   3.000000
```

Return the first 3 rows of the DataFrame df.

[47]:

```python
df.iloc[:3]

# or equivalently

df.head(3)
```

[47]:

```
  animal  age  visits priority
a    cat  2.5       1      yes
b    cat  3.0       3      yes
c  snake  0.5       2       no
```

Select just the 'animal' and 'age' columns from the DataFrame df.

[48]:

```python
df.loc[:, ['animal', 'age']]

# or

df[['animal', 'age']]
```

[48]:

```
  animal  age
a    cat  2.5
b    cat  3.0
c  snake  0.5
d    dog  NaN
e    dog  5.0
f    cat  2.0
g  snake  4.5
h    cat  NaN
i    dog  7.0
j    dog  3.0
```

Change the age in row 'f' to 1.5.

[49]:
```
df.loc['f', 'age'] = 1.5
```

Calculate the mean age for each different animal in df.

[50]:
```
df.groupby('animal')['age'].mean()
```

[50]:
```
animal
cat      2.333333
dog      5.000000
snake    2.500000
Name: age, dtype: float64
```

In the 'animal' column, change the 'snake' entries to 'python'.

[51]:
```
df['animal'] = df['animal'].replace('snake', 'python')
print(df)
```

```
   animal  age  visits priority
a     cat  2.5       1      yes
b     cat  3.0       3      yes
c  python  0.5       2       no
d     dog  NaN       3      yes
e     dog  5.0       2       no
f     cat  1.5       3       no
g  python  4.5       1       no
h     cat  NaN       1      yes
i     dog  7.0       2       no
j     dog  3.0       1       no
```

For each animal type and each number of visits, find the mean age. In other words, each row is an animal, each column is a number of visits and the values are the mean ages (hint: use a pivot table).

[52]:
```
df.pivot_table(index='animal', columns='visits', values='age', aggfunc='mean')
```

[52]:
```
visits    1    2     3
animal
cat     2.5  NaN  2.25
dog     3.0  6.0   NaN
python  4.5  0.5   NaN
```

Given a DataFrame, subtract the row mean from each element in the row?

[53]:
```
# a 5x3 frame of float values
df = pd.DataFrame(np.random.random(size=(5, 3)))
df.sub(df.mean(axis=1), axis=0)
```

[53]:
```
          0         1         2
0  0.270479 -0.137419 -0.133059
```

```
1 -0.299757  0.314608 -0.014851
2 -0.026786  0.216316 -0.189530
3 -0.298346 -0.101595  0.399941
4 -0.076117 -0.288617  0.364733
```

1.3.1 Series and Datetimeindex

Create a DatetimeIndex that contains each business day of 2015 and use it to index a Series of random numbers. Let's call this Series s.

[54]:
```python
dti = pd.date_range(start='2015-01-01', end='2015-12-31', freq='B')
s = pd.Series(np.random.rand(len(dti)), index=dti)
print(s)
```

```
2015-01-01    0.815966
2015-01-02    0.322974
2015-01-05    0.972098
2015-01-06    0.987351
2015-01-07    0.408660
                …
2015-12-25    0.440462
2015-12-28    0.844077
2015-12-29    0.076204
2015-12-30    0.481128
2015-12-31    0.466850
Freq: B, Length: 261, dtype: float64
```

Find the sum of the values in s for every Wednesday.

[55]:
```python
s[s.index.weekday == 2].sum()
```

[55]:
```
27.95818113929446
```

For each calendar month in s, find the mean of values.

[56]:
```python
s.resample('M').mean()
```

[56]:
```
2015-01-31    0.459806
2015-02-28    0.522544
2015-03-31    0.415980
2015-04-30    0.569095
2015-05-31    0.522860
2015-06-30    0.492323
2015-07-31    0.469943
2015-08-31    0.593500
2015-09-30    0.477171
2015-10-31    0.578573
2015-11-30    0.494291
2015-12-31    0.539266
```

```
Freq: M, dtype: float64
```

For each group of four consecutive calendar months in s, find the date on which the highest value occurred.

[57]:
```
s.groupby(pd.Grouper(freq='4M')).idxmax()
```

[57]:
```
2015-01-31   2015-01-06
2015-05-31   2015-03-13
2015-09-30   2015-06-02
2016-01-31   2015-10-28
Freq: 4M, dtype: datetime64[ns]
```

1.3.2 Cleaning Data

The DataFrame to use in the following puzzles:

[58]:
```
df = pd.DataFrame({'From_To': ['LoNDon_paris', 'MAdrid_miLAN',␣
 ↪'londON_StockhOlm',
                               'Budapest_PaRis', 'Brussels_londOn'],
              'FlightNumber': [10045, np.nan, 10065, np.nan, 10085],
              'RecentDelays': [[23, 47], [], [24, 43, 87], [13], [67, 32]],
                   'Airline': ['KLM(!)', '<Air France> (12)', '(British Airways.
 ↪ )',
                               '12. Air France', '"Swiss Air"']})

df.head()
```

[58]:
```
            From_To  FlightNumber  RecentDelays               Airline
0      LoNDon_paris       10045.0      [23, 47]                KLM(!)
1      MAdrid_miLAN           NaN            []     <Air France> (12)
2  londON_StockhOlm       10065.0  [24, 43, 87]   (British Airways. )
3    Budapest_PaRis           NaN          [13]        12. Air France
4   Brussels_londOn       10085.0      [67, 32]           "Swiss Air"
```

Some values in the the FlightNumber column are missing. These numbers are meant to increase by 10 with each row so 10055 and 10075 need to be put in place. Fill in these missing numbers and make the column an integer column (instead of a float column).

[59]:
```
df['FlightNumber'] = df['FlightNumber'].interpolate().astype(int)
print(df)
```

```
            From_To  FlightNumber  RecentDelays               Airline
0      LoNDon_paris         10045      [23, 47]                KLM(!)
1      MAdrid_miLAN         10055            []     <Air France> (12)
2  londON_StockhOlm         10065  [24, 43, 87]   (British Airways. )
3    Budapest_PaRis         10075          [13]        12. Air France
4   Brussels_londOn         10085      [67, 32]           "Swiss Air"
```

The From_To column would be better as two separate columns! Split each string on the underscore delimiter _ to give a new temporary DataFrame with the correct values. Assign the correct column names to this temporary DataFrame.

[60]:
```
temp = df.From_To.str.split('_', expand=True)
temp.columns = ['From', 'To']
```

Notice how the capitalisation of the city names is all mixed up in this temporary DataFrame. Standardise the strings so that only the first letter is uppercase (e.g. "londON" should become "London".)

[61]:
```
temp['From'] = temp['From'].str.capitalize()
temp['To'] = temp['To'].str.capitalize()
print(temp)
```

```
       From         To
0    London      Paris
1    Madrid      Milan
2    London  Stockholm
3  Budapest      Paris
4  Brussels     London
```

Delete the From_To column from df and attach the temporary DataFrame from the previous questions.

[62]:
```
df = df.drop('From_To', axis=1)
df = df.join(temp)
print(df)
```

```
   FlightNumber  RecentDelays              Airline      From         To
0         10045      [23, 47]               KLM(!)    London      Paris
1         10055            []    <Air France> (12)    Madrid      Milan
2         10065  [24, 43, 87]  (British Airways. )    London  Stockholm
3         10075          [13]       12. Air France  Budapest      Paris
4         10085      [67, 32]          "Swiss Air"  Brussels     London
```

1.3.3 Plotting

Pandas is integrated with the plotting library matplotlib, and makes plotting DataFrames very user-friendly! Plotting in a notebook environment usually makes use of the following boilerplate:

matplotlib is the plotting library which pandas' plotting functionality is built upon, and it is usually aliased to plt.

%matplotlib inline tells the notebook to show plots inline, instead of creating them in a separate window.

plt.style.use('ggplot') is a style theme that most people find agreeable, based upon the styling of R's ggplot package.

See the documentation https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.plot.html if you get stuck!

```
[63]: import matplotlib.pyplot as plt
      %matplotlib inline
      plt.style.use('ggplot')
```

```
[64]: df = pd.DataFrame({"xs":[1,5,2,8,1], "ys":[4,2,1,9,6]})
```

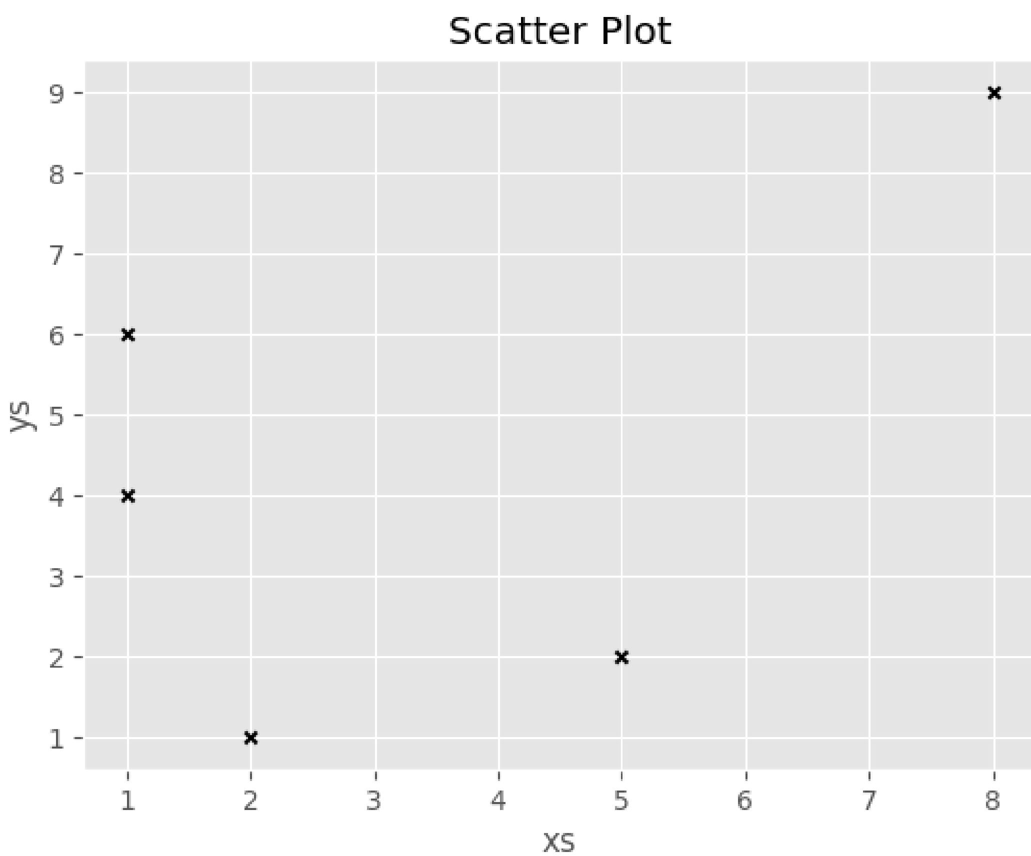
1.3.4 1.31)

For starters, make a scatter plot of this random data, but use black X's instead of the default markers. Add title "Scatter Plot" to the plot. Use df from previous cell.

NOTE: Don't forget to add [any] title and axes labels

```
[65]: df.plot.scatter(x="xs", y="ys", color = "black", marker = "x", title="Scatter␣
      ↪Plot")
```

```
[65]: <AxesSubplot: title={'center': 'Scatter Plot'}, xlabel='xs', ylabel='ys'>
```



Columns in your DataFrame can also be used to modify colors and sizes. Bill has been keeping track of his performance at work over time, as well as how good he was feeling that day, and whether he had a cup of coffee in the morning. Make a plot which incorporates all four features of this DataFrame.

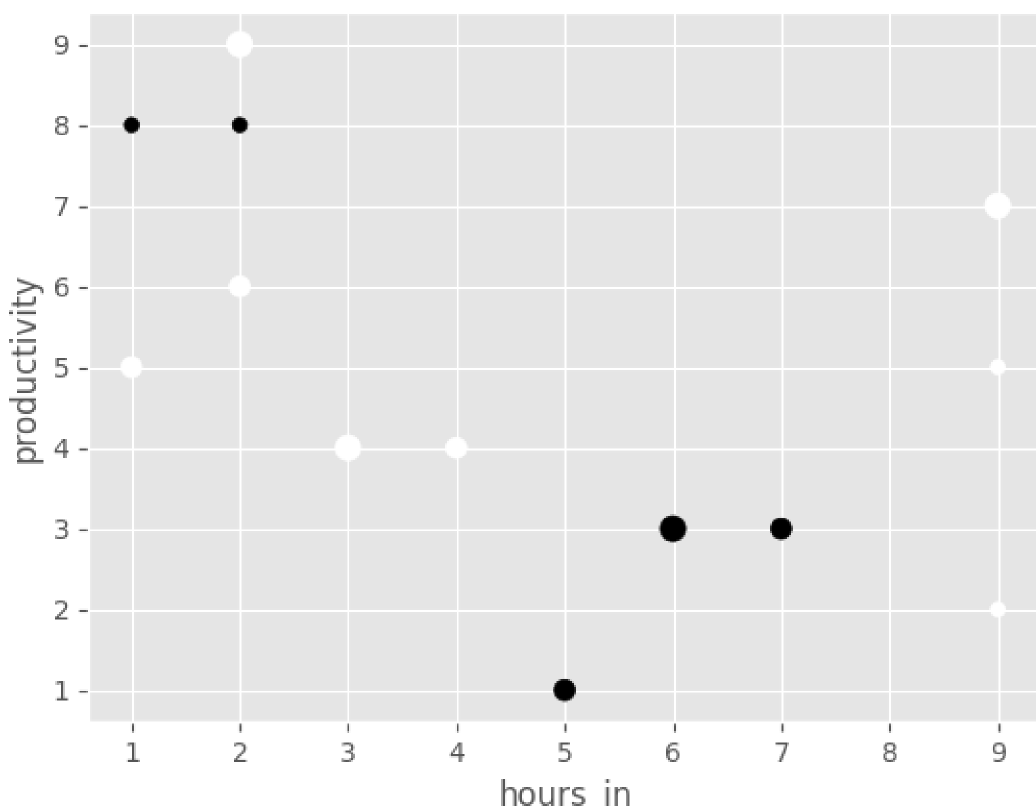
(Hint: If you're having trouble seeing the plot, try multiplying the Series which you choose to represent size by 10 or more)

The chart doesn't have to be pretty: this isn't a course in data viz!

[66]:
```
df2 = pd.DataFrame({"productivity":[5,2,3,1,4,5,6,7,8,3,4,8,9],
                    "hours_in"    :[1,9,6,5,3,9,2,9,1,7,4,2,2],
                    "happiness"   :[2,1,3,2,3,1,2,3,1,2,2,1,3],
                    "caffienated" :[0,0,1,1,0,0,0,0,1,1,0,1,0]})

df2.plot.scatter("hours_in", "productivity", s = df2.happiness * 30, c = df2.
 ↪caffienated)
```

[66]: <AxesSubplot: xlabel='hours_in', ylabel='productivity'>



1.3.5 1.33)

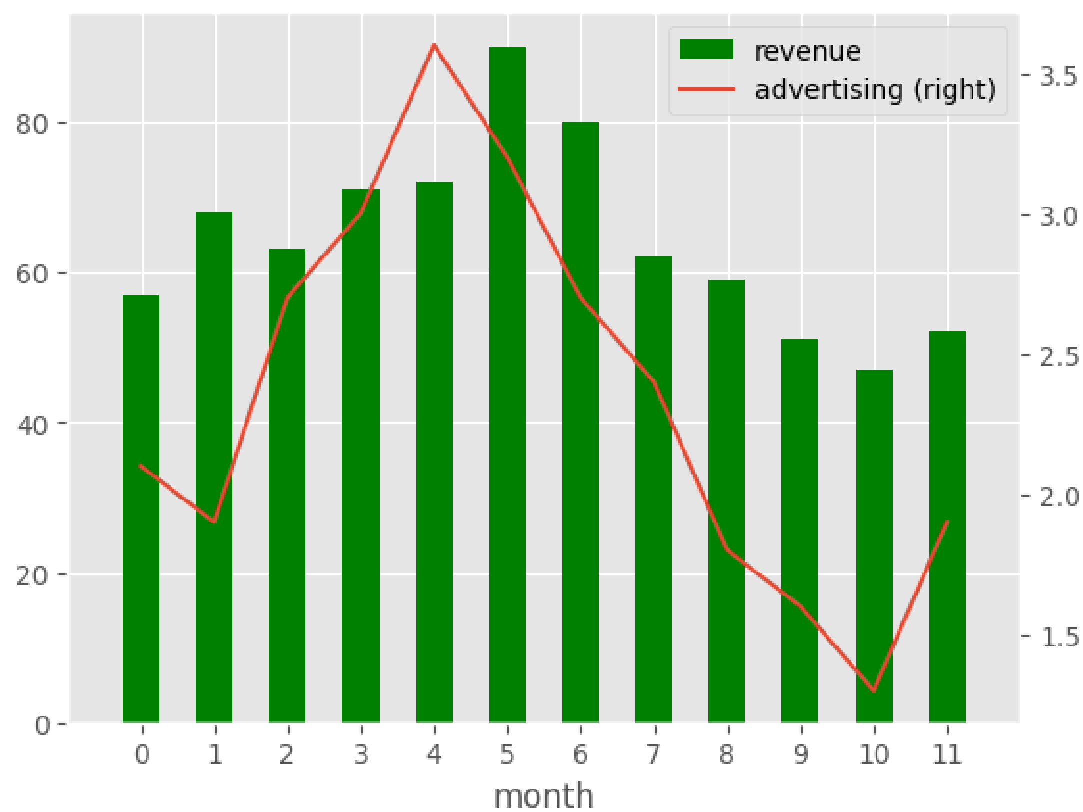
What if we want to plot multiple things? Pandas allows you to pass in a matplotlib Axis object for plots, and plots will also return an Axis object.

Make a bar plot of monthly revenue with a line plot of monthly advertising spending (numbers in millions) - Two plots should be in one figure - Make sure that the y-axis scales of 2 plots are different - Be sure to include legend

[67]:
```
df = pd.DataFrame({"revenue":[57,68,63,71,72,90,80,62,59,51,47,52],
                   "advertising":[2.1,1.9,2.7,3.0,3.6,3.2,2.7,2.4,1.8,1.6,1.3,1.
 ↪9],
                   "month":range(12)
                  })

ax = df.plot.bar("month", "revenue", color = "green")
df.plot.line("month", "advertising", secondary_y = True, ax = ax)
ax.set_xlim((-1,12))
```

[67]: (-1.0, 12.0)



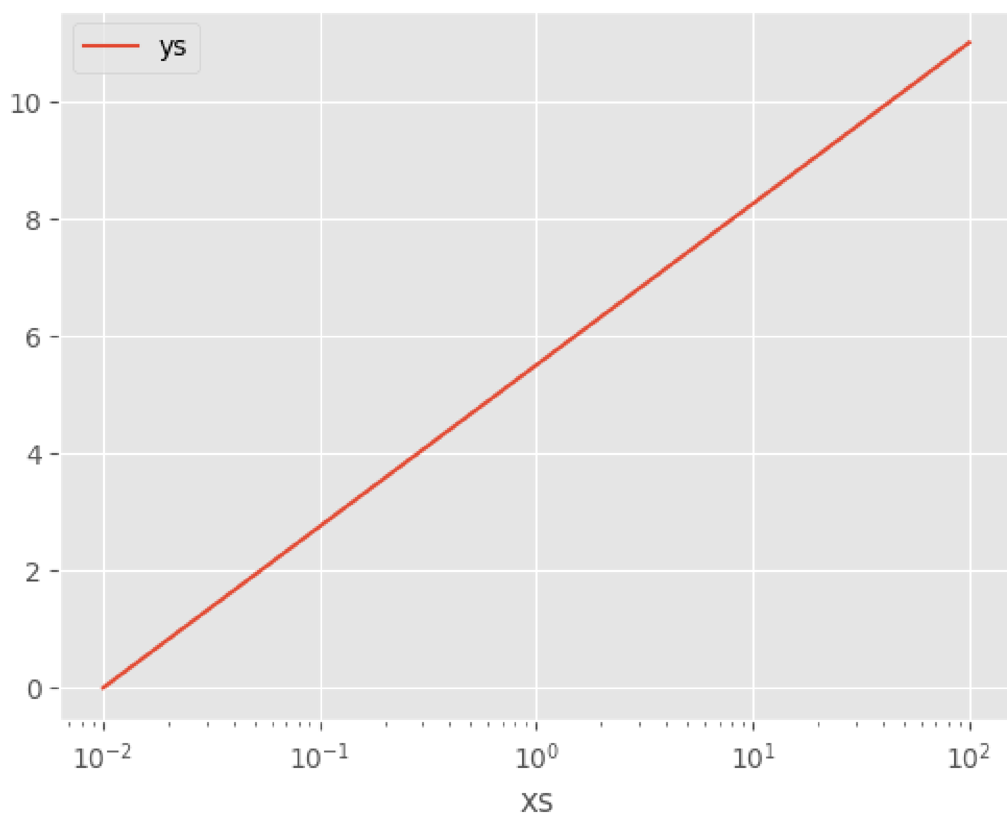
1.3.6 1.33)

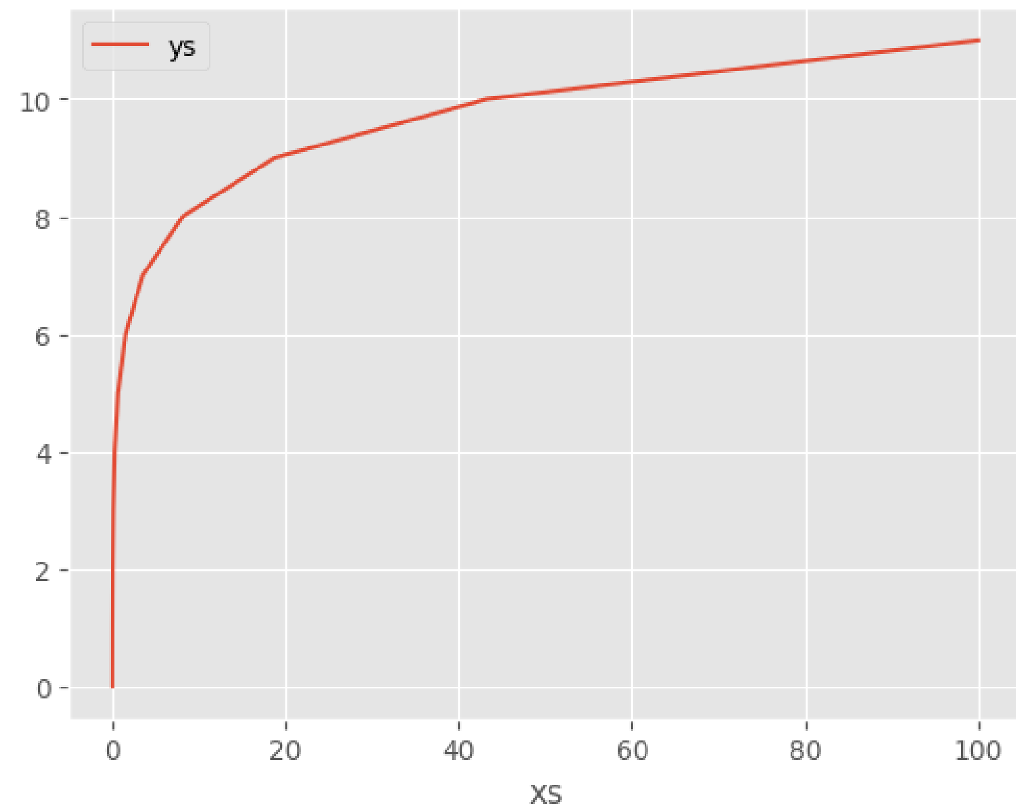
What if we want to put the x-axis in a different scale? Create two line plots with xs as x-axis and ys as y-axis. First plot uses log scaling on x-axis, and the second plot uses default scaling on x-axis.

[68]:
```
df3 = pd.DataFrame({"xs":np.logspace(-2, 2, base=10, num=12),
                    "ys":range(12)
                   })

df3.plot.line("xs", "ys", logx = True)
df3.plot.line("xs", "ys", logx = False)
```

[68]: <AxesSubplot: xlabel='xs'>





##Matrix Manipulations Lets first create a matrix and perform some manipulations of it.

Using numpy's matrix data structure, define the following matricies:

$$A = \begin{bmatrix} 3 & 5 & 9 \\ 3 & 3 & 4 \\ 5 & 9 & 17 \end{bmatrix}$$

$$B = \begin{bmatrix} 2 \\ 1 \\ 4 \end{bmatrix}$$

After this solve the matrix equation:

$$Ax = B$$

Now write three functions for matrix multiply $C = AB$ in each of the following styles:

1. By using nested for loops to impliment the naive algorithm ($C_{ij} = \sum_{k=0}^{m-1} A_{ik}B_{kj}$)
2. Using numpy's built in martrix multiplication

Both methods should have the same answer

[69]:
```
A=np.matrix('3 5 9; 3 3 4; 5 9 17', dtype=float)
B=np.matrix('2; 1; 4', dtype=float)
#print A.shape, A.dtype, A[0,:]
#print B.shape, B.dtype, B
x = np.linalg.solve(A, B)
print('x =', x)
print()

def mm1(A, B):
    if A.shape[1] != B.shape[0]:
        print( "Number of columns of A must equal number of rows of B")
        return None
    C=np.zeros((A.shape[0], B.shape[1]))
    for i in range(A.shape[0]):
        for j in range(B.shape[1]):
            for k in range(A.shape[1]):
                C[i, j] += A[i, k] * B[k, j]
    return C

def mm2(A, B):
#    return(A * B)
    return(np.dot(A, B))

print( mm1(A, B))
print()
print( mm2(A, B))
```

```
x = [[ 1.]
 [-2.]
 [ 1.]]

[[47.]
 [25.]
 [87.]]

[[47.]
 [25.]
 [87.]]
```

1.4 Part 2

Getting used to the data

[70]:
```
# Reads text file and uses '|' as separator
auto = pd.read_table('data/tabular/auto_mpg.txt', sep='|')
auto.head()
```

[70]:
```
    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0         130    3504          12.0
1  15.0          8         350.0         165    3693          11.5
2  18.0          8         318.0         150    3436          11.0
3  16.0          8         304.0         150    3433          12.0
4  17.0          8         302.0         140    3449          10.5

   model_year  origin                   car_name
0          70       1  chevrolet chevelle malibu
1          70       1          buick skylark 320
2          70       1         plymouth satellite
3          70       1              amc rebel sst
4          70       1                ford torino
```

Answer the following questions about the data:

a) What is the shape of the data?

[71]:
```python
auto.shape # There are 392 rows and 9 columns
```

[71]:
```
(392, 9)
```

b) How many rows and columns are there?

[72]:
```python
auto.columns # This lists the column names that are available
```

[72]:
```
Index(['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'model_year', 'origin', 'car_name'],
      dtype='object')
```

c) What variables are available?

[73]:
```python
auto.info() # This lists the column names as well as their data type.
# You can infer the range from the information available in describe
auto.describe() # This will give you the five number summary for all numeric␣
 ↪variables
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 392 entries, 0 to 391
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   mpg           392 non-null    float64
 1   cylinders     392 non-null    int64
 2   displacement  392 non-null    float64
 3   horsepower    392 non-null    int64
 4   weight        392 non-null    int64
 5   acceleration  392 non-null    float64
 6   model_year    392 non-null    int64
```

```
 7   origin        392 non-null    int64
 8   car_name      392 non-null    object
dtypes: float64(3), int64(5), object(1)
memory usage: 27.7+ KB
```

[73]:

```
              mpg   cylinders  displacement  horsepower       weight  \
count  392.000000  392.000000    392.000000  392.000000   392.000000
mean    23.445918    5.471939    194.411990  104.469388  2977.584184
std      7.805007    1.705783    104.644004   38.491160   849.402560
min      9.000000    3.000000     68.000000   46.000000  1613.000000
25%     17.000000    4.000000    105.000000   75.000000  2225.250000
50%     22.750000    4.000000    151.000000   93.500000  2803.500000
75%     29.000000    8.000000    275.750000  126.000000  3614.750000
max     46.600000    8.000000    455.000000  230.000000  5140.000000

       acceleration  model_year      origin
count    392.000000  392.000000  392.000000
mean      15.541327   75.979592    1.576531
std        2.758864    3.683737    0.805518
min        8.000000   70.000000    1.000000
25%       13.775000   73.000000    1.000000
50%       15.500000   76.000000    1.000000
75%       17.025000   79.000000    2.000000
max       24.800000   82.000000    3.000000
```

d) What are the ranges for the values in each numeric column?

[78]:

```python
auto.min(numeric_only=True) # This will give you all of the minimums for␣
 ↪numeric variables
auto.max(numeric_only=True) # This will give you all of the maximums for␣
 ↪numeric variables
# You can calculate the range with the above info as shown below.
auto.max(numeric_only=True) - auto.min(numeric_only=True) # Range
```

[78]:

```
mpg               37.6
cylinders          5.0
displacement     387.0
horsepower       184.0
weight          3527.0
acceleration      16.8
model_year        12.0
origin             2.0
dtype: float64
```

e) What is the average value for each column? Does that differ significantly from the median?

[80]:

```python
mu = auto.mean(numeric_only=True) # Means for all numeric variables
median = auto.median(numeric_only=True) # Medians for all numeric variables
```

[81]:
```python
# How much greater is the mean than the median?
diff = mu - median
# The means are somewhat greater than the medians.
print(mu, median, diff)
```

```
mpg                23.445918
cylinders           5.471939
displacement      194.411990
horsepower        104.469388
weight           2977.584184
acceleration       15.541327
model_year         75.979592
origin              1.576531
dtype: float64 mpg                22.75
cylinders           4.00
displacement      151.00
horsepower         93.50
weight           2803.50
acceleration       15.50
model_year         76.00
origin              1.00
dtype: float64 mpg                0.695918
cylinders          1.471939
displacement      43.411990
horsepower        10.969388
weight           174.084184
acceleration       0.041327
model_year        -0.020408
origin             0.576531
dtype: float64
```

Answer the following questions about the data:

a) Which 5 cars get the best gas mileage?

[82]:
```python
# 5 cars that get best gas mileage
auto.sort_values(by="mpg", ascending=False)[0:5][['car_name','mpg']]
```

[82]:

	car_name	mpg
320	mazda glc	46.6
327	honda civic 1500 gl	44.6
323	vw rabbit c (diesel)	44.3
388	vw pickup	44.0
324	vw dasher (diesel)	43.4

b) Which 5 cars with more than 4 cylinders get the best gas mileage?

[83]:
```python
# 5 cars with more than 4 cylinders that get the best gas mileage
auto[auto.cylinders > 4].sort_values(by='mpg', ascending=False)[0:
↪5][['car_name','mpg']]
```

[83]:

	car_name	mpg
381	oldsmobile cutlass ciera (diesel)	38.0
325	audi 5000s (diesel)	36.4
330	datsun 280-zx	32.7
355	volvo diesel	30.7
304	chevrolet citation	28.8

c) Which 5 cars get the worst gas mileage?

[84]:
```python
# 5 cars that get worst gas mileage
auto.sort_values(by='mpg')[0:5][['car_name','mpg']]
```

[84]:

	car_name	mpg
28	hi 1200d	9.0
26	chevy c20	10.0
25	ford f250	10.0
27	dodge d200	11.0
123	oldsmobile omega	11.0

d) Which 5 cars with 4 or fewer cylinders get the worst gas mileage?

[85]:
```python
# 5 cars with 4 or fewer cylinders that get the worst gas mileage
auto[auto.cylinders > 4].sort_values(by='mpg')[0:5][['car_name','mpg']]
```

[85]:

	car_name	mpg
28	hi 1200d	9.0
25	ford f250	10.0
26	chevy c20	10.0
66	mercury marquis	11.0
123	oldsmobile omega	11.0

Part 4 Use groupby and aggregations to explore the relationships between mpg and the other variables. Which variables seem to have the greatest effect on mpg? Some examples of things you might want to look at are: - What is the mean mpg for cars for each number of cylindres (i.e. 3 cylinders, 4 cylinders, 5 cylinders, etc)? - Did mpg rise or fall over the years contained in this dataset? - What is the mpg for the group of lighter cars vs the group of heaver cars? Note: Be creative in the ways in which you divide up the data. You are trying to create segments of the data using logical filters and comparing the mpg for each segment of the data.

[86]:
```python
# Mean mpg for cars for each number of cylinders
auto.groupby(by='cylinders').mpg.mean()

# Mpg usually rose over the years contained in this dataset
auto.groupby(by='model_year').mpg.mean()
```

```
# The mpg for the group of lighter cars vs the group of heavier cars
# We can divide the dataset in half by the median (the lower half being the
# lighter cars and the upper half being the heavier cars).
auto[auto.weight <= auto.weight.median()].mpg.mean() # light cars mean mpg
auto[auto.weight > auto.weight.median()].mpg.mean() # heavier cars mean mpg
# It appears that the lighter cars get better gas mileage than the heavier cars

# This question was pretty open-ended, but here are some other things you could
 ↪have looked at

# The average mpg for the four quartiles of displacement
# We didn't talk about the 'quantile' function in class, but it's a useful one!
auto[auto.displacement <= auto.displacement.quantile(0.25)].mpg.mean()
auto[(auto.displacement > auto.displacement.quantile(0.25)) & (auto.
 ↪displacement <= auto.displacement.quantile(0.50))].mpg.mean()
auto[(auto.displacement > auto.displacement.quantile(0.50)) & (auto.
 ↪displacement <= auto.displacement.quantile(0.75))].mpg.mean()
auto[auto.displacement > auto.displacement.quantile(0.75)].mpg.mean()
# It appears that as engine displacement (size) increases, the average mpg
 ↪decreases.  This makes sense.

# Instead of using the somewhat complicated logic of the 'quantile', you can
 ↪easily divide your dataset
# into buckets using the `cut` function.
auto.groupby(pd.cut(auto.horsepower,5)).mpg.mean()
# It appears that as horsepower increases, the average mpg decreases.  This
 ↪makes sense.

auto.groupby(pd.cut(auto.acceleration, 5)).mpg.mean()
# It appears that as acceleration increases, the average mpg increases.
```

[86]:
```
acceleration
(7.983, 11.36]    15.185714
(11.36, 14.72]    20.879259
(14.72, 18.08]    25.243195
(18.08, 21.44]    25.637500
(21.44, 24.8]     31.945455
Name: mpg, dtype: float64
```

Let's now look how MPG has changed over time, while also considering how specific groups have changed– look at low, mid, and high power cars based upon their horsepower and see how these groups have changed over time.

In his data, he called the original dataset 'auto'.

Now to look at how efficency has changed over time based on power and weight classes, two things that we know play a large role in gas mileage. First, we create a table of

efficeincy by power class and year.

[87]:
```
# Now to see efficiency change over time based on power and weight classes, two␣
↪things
# play a role in gas mileage. First, create a table of efficiency by power␣
↪class and year.
horsey = pd.DataFrame()

# Defines low power as below 100 horsepower
horsey['low_power'] = auto[(auto.horsepower < 100)].groupby('model_year').mpg.
↪mean()

# Defines mid-power as between 100 and 150 (inclusive) horsepower
horsey['mid_power'] = auto[(auto.horsepower >= 100) & (auto.horsepower <= 150)].
↪groupby('model_year').mpg.mean()

# Defines high power as above 150 horsepower
horsey['high_power'] = auto[auto.horsepower > 150].groupby('model_year').mpg.
↪mean()
```

[88]:
```
horsey
```

[88]:

model_year	low_power	mid_power	high_power
70	23.300000	18.333333	13.076923
71	26.357143	17.285714	13.333333
72	23.500000	15.000000	12.857143
73	22.166667	16.352941	12.727273
74	27.312500	15.500000	NaN
75	22.470588	17.500000	16.000000
76	25.750000	17.071429	15.500000
77	28.433333	18.100000	15.666667
78	28.363158	19.350000	17.700000
79	29.225000	20.266667	16.900000
80	34.516667	28.100000	NaN
81	31.372727	25.833333	NaN
82	32.607143	23.500000	NaN

[89]:
```
auto["power_class"]=None # create column with default vals
auto.loc[auto["horsepower"] < 100,"power_class"] = "low" # set those that are␣
↪low
auto.head(20)
```

[89]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	\
0	18.0	8	307.0	130	3504	12.0	
1	15.0	8	350.0	165	3693	11.5	
2	18.0	8	318.0	150	3436	11.0	

3	16.0	8	304.0	150	3433	12.0
4	17.0	8	302.0	140	3449	10.5
5	15.0	8	429.0	198	4341	10.0
6	14.0	8	454.0	220	4354	9.0
7	14.0	8	440.0	215	4312	8.5
8	14.0	8	455.0	225	4425	10.0
9	15.0	8	390.0	190	3850	8.5
10	15.0	8	383.0	170	3563	10.0
11	14.0	8	340.0	160	3609	8.0
12	15.0	8	400.0	150	3761	9.5
13	14.0	8	455.0	225	3086	10.0
14	24.0	4	113.0	95	2372	15.0
15	22.0	6	198.0	95	2833	15.5
16	18.0	6	199.0	97	2774	15.5
17	21.0	6	200.0	85	2587	16.0
18	27.0	4	97.0	88	2130	14.5
19	26.0	4	97.0	46	1835	20.5

	model_year	origin	car_name	power_class
0	70	1	chevrolet chevelle malibu	None
1	70	1	buick skylark 320	None
2	70	1	plymouth satellite	None
3	70	1	amc rebel sst	None
4	70	1	ford torino	None
5	70	1	ford galaxie 500	None
6	70	1	chevrolet impala	None
7	70	1	plymouth fury iii	None
8	70	1	pontiac catalina	None
9	70	1	amc ambassador dpl	None
10	70	1	dodge challenger se	None
11	70	1	plymouth 'cuda 340	None
12	70	1	chevrolet monte carlo	None
13	70	1	buick estate wagon (sw)	None
14	70	3	toyota corona mark ii	low
15	70	1	plymouth duster	low
16	70	1	amc hornet	low
17	70	1	ford maverick	low
18	70	3	datsun pl510	low
19	70	2	volkswagen 1131 deluxe sedan	low