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

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

Resources for python

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

Learning resources

- Interactive Python (http://interactivepython.org/) An online book that includes embedded live excercises. Fun!
- Dive Into Python (http://www.diveintopython.net/) An excellent, thorough book.
- tutorial point (http://www.tutorialspoint.com/python/index.htm) 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 (https://docs.python.org/3/library/)
- Quick Reference from Tutorial Point (http://www.tutorialspoint.com/python/python_quick_guide.htm)

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).

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

In [2]:
```
# instantiate empty containers
my_list = []
my_tuple = ()
my_string = ''
my_dict = {}

# add single elements
elem = 10
my_list.append(elem)
my_tuple = ('value')
my_string = 'value'
my_dict.update({elem : 'value'})
```

c) Provide 1 or more use cases for each.

List: This can be used to store the names of companies in the S&P500.
Tuple: This can be used to store different types of variables, for example, I am entering a stock order that includes the ticker (string), price (float), size (int), and other information. The tuple can store all these different types in one bundle.
String: This data struct can store a stock ticker or my name.
Dictionary: This can be used to have all the names of people in this class (first and last to avoid collisions), paired with their GPA's and or ages.

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

a) Do it using a for loop

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

In [4]:
```
def sum_forloop(n):
    """
    Function that returns the sum of all numbers in range [0, n].
    :param n: upper limit of summation, which is positive integer greater than 0
    :return: Sum from i=0 to i=n.
    """
    if type(n) is not int or n < 0:
        # check input meets conditions
        return None
    sum = 0
    for i in range(n+1):
        sum += i
    return sum

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.

In [5]:
```
sum([x for x in range(i+1)])
```

Out[5]:
```
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).

In [6]:
```
my_dict = {'CS135': 'Machine Learning', 'EE129': 'Computer Communication Networks'}

for CRN, class_name in my_dict.items():
    print(f'{CRN = }, {class_name = }')
```

```
CRN = 'CS135', class_name = 'Machine Learning'
CRN = 'EE129', class_name = 'Computer Communication Networks'
```

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)

In [7]:
```
import re

my_list = []
new_dict = {}

for CRN, course_name in my_dict.items():
    time_of_day = str(input(f"Enter the time of day in military format for class '{CRN}': "))
    new_dict[CRN] = time_of_day

for CRN, time in new_dict.items():
    if not re.match(r'^([0-9]|[0-1][0-9]|[2][0-3]):([0-5][0-9])(:[0-5][0-9])?$', time):
        print(f'invalid time {time} entered for class {CRN}')
```

```
Enter the time of day in military format for class 'CS135': 10:30
Enter the time of day in military format for class 'EE129': 10:30
```

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

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

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


temp_c = 60
temp_f = 45

temp_c_out = fahrenheit2celsius(temp_f)
temp_f_out = fahrenheit2celsius(temp_c)

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

```
45 deg. F is 7.222222222222222 deg. C
60 deg. C is 15.555555555555555 deg. F
```

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

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

In [9]:
```
for i in [1,2,3,4,5,4,3,2,1]:
    out = []
    count = i
    for j in range(count):
        if count % 2 == 1:
            out.append('O')
        else:
            out.append('X')
        count -= 1
    if i % 2 == 0:
        out.reverse()
    output = ''.join(out)
    print(output)
```

```
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

In [10]:
```
month = str(input('Please enter the month: ')).strip()
day = int(input('Please enter the day: '))

season_dict = {}
season_dict['winter'] = [['December', 21, 31],  ['January', 31], ['February', 29], ['March', 1, 19]]     # Dec 21 - Mar 19
season_dict['spring'] = [['March', 20, 31],     ['April', 30],   ['May', 31],      ['June', 1, 20]]      # Mar 20 - Jun 20
season_dict['summer'] = [['June', 21, 30],      ['July', 31],    ['August', 31],   ['September', 1, 22]] # Jun 21 - Sep 22
season_dict['autumn'] = [['September', 23, 30], ['October', 31], ['November', 30], ['December', 1, 20]]  # Sep 23 - Dec 20

mths = ['January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December']

def bounds_check():
    error = False

    if day < 1 or day > 31:  # always true
        print(f'Day {day} out of bounds for month {month}'); error = True
    if month not in mths:
        print(f'Month {month} not entered correctly'); error = True

    # need to check days for overlap months
    if not error:
        for season, months in season_dict.items():
            # 4th month in a season doesn't matter here as upper bound is lower
            if month in months[0][0] and day > months[0][2]:
                print(f'Day {day} out of bounds for month {month}'); error = True
            elif (month in months[1][0] and day > months[1][1]) or (month in months[2][0] and day > months[2][1]):
                print(f'Day {day} out of bounds for month {month}'); error = True

    return error

def main(error=False):
    if not error:
        for season, months in season_dict.items():
            # easy middle months cases
            if month in months[1][0] and day <= months[1][1]:
                print(f'Season is {season}')
            elif month in months[2][0] and day <= months[2][1]:
                print(f'Season is {season}')

            # overlap months
            if month in months[0][0] and day >= months[0][1] and day <= months[0][2]:
                print(f'Season is {season}')
            elif month in months[3][0] and day <= months[3][2]:
                print(f'Season is {season}')

error = bounds_check()
main(error)
```

```
Please enter the month: March
Please enter the day: 30
Season is spring
```

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

In [11]:
```
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 '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 neighbors, i.e., [..., 'val', 'val', ...] = True; else, False
    """
    prev = None
    for i in range(len(xs)):
        if i == 0: # first elem
            prev = xs[i]
            continue
        else:
            if (xs[i] == prev) and (xs[i] == val):  # 1) curr equal to prev; 2) curr equal to val
                return True
            prev = xs[i] # set prev to curr before next iteration

    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
```

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

Import other libraries, such that numpy library is called by with np and pandas with pd

In [12]:
```
import os
import numpy as np
```

a) Numpy Basics

*Make sure to leave random seeds in each cell so that the outputs match the expected answer.

1)

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

In [13]:
```
np.random.seed(123)
arr = np.random.rand(10, 10)
print(f'{arr = }\n\n{arr.max() = }\n{arr.min() = }')
```

```
arr = array([[0.69646919, 0.28613933, 0.22685145, 0.55131477, 0.71946897,
        0.42310646, 0.9807642 , 0.68482974, 0.4809319 , 0.39211752],
       [0.34317802, 0.72904971, 0.43857224, 0.0596779 , 0.39804426,
        0.73799541, 0.18249173, 0.17545176, 0.53155137, 0.53182759],
       [0.63440096, 0.84943179, 0.72445532, 0.61102351, 0.72244338,
        0.32295891, 0.36178866, 0.22826323, 0.29371405, 0.63097612],
       [0.09210494, 0.43370117, 0.43086276, 0.4936851 , 0.42583029,
        0.31226122, 0.42635131, 0.89338916, 0.94416002, 0.50183668],
       [0.62395295, 0.1156184 , 0.31728548, 0.41482621, 0.86630916,
        0.25045537, 0.48303426, 0.98555979, 0.51948512, 0.61289453],
       [0.12062867, 0.8263408 , 0.60306013, 0.54506801, 0.34276383,
        0.30412079, 0.41702221, 0.68130077, 0.87545684, 0.51042234],
       [0.66931378, 0.58593655, 0.6249035 , 0.67468905, 0.84234244,
        0.08319499, 0.76368284, 0.24366637, 0.19422296, 0.57245696],
       [0.09571252, 0.88532683, 0.62724897, 0.72341636, 0.01612921,
        0.59443188, 0.55678519, 0.15895964, 0.15307052, 0.69552953],
       [0.31876643, 0.6919703 , 0.55438325, 0.38895057, 0.92513249,
        0.84167   , 0.35739757, 0.04359146, 0.30476807, 0.39818568],
       [0.70495883, 0.99535848, 0.35591487, 0.76254781, 0.59317692,
        0.6917018 , 0.15112745, 0.39887629, 0.2408559 , 0.34345601]])

arr.max() = 0.9953584820340174
arr.min() = 0.01612920669501683
```

2)

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

In [14]:
```
np.random.seed(123)
Z = np.random.uniform(0, 10, 10)

print(f'{Z = }\n')

m1 = np.array(Z, int)
# m2 = [int(x) for x in Z]
m2 = Z-Z%1
m3 = Z.astype(int)
m4 = np.floor(Z)
m5 = np.trunc(Z)

for method in [m1, m2, m3, m4, m5]:
    print(f'{method = }')
```

```
Z = array([6.96469186, 2.86139335, 2.26851454, 5.51314769, 7.1946897 ,
       4.2310646 , 9.80764198, 6.84829739, 4.80931901, 3.92117518])

method = array([6, 2, 2, 5, 7, 4, 9, 6, 4, 3])
method = array([6., 2., 2., 5., 7., 4., 9., 6., 4., 3.])
method = array([6, 2, 2, 5, 7, 4, 9, 6, 4, 3])
method = array([6., 2., 2., 5., 7., 4., 9., 6., 4., 3.])
method = array([6., 2., 2., 5., 7., 4., 9., 6., 4., 3.])
```

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

In [167]:
```
# np.random.uniform(0, 1, size=20)
# vect = np.linspace(start=0, stop=1, endpoint=False, num=20)

# x0 = np.linspace(start=0, stop=1, num=20, endpoint=True) # has 0 and 1
# x1 = np.linspace(start=0, stop=1, num=20, endpoint=False) # has 0
# x2 = np.linspace(start=0, stop=1, num=20, endpoint=True)[1:-1] # only 18 values
# x3 = np.linspace(start=0, stop=1, num=20, endpoint=False)[1:-1] # only 18 values
x4 = np.linspace(start=0, stop=1, num=22, endpoint=True)[1:-1] # starts with 22 vals, snips off 0,1
x5 = np.linspace(start=0, stop=1, num=21, endpoint=False)[1:] # starts with 21 vals, snips off 0

# print(f'{x0}\n{x1}\n{x2}\n{x3}\n')
print(f'{x4}\n{x5}\n')
# https://www.w3resource.com/python-exercises/numpy/python-numpy-exercise-66.php
```

```
[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]
[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

In [16]:
```
np.random.seed(123)
vect = np.random.rand(15)
print(vect)
sorted = np.sort(vect)
print(sorted)
```

```
[0.69646919 0.28613933 0.22685145 0.55131477 0.71946897 0.42310646
 0.9807642  0.68482974 0.4809319  0.39211752 0.34317802 0.72904971
 0.43857224 0.0596779  0.39804426]
[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 and B, check if they are equal

In [17]:
```
np.random.seed(123)
A = np.random.randint(0, 2, 5)
B = np.random.randint(0, 2, 5)
```

In [18]:
```
# np.equal(A, B) # elem by elem
np.array_equal(A, B) # entire array comparison
# np.allclose(A, B) # another option
```

Out[18]:
```
False
```

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 (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.plot.html) if you get stuck!

Make an array immutable (read-only)

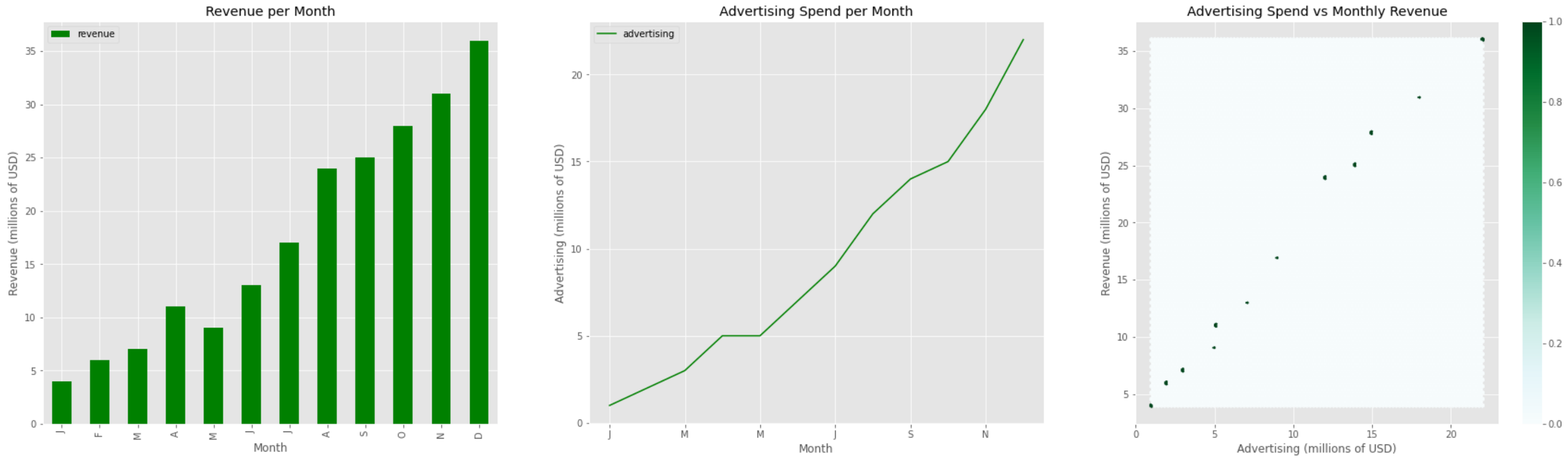
In [19]:
```
Z = np.zeros(10)
Z.flags.writeable = False
```

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)

In [20]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
plt.style.use('ggplot')

money = {'month' : ['J', 'F', 'M', 'A', 'M', 'J', 'J', 'A', 'S', 'O', 'N', 'D'],
         'revenue' : [4, 6, 7, 11, 9, 13, 17, 24, 25, 28, 31, 36],
         'advertising' : [1, 2, 3, 5, 5, 7, 9, 12, 14, 15, 18, 22]}

df = pd.DataFrame(money); # print(df)
# df.plot(x='revenue', y='advertising', xlabel='revenue (millions)', ylabel='advertising (millions)', color='green', legend=None, title='Advertising Spend vs Monthly Revenue')
# plt.show()

fig, axes = plt.subplots(1, 3, figsize=(30, 8))
df.plot(ax=axes[0], x='month', y='revenue', kind='bar', color='green', xlabel='Month', ylabel='Revenue (millions of USD)', title='Revenue per Month')
df.plot(ax=axes[1], x='month', y='advertising', kind='line', color='green', xlabel='Month', ylabel='Advertising (millions of USD)', title='Advertising Spend per Month')
df.plot(ax=axes[2], x='advertising', y='revenue', kind='hexbin', xlabel='Advertising (millions of USD)', ylabel='Revenue (millions of USD)', title='Advertising Spend vs Monthly Revenue')
```

Out[20]: <matplotlib.axes._subplots.AxesSubplot at 0x7f238397a100>



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).

In [21]:
```
# arr = [ ( 'position', [('x', float), ('y', float)] ),
#         ( 'color',    [('r', float), ('g', float), ('b', float)] ) ]
# zeroed = np.zeros(10, arr)
# print(zeroed)

arr_np = np.array([((0,0), (0,0,0))],
         dtype = [ ( 'position', [('x', np.float16), ('y', np.float16)] ),
                   ( 'color',    [('r', np.float16), ('g', np.float16), ('b', np.float16)] ) ] )

print(arr_np)
```

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

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

In [22]:
```
np.random.seed(123)
A = np.random.randint(0, 10, (3, 4, 3, 4))
sum = A.sum(axis=(-2,-1))
sum
```

Out[22]:
```
array([[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

Each color is represented by 3 bits, so the output should be a list of elements, each with 3 bits (all unique permutations of 0s and 1s).

Can you convert colors? How about unique across a different axis?

In [23]:
```
np.random.seed(123)
w, h = 16, 16
I = np.random.randint(0, 2, (h, w, 3)).astype(np.ubyte)
I.shape

nums_list = []
for matrix in I:
    for bits3 in matrix:
        decimal = bits3[2] * 4 + bits3[1] * 2 + bits3[0]
        nums_list.append(decimal)

print(f'we have {len(list(set(nums_list)))} unique colors in this image\n')

E = len(np.unique(nums_list)) # numpy version
# E

# 3 bits = 8 permutations -> will almost always be 8 but theoretically could be less based on random seed
# 16 rows * 8 perms
```

```
we have 8 unique colors in this image
```

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

In [24]:
```
X = [1, 2, 3, 4, 5, 6]
I = [1, 3, 9, 3, 4, 1]
F = np.bincount(I, X)
F
```

Out[24]:
```
array([0., 7., 0., 6., 5., 0., 0., 0., 0., 3.])
```

How to read the following file?

In [25]:
```
fpath = os.path.join("data", "tabular", "missing.dat")
with open(fpath, 'r') as fi:
    # fi.readlines()
    for line in fi:
        print(line)
```

```
# File content:

# -------------

1,2,3,4,5

6,,,7,8

,,9,10,11

# -------------
```

Convert a vector of ints into a matrix binary representation

In [26]:
```
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(f'{I}\n{B}')
```

```
[  0   1   2   3  15  16  32  64 128]
[[0 0 0 0 0 0 0 0]
 [1 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0]
 [1 1 0 0 0 0 0 0]
 [1 1 1 1 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 1]]
```

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

In [27]:
```
Z = np.random.randint(0, 2, (6, 3))
print(Z)
np.unique(Z, axis=0)
```

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

Out[27]:
```
array([[0, 0, 0],
       [0, 1, 0],
       [0, 1, 1]])
```

Pandas

Made-up data representing animals and trips to vet

In [28]:
```
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.

In [29]:
```
df = pd.DataFrame(data=data, index=labels)
df
```

Out[29]:

	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.

In [30]:
```
print(f'{df.describe()}\n{df.info()}\n')
```

```
<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
            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
None
```

Return the first 3 rows of the DataFrame df.

In [31]:
```
df.head(3)
```

Out[31]:

	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.

In [32]:
```
df[['age', 'animal']]
```

Out[32]:

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

Change the age in row 'f' to 1.5.

In [33]:
```
df['age']['f'] = 1.5
df
```

```
<ipython-input-33-133ae7ba59e6>:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df['age']['f'] = 1.5
```

Out[33]:

	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	1.5	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

Calculate the mean age for each different animal in df.

In [34]:
```
df.groupby('animal')['age'].mean()
```

Out[34]:
```
animal
cat      2.333333
dog      5.000000
snake    2.500000
Name: age, dtype: float64
```

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

In [35]:
```
df['animal'] = df['animal'].replace(to_replace='snake', value='python')
df
```

Out[35]:

	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).

In [36]:
```
# df.groupby(['animal', 'visits'])['age'].mean()
df.pivot_table(index="animal", columns="visits", aggfunc="mean")["age"]
```

Out[36]:

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?

In [37]:
```
# a 5x3 frame of float values
df_floats = pd.DataFrame(np.random.random(size=(5, 3)))
print(df_floats)
# df_floats - df_floats.mean()

df_floats.sub(df_floats.mean(axis=1), axis=0)
```

```
          0         1         2
0  0.491190  0.270176  0.360424
1  0.210653  0.421200  0.218035
2  0.845753  0.456271  0.279802
3  0.932892  0.314351  0.909715
4  0.043418  0.707115  0.483889
```

Out[37]:

	0	1	2
0	0.117260	-0.103754	-0.013506
1	-0.072643	0.137904	-0.065261
2	0.318477	-0.071004	-0.247473
3	0.213906	-0.404635	0.190729
4	-0.368056	0.295641	0.072415

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.

In [38]:
```
index = pd.bdate_range(start='2015-01-01', end='2015-12-31')
# s = pd.DataFrame(np.random.rand(index.shape[0]), index=index)
s = pd.Series(np.random.rand(index.shape[0]), index=index)
s
```

Out[38]:
```
2015-01-01    0.444221
2015-01-02    0.036323
2015-01-05    0.040683
2015-01-06    0.332754
2015-01-07    0.947120
                ...
2015-12-25    0.587616
2015-12-28    0.967362
2015-12-29    0.657667
2015-12-30    0.584904
2015-12-31    0.518773
Freq: B, Length: 261, dtype: float64
```

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

In [39]:
```
# np.sum(s[s.index.dayofweek == 2][0])
s[index.weekday == 2].sum()
```

Out[39]:
```
28.20463133931073
```

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

In [40]:
```
s.groupby(s.index.month).mean()
```

Out[40]:
```
1     0.514598
2     0.411958
3     0.482855
4     0.465519
5     0.586244
6     0.459178
7     0.499775
8     0.579787
9     0.509425
10    0.545982
11    0.496362
12    0.549904
dtype: float64
```

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

In [41]:
```
s.groupby(pd.Grouper(freq='4M')).idxmax()  # SHOULD NOT BE SPILLING INTO 2016 !!!!!!!!!!!!!!!!!!!!!!

# idx = pd.bdate_range(start='2015-01-01', end='2015-09-30')
# t = pd.DataFrame(np.random.rand(idx.shape[0]), index=idx)

# t.groupby(pd.Grouper(freq='4M')).idxmax()
```

Out[41]:
```
2015-01-31   2015-01-27
2015-05-31   2015-04-03
2015-09-30   2015-06-10
2016-01-31   2015-11-18
Freq: 4M, dtype: datetime64[ns]
```

Cleaning Data

The DataFrame to use in the following puzzles:

In [42]:
```
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()
```

Out[42]:

	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).

In [43]:
```
prev = ''
arr = []

for entry in df['FlightNumber']:
    value = 0
    if pd.isna(entry):
        value = int(prev + 10)
        arr.append(value)
    else:
        value = int(entry)
        arr.append(value)
    prev = int(value)

df['FlightNumber'] = arr
df
```

Out[43]:

	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.

In [44]:
```
from_col = []
to_col = []
for entry in df['From_To']:
    frm, to = entry.split('_')[0], entry.split('_')[1]
    from_col.append(frm); to_col.append(to)

df_temp = pd.DataFrame()
df_temp['From'] = from_col
df_temp['To'] = to_col

df_temp
```

Out[44]:

	From	To
0	LoNDon	paris
1	MAdrid	miLAN
2	londON	StockhOlm
3	Budapest	PaRis
4	Brussels	londOn

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".)

In [45]:
```
df_temp['From'] = df_temp['From'].str.title()
df_temp['To'] = df_temp['To'].str.title()
# df_temp.columns = ['From', 'To']
df_temp
```

Out[45]:

	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.

In [46]:
```
try:
    df = df.drop('From_To', axis=1)
    df = df.join(df_temp)
except:
    pass

df
```

Out[46]:

	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

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 (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.plot.html) if you get stuck!

In [47]:
```
import matplotlib.pyplot as plt
import pandas as pd
%matplotlib inline
plt.style.use('ggplot')
```

In [48]:
```
df = pd.DataFrame({"xs":[1,5,2,8,1], "ys":[4,2,1,9,6]})
df
```

Out[48]:

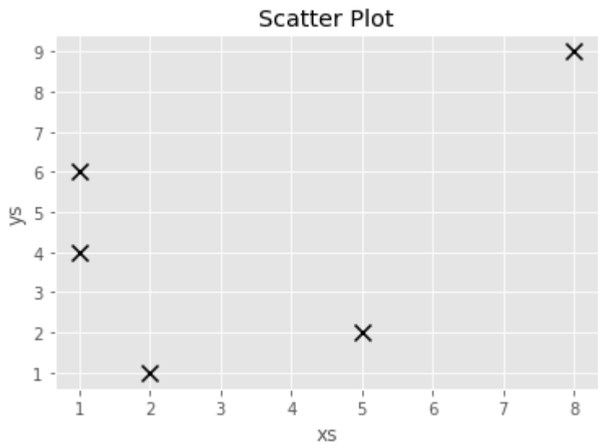
	xs	ys
0	1	4
1	5	2
2	2	1
3	8	9
4	1	6

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

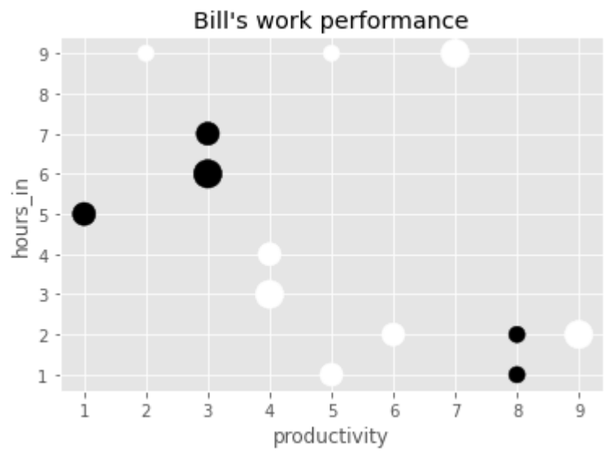
In [49]:
```
df.plot(x='xs', y='ys', kind='scatter', s=100, title='Scatter Plot', color='black', marker='x')
plt.show()
```



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)

In [50]:
```
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(x='productivity', y='hours_in', s=df2.happiness * 100, c=df2.caffienated, kind='scatter', title="Bill's work performance") # s = size of dots, c = color of dots
plt.show()
```



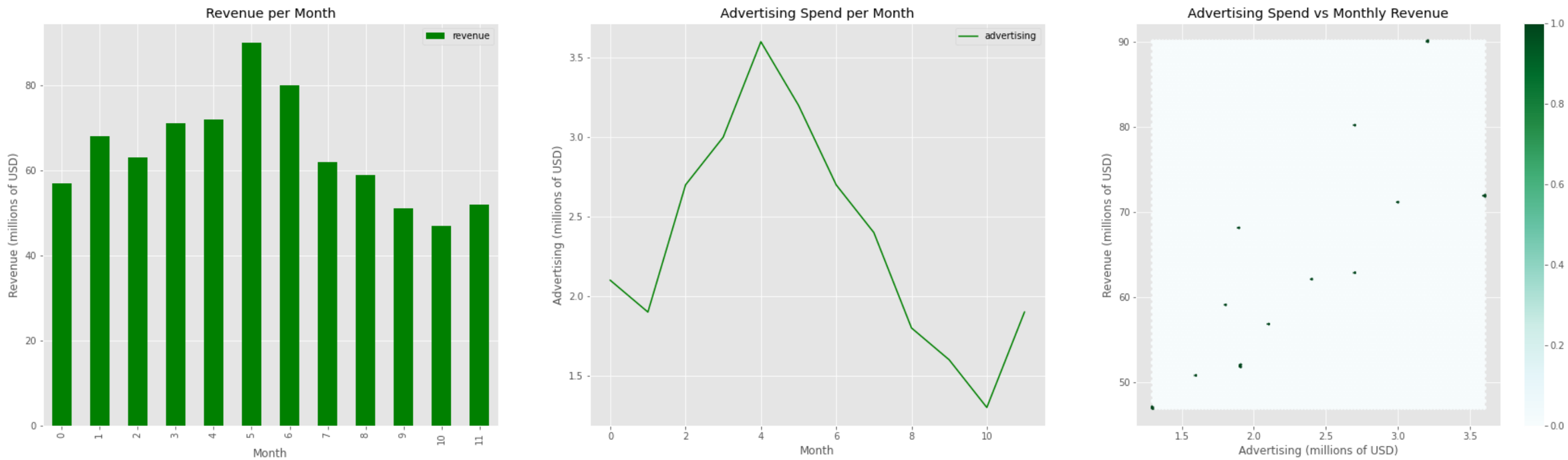
1.33)

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

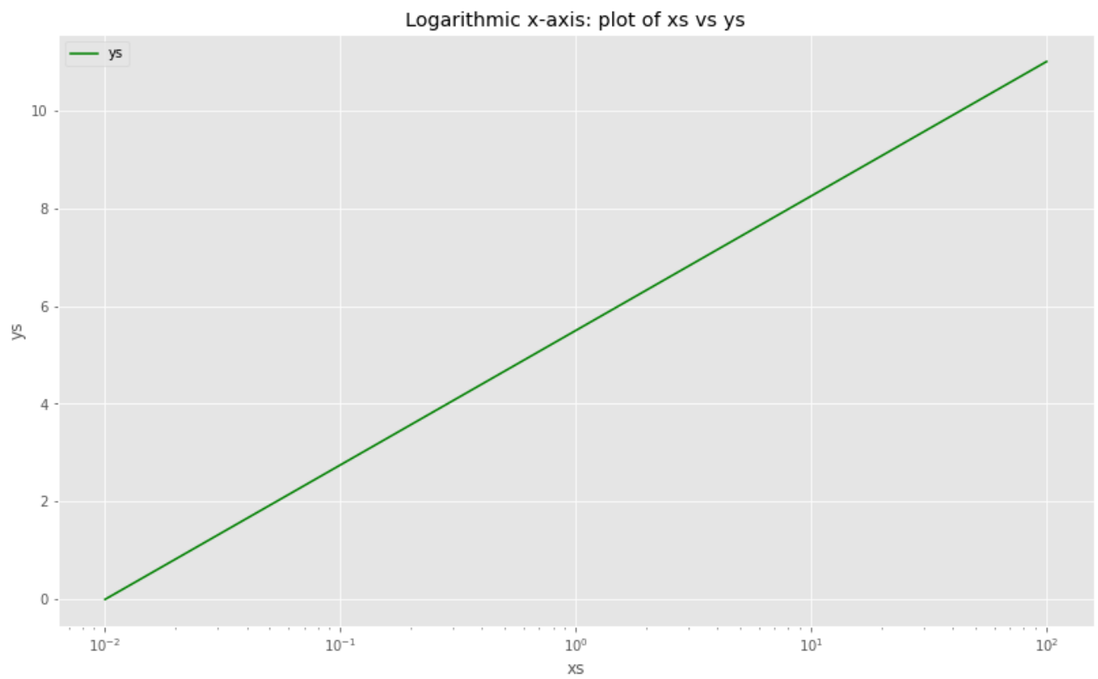
In [51]:
```
df_to_plot = pd.DataFrame({"revenue":[57,68,63,71,72,90,80,62,59,51,47,52],
                   "advertising":[2.1,1.9,2.7,3.0,3.6,3.2,2.7,2.4,1.8,1.6,1.3,1.9],
                   "month":range(12)
                  })

fig, axes = plt.subplots(1, 3, figsize=(30, 8))
df_to_plot.plot(ax=axes[0], x='month', y='revenue', kind='bar', color='green', xlabel='Month', ylabel='Revenue (millions of USD)', title='Revenue per Month')
df_to_plot.plot(ax=axes[1], x='month', y='advertising', kind='line', color='green', xlabel='Month', ylabel='Advertising (millions of USD)', title='Advertising Spend per Month')
df_to_plot.plot(ax=axes[2], x='advertising', y='revenue', kind='hexbin', xlabel='Advertising (millions of USD)', ylabel='Revenue (millions of USD)', title='Advertising Spend vs Monthly Revenue')
plt.show()
```

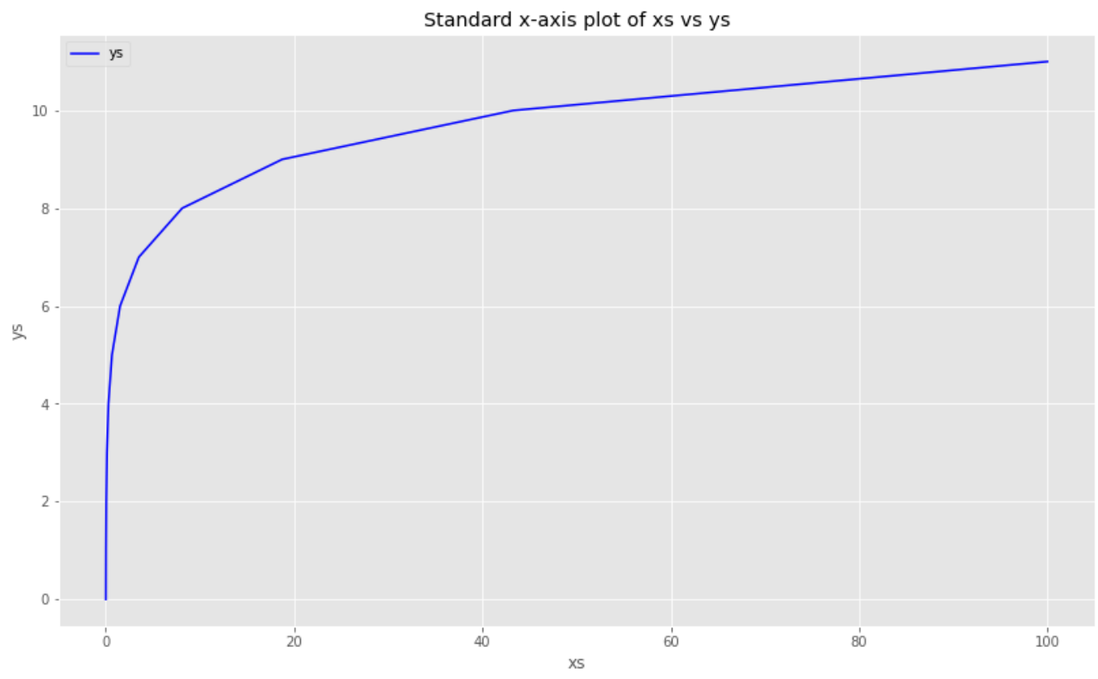


1.33)

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.

In [52]:
```
import numpy as np
df3 = pd.DataFrame({"xs":np.logspace(-2, 2, base=10, num=12),
                    "ys":range(12)
                   })

fig, axes = plt.subplots(1, 2, figsize=(30, 8))
df3.plot(ax=axes[0], x='xs', y='ys', xlabel='xs', ylabel='ys', kind='line', color='green', logx=True, title='Logarithmic x-axis: plot of xs vs ys')
df3.plot(ax=axes[1], x='xs', y='ys', xlabel='xs', ylabel='ys', kind='line', color='blue', title='Standard x-axis plot of xs vs ys')
plt.show()
```





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:

$$Ax = B$$

In [53]:
```
A = np.array([3, 5, 9, 3, 3, 4, 5, 9, 17]).reshape(3,3)
B = np.array([2, 1, 4]).reshape(3,1)
ans = np.linalg.solve(A, B)
print(f'{A}\n{B}\n{ans = }')
```

```
[[ 3  5  9]
 [ 3  3  4]
 [ 5  9 17]]
[[2]
 [1]
 [4]]
ans = array([[ 1.],
       [-2.],
       [ 1.]])
```

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

In [54]:
```
temp_arr = []
count = 0; row_sum = 0

for idxi, i in np.ndenumerate(A): # tuple indexing (0,0) ... (2,2)
    count += 1
    for idxj, j in np.ndenumerate(B): # tuple indexing (0,0) ... (2,0)
        if idxi[1] == idxj[0]:
            row_sum += i*j

    if count % len(B) == 0:
        temp_arr.append(row_sum); row_sum = 0

'''
3*2 + 5*1 + 9*4  = 47
3*2 + 3*1 + 4*4  = 25
5*2 + 9*1 + 4*17 = 87
'''
C1 = np.array(temp_arr).reshape(3,1)
# C2 =  np.multiply(A, B)
C2 =  np.dot(A, B)
C3 =  np.matmul(A, B)
C4 = A@B
print(f'{C1 = }\n{C2 = }\n{C3 = }\n{C4 = }\n')
```

```
C1 = array([[47],
       [25],
       [87]])
C2 = array([[47],
       [25],
       [87]])
C3 = array([[47],
       [25],
       [87]])
C4 = array([[47],
       [25],
       [87]])
```

Part 2

Getting used to the data

In [55]:
```
# Reads text file and uses '|' as separator
auto = pd.read_table('data/tabular/auto_mpg.txt', sep='|')
auto.head()
```

Out[55]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino

Answer the following questions about the data:

a) What is the shape of the data?

In [56]:
```
auto.shape
```

Out[56]:
```
(392, 9)
```

b) How many rows and columns are there?

In [57]:
```
print(f'{auto.shape[0]} rows and {auto.shape[1]} columns')
```

```
392 rows and 9 columns
```

c) What variables are available?

In [58]:
```
auto.columns
```

Out[58]:
```
Index(['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'model_year', 'origin', 'car_name'],
      dtype='object')
```

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

In [59]:
```
maxes = auto.select_dtypes('number').max()
mins = auto.select_dtypes('number').min()
print(f'mins:\n{mins}\n\nmaxes:\n{maxes}')
# auto.select_dtypes('number').max() - auto.select_dtypes('number').min()
```

```
mins:
mpg                9.0
cylinders          3.0
displacement      68.0
horsepower        46.0
weight          1613.0
acceleration       8.0
model_year        70.0
origin             1.0
dtype: float64

maxes:
mpg               46.6
cylinders          8.0
displacement     455.0
horsepower       230.0
weight          5140.0
acceleration      24.8
model_year        82.0
origin             3.0
dtype: float64
```

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

In absolute terms, the displacement and weight values differ a decent amount between the average and the median, with horsepower varying as well. The other categories have very small numerical differences of <= 2 in absolute value.
If we look at percent change between averages and medians; however, cylinders, displacement, and origin are all significantly different at roughly 30% and above.

In [60]:
```
import warnings # to suppress pesky warnings from interfering from reading the analysis \/
warnings.simplefilter(action='ignore', category=FutureWarning)
''' example:
<ipython-input-325-60fbf316ac7b>:1: FutureWarning: Dropping of nuisance columns in DataFrame reductions
 (with 'numeric_only=None') is deprecated; in a future version this will raise TypeError.
 Select only valid columns before calling the reduction.
'''
averages = auto.mean()
medians = auto.median()
diffs = abs(averages - medians)

pct_change = []
for i, j in zip(averages, medians):
    ans = abs(round( ((j-i)/abs(j) * 100), 2))
    pct_change.append(ans)

temp_df = pd.DataFrame(mins).T # index 0
temp_df.loc[1] = maxes
temp_df.loc[2] = averages
temp_df.loc[3] = medians
temp_df.loc[4] = diffs
temp_df.loc[5] = pct_change

temp_df.index = ['mins', 'maxes', 'averages', 'medians', 'abs difference (avg, med)', 'pct change (avg, med)']
temp_df
```

Out[60]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin
mins	9.000000	3.000000	68.00000	46.000000	1613.000000	8.000000	70.000000	1.000000
maxes	46.600000	8.000000	455.00000	230.000000	5140.000000	24.800000	82.000000	3.000000
averages	23.445918	5.471939	194.41199	104.469388	2977.584184	15.541327	75.979592	1.576531
medians	22.750000	4.000000	151.00000	93.500000	2803.500000	15.500000	76.000000	1.000000
abs difference (avg, med)	0.695918	1.471939	43.41199	10.969388	174.084184	0.041327	0.020408	0.576531
pct change (avg, med)	3.060000	36.800000	28.75000	11.730000	6.210000	0.270000	0.030000	57.650000

Answer the following questions about the data:

a) Which 5 cars get the best gas mileage?

In [61]:
```
auto.sort_values('mpg', ascending=False).head(5)
```

Out[61]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
320	46.6	4	86.0	65	2110	17.9	80	3	mazda glc
327	44.6	4	91.0	67	1850	13.8	80	3	honda civic 1500 gl
323	44.3	4	90.0	48	2085	21.7	80	2	vw rabbit c (diesel)
388	44.0	4	97.0	52	2130	24.6	82	2	vw pickup
324	43.4	4	90.0	48	2335	23.7	80	2	vw dasher (diesel)

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

In [62]:
```
auto[auto.cylinders > 4].sort_values(by=['mpg'], ascending=False).head(5)
```

Out[62]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
381	38.0	6	262.0	85	3015	17.0	82	1	oldsmobile cutlass ciera (diesel)
325	36.4	5	121.0	67	2950	19.9	80	2	audi 5000s (diesel)
330	32.7	6	168.0	132	2910	11.4	80	3	datsun 280-zx
355	30.7	6	145.0	76	3160	19.6	81	2	volvo diesel
304	28.8	6	173.0	115	2595	11.3	79	1	chevrolet citation

c) Which 5 cars get the worst gas mileage?

In [63]:
```
auto.sort_values('mpg', ascending=True).head(5)
```

Out[63]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
28	9.0	8	304.0	193	4732	18.5	70	1	hi 1200d
26	10.0	8	307.0	200	4376	15.0	70	1	chevy c20
25	10.0	8	360.0	215	4615	14.0	70	1	ford f250
27	11.0	8	318.0	210	4382	13.5	70	1	dodge d200
123	11.0	8	350.0	180	3664	11.0	73	1	oldsmobile omega

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

In [64]:
```
auto[auto.cylinders <= 4].sort_values(by=['mpg'], ascending=True).head(5)
```

Out[64]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
110	18.0	3	70.0	90	2124	13.5	73	3	maxda rx3
75	18.0	4	121.0	112	2933	14.5	72	2	volvo 145e (sw)
119	19.0	4	121.0	112	2868	15.5	73	2	volvo 144ea
70	19.0	3	70.0	97	2330	13.5	72	3	mazda rx2 coupe
111	19.0	4	122.0	85	2310	18.5	73	1	ford pinto

Part 3

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:

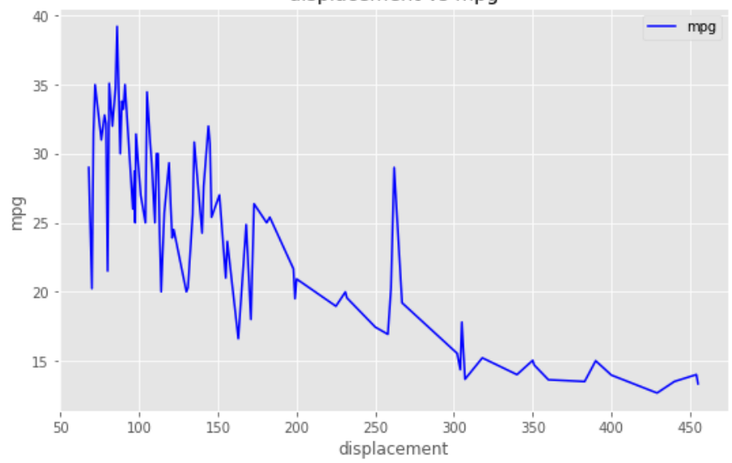
- What is the mean mpg for cars for each number of cylinders?
  (i.e. 3 cylinders, 4 cylinders, 5 cylinders, etc)?

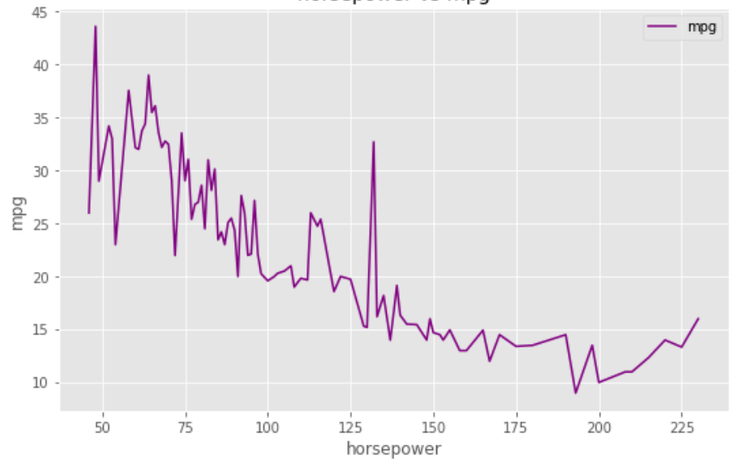
In [65]:
```
# mean mpg for each amount of cylinders
dataf = pd.DataFrame(auto.groupby('cylinders')['mpg'].mean())
dataf.reset_index(inplace=True)
dataf.columns = ['cylinders', 'mpg (mean)']
dataf
```

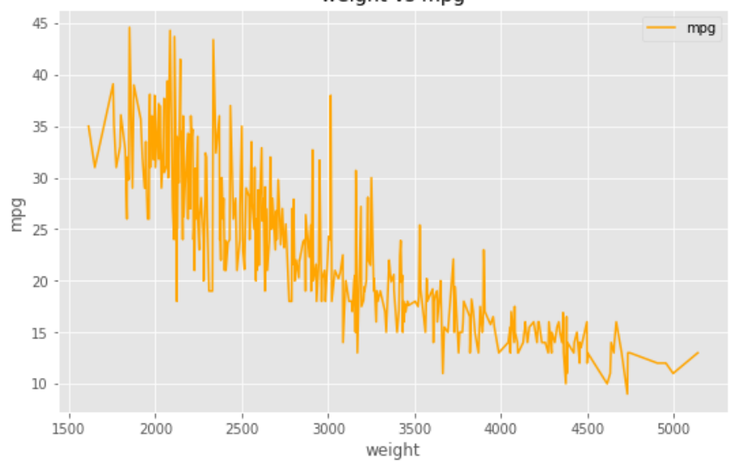
Out[65]:

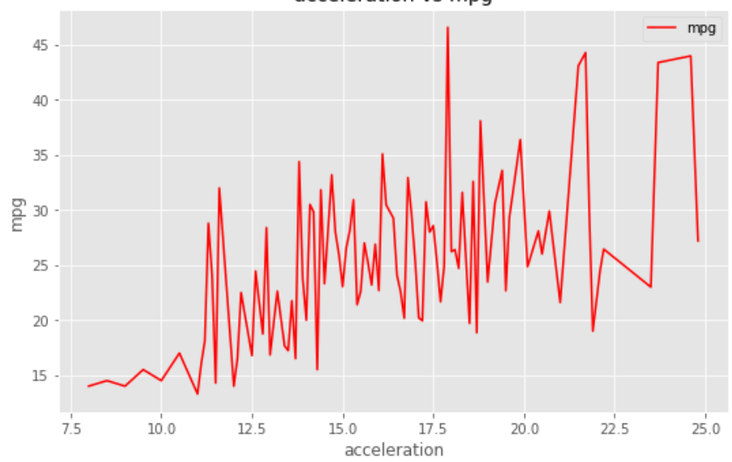
	cylinders	mpg (mean)
0	3	20.550000
1	4	29.283920
2	5	27.366667
3	6	19.973494
4	8	14.963107

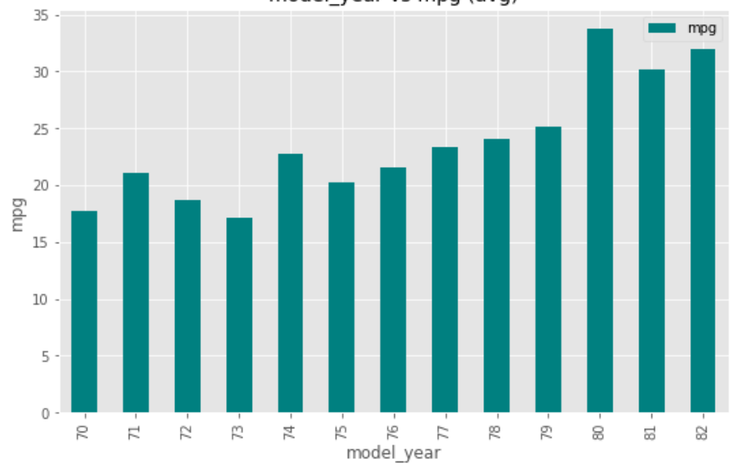
In [66]:
```
import matplotlib.pyplot as plt
import seaborn as sns

# auto.info()
# tell us the number of distinct values and their means per category vs 'mpg'
for count, col in enumerate(auto.columns[1:7]):
    pass # print(col, count)
    # print(auto.groupby(col)['mpg'].mean())

# ['cylinders', 'displacement', 'horsepower', 'weight', 'acceleration', 'model_year'] # categories we care about vs 'mpg' for graphing

# will want to bin data with many rows ['displacement:81', 'horsepower:93', 'weight:346', 'acceleration:95',  ]
# no need to bin 'cylinders:5' and 'model_year:13'

# mpg_cyl = pd.DataFrame(data=auto.groupby('cylinders')['mpg'].mean(), columns=['cylinders', 'mpg'])
mpg_cyl = pd.DataFrame(data=auto.groupby('cylinders')['mpg'].mean())
mpg_dpl = pd.DataFrame(data=auto.groupby('displacement')['mpg'].mean())
mpg_hpw = pd.DataFrame(data=auto.groupby('horsepower')['mpg'].mean())
mpg_wgt = pd.DataFrame(data=auto.groupby('weight')['mpg'].mean())
mpg_acc = pd.DataFrame(data=auto.groupby('acceleration')['mpg'].mean())
mpg_myr = pd.DataFrame(data=auto.groupby('model_year')['mpg'].mean())

for datafr in [mpg_cyl, mpg_dpl, mpg_hpw, mpg_wgt, mpg_acc, mpg_myr]:
    datafr.reset_index(inplace=True)

%matplotlib inline
plt.style.use('ggplot')

fig, axes = plt.subplots(2, 3, figsize=(30, 12))
mpg_cyl.plot(ax=axes[0, 0], x='cylinders',    y='mpg', kind='bar',  xlabel='cylinders',    ylabel='mpg', title='cylinders vs mpg (avg)',    color='green')
mpg_dpl.plot(ax=axes[0, 1], x='displacement', y='mpg', kind='line', xlabel='displacement', ylabel='mpg', title='displacement vs mpg', color='blue')
mpg_hpw.plot(ax=axes[0, 2], x='horsepower',   y='mpg', kind='line', xlabel='horsepower',   ylabel='mpg', title='horsepower vs mpg',   color='purple')
mpg_wgt.plot(ax=axes[1, 0], x='weight',       y='mpg', kind='line', xlabel='weight',       ylabel='mpg', title='weight vs mpg',       color='orange')
mpg_acc.plot(ax=axes[1, 1], x='acceleration', y='mpg', kind='line', xlabel='acceleration', ylabel='mpg', title='acceleration vs mpg', color='red')
mpg_myr.plot(ax=axes[1, 2], x='model_year',   y='mpg', kind='bar',  xlabel='model_year',   ylabel='mpg', title='model_year vs mpg (avg)',   color='teal')

plt.show()
```











How has mpg trended over the years?
Over the years, average mpg has trended upwarded (increased). See plot 6 of 6.

In [142]:
```
auto.sort_values('model_year', inplace=True)
auto # https://datatofish.com/sort-pandas-dataframe/
```

Out[142]:

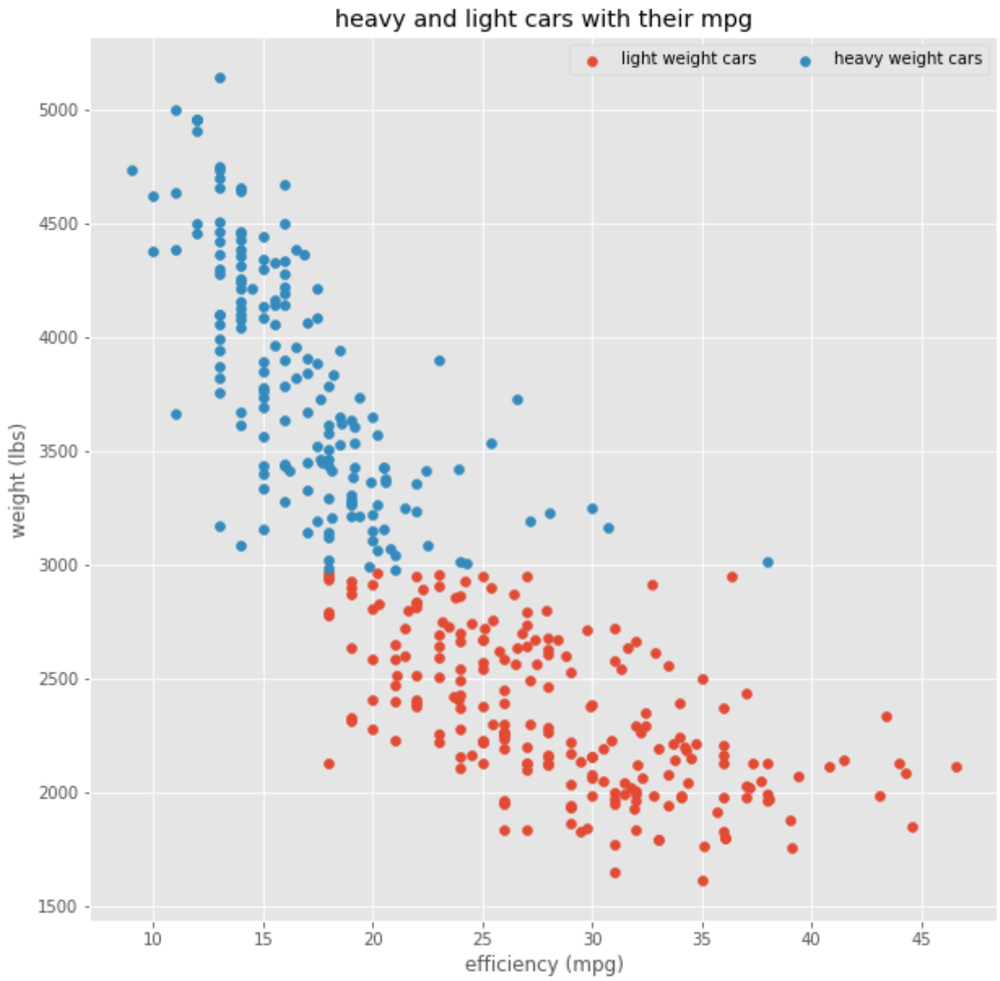
	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin	car_name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
7	14.0	8	440.0	215	4312	8.5	70	1	plymouth fury iii
12	15.0	8	400.0	150	3761	9.5	70	1	chevrolet monte carlo
11	14.0	8	340.0	160	3609	8.0	70	1	plymouth 'cuda 340
10	15.0	8	383.0	170	3563	10.0	70	1	dodge challenger se
...	...	...	...	...	...	...	...	...	...
379	38.0	4	91.0	67	1995	16.2	82	3	datsun 310 gx
378	32.0	4	91.0	67	1965	15.7	82	3	honda civic (auto)
382	26.0	4	156.0	92	2585	14.5	82	1	chrysler lebaron medallion
376	34.0	4	108.0	70	2245	16.9	82	3	toyota corolla
391	31.0	4	119.0	82	2720	19.4	82	1	chevy s-10

392 rows × 9 columns

What is the mpg for the group of lighter cars vs the group of heavier 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.

The mpg for the lighter cars is typically much higher than for heavier cars, as we would expect.

In [147]:
```
# From chart above:
# Car weight range: (1613 - 5140 pounds)
# Car weight average (2977.58 pounds)
# Car weight median: (2803.5 pounds)

# https://stackoverflow.com/questions/17411940/matplotlib-scatter-plot-legend

light = pd.DataFrame(auto[auto.weight <= auto['weight'].mean()].sort_values(by=['model_year'], ascending=True), columns=['model_year', 'weight', 'mpg'])
heavy = pd.DataFrame(auto[auto.weight > auto['weight'].mean()].sort_values(by=['model_year'], ascending=True), columns=['model_year', 'weight', 'mpg'])
# print(f'{light}\n{heavy}\n')

g0 = plt.scatter(x=light['mpg'], y=light['weight'])
g1 = plt.scatter(x=heavy['mpg'], y=heavy['weight'])
plt.legend((g0, g1), ('light weight cars', 'heavy weight cars'),
           scatterpoints=1, loc='upper right', ncol=3, fontsize=10)
plt.gca().update(dict(xlabel='efficiency (mpg)', ylabel='weight (lbs)', title='heavy and light cars with their mpg'))
plt.show()
```



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.

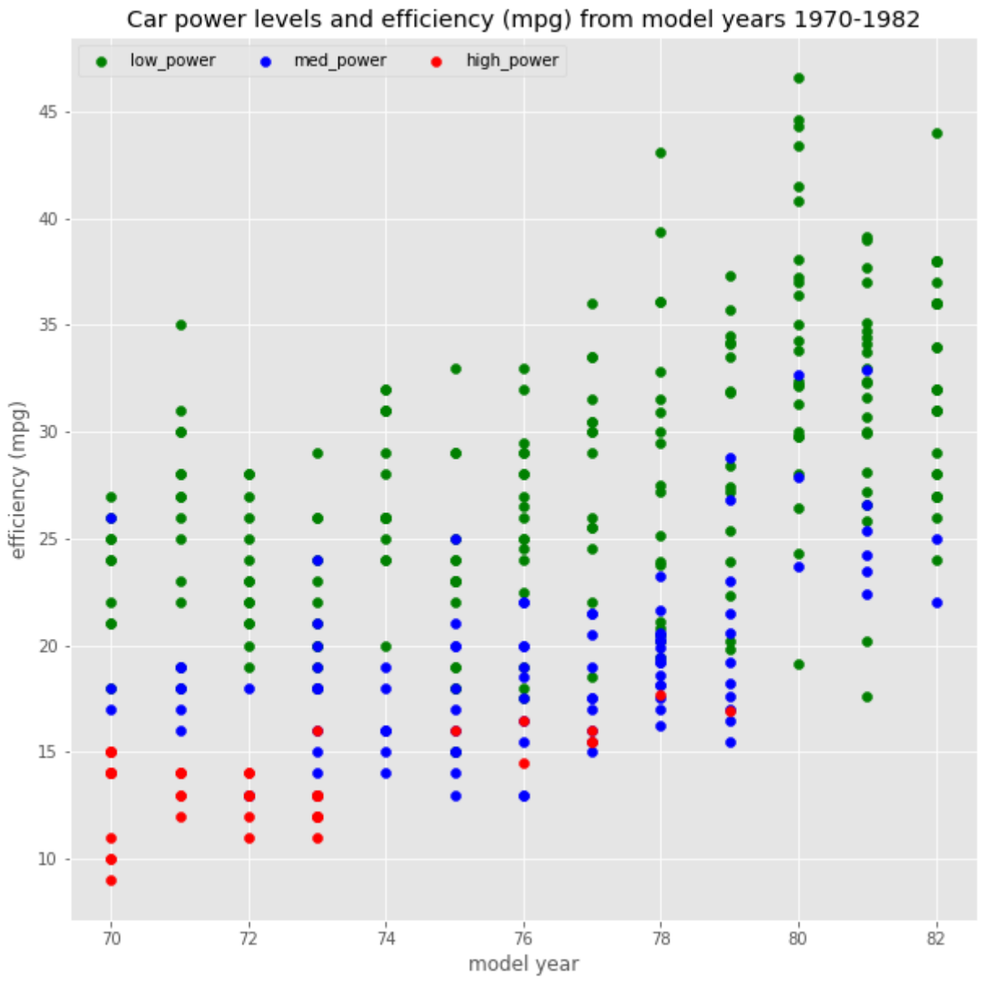
Defines low power as below 100 horsepower
Defines mid power as between 100 and 150 (inclusive) horsepower
Defines high power as above 150 horsepower

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 efficiency by power class and year.

In [148]:
```
# MY, HP, MPG in that priority (low to high) sort...power levels defined above
low_power = auto[auto["horsepower"] < 100].sort_values(by=['model_year', 'horsepower', 'mpg'], axis=0, ascending=[True, True, True])
med_power = auto[(auto["horsepower"] >= 100) & (auto["horsepower"] < 150)].sort_values(by=['model_year', 'horsepower', 'mpg'], axis=0, ascending=[True, True, True])
high_power = auto[auto["horsepower"] > 150].sort_values(by=['model_year', 'horsepower', 'mpg'], axis=0, ascending=[True, True, True])

# high_power
```

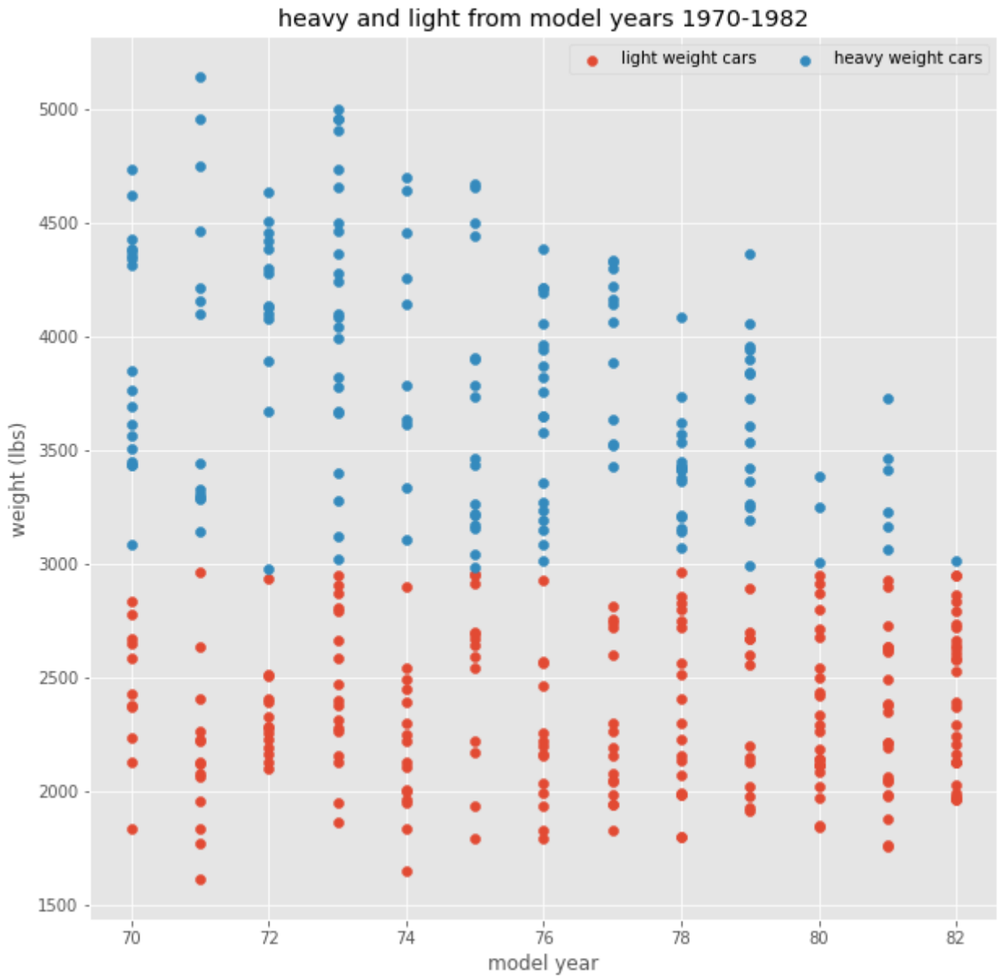
In [149]:
```
p1 = plt.scatter(x=low_power['model_year'], y=low_power['mpg'], color='green')
p2 = plt.scatter(x=med_power['model_year'], y=med_power['mpg'], color='blue')
p3 = plt.scatter(x=high_power['model_year'], y=high_power['mpg'], color='red')

plt.legend((p1, p2, p3), ('low_power', 'med_power', 'high_power'),
           scatterpoints=1, loc='upper left', ncol=3, fontsize=10)
plt.gca().update(dict(xlabel='model year', ylabel='efficiency (mpg)', title='Car power levels and efficiency (mpg) from model years 1970-1982'))
plt.rcParams["figure.figsize"] = (10, 10)
plt.show()
```



We see here that efficiency trends upwards based on model year in the low horsepower category.
There are no cars at the most recent years (1980-1982) so it is harder to draw a conclusion of mpg trends. It appears as if the worst mpg cars for the high HP category only existing from 1970-1973 with overall mpg averages improving in the 1970s for the cars in the dataset.
The medium HP cars improved in the 1980s, but there are less of them in that time period than the 1970s for each year. The peaks of mpg's absolutely improved for medium HP cars from 1979-1983.

In [171]:
```
light_weight = auto[auto.weight <= auto['weight'].mean()].sort_values(by=['model_year', 'weight'], ascending=[True, True])
heavy_weight = auto[auto.weight > auto['weight'].mean()].sort_values(by=['model_year', 'weight'], ascending=[True, True])

# heavy_weight
```

In [172]:
```
k0 = plt.scatter(x=light['model_year'], y=light['weight'])
k1 = plt.scatter(x=heavy['model_year'], y=heavy['weight'])
plt.legend((k0, k1), ('light weight cars', 'heavy weight cars'),
           scatterpoints=1, loc='upper right', ncol=3, fontsize=10)
plt.gca().update(dict(xlabel='model year', ylabel='weight (lbs)', title='heavy and light from model years 1970-1982'))
plt.show()
```



As the years progress from 1970 to 1982, the amount of light cars (below the mean of ~2977.58 pounds) is very consistent; however, the amount of heavy cars drops significantly.
The peak weight per model year shows a consistent downtrend as well.