lab1

September 15, 2023

0.1 Lab 1: Introduction to Python

Jack Krebsbach Math 313

Sep 15 2023

Basic Commands Any function we can get more information on by placing a `?` after the function without invoking it.

To log objects from our explorations we can use `print` to create output from the cell.

[ ]:
```
# Example print
print?
```

[ ]:
```
# Print function outputs any input
x = 3
print(x+3)

print('hello word')
#If we wish to create a new line we use the \n character
print('hello\nworld')
print?
```

```
6
hello word
hello
world
```

[ ]:
```
# Strings are textual data. We surround text with double or single quotes, but␣
↪they must be consistent for each string 'hello" is invalid.
# We can concatenate them like so.
x= 'Statistical'
y= 'Learning'
z = x+' '+y
print(z)

# Notice that we can assign strings to variables and concatenate them like so.␣
↪We can also concatenate them correctly.
print('Hello' + ' ' + 'World')
```

Statistical Learning
Hello World

[ ]:
```
# A string is type of sequence (generic term for ordered list). We can thus␣
↪grab indices of both strings and other lists. Note that lists can store␣
↪arbitrary objects.
# Indices start at 0 in python, Ex
x = 'Math'
print(x[0])
y = [1,2,3,4]
print(y[3])
# If we try and grab y[4] it will error because it is out of founds.
```

```
M
4
```

0.1.1 Numerical Python

To work with data we usually use packages `numpy` or `scipy` We can access these packages by importing them at the top of our script or jupyter notebook.

Numpy When working with numpy an *array* is a generic term for a multidimensional set of numbers. We can create one-dimensional arrays or multidimensional arrays.

If two *arrays* are of the same length we can do things like add them together! If they are not the same shape we will get an error.

[ ]:
```
# Ex renaming the export
import numpy as np
# We can fnd out more information by accessing the DocString We would run -> np.
↪array?
# Ex create one dimensional arrays (vectors)
x = np.array([1,2,3,4])
y = np.array([2,3,4,9])
# Note for this two work x and y have to be the same shape
z = x + y
print(z)
```

```
[ 3  5  7 13]
```

[ ]:
```
# Notice we can import named exports the following and even rename them!
from scipy import pi as my_named_pi
# Create a multidimensional array
t = np.array([my_named_pi,1])
# We can grab the dimension of this array
print(t.ndim)
```

```
1
```

Data Types It is important to understand what the data type of these objects we are working with. When working with numpy we can also access the datatype using builtin functions, but we can also ask the datatype using the base Python functionality.

[ ]:
```
#Consider the following data types in python
x = 5
y = 5.5
z = my_named_pi
k = 'Statistics'
array= [1,2,3,4,5]
array3= [1,3,'hello','world']
print(type(x), type(y), type(z), type(k), type(array), type(array3))

#Note that python does not distinguish the type between a list of integers and␣
 ↪a mixed list with integers and strings
# Now look at the data type using numpy
np_array = np.array([1,3,4,5])
np_float_array = np.array([my_named_pi, 2,3])
print(np_array.dtype)
print(np_float_array.dtype)
# Notice that when my numpy array contains floats and integers we coerce the␣
 ↪integers into floats (floats are more precise than integers).
```

```
<class 'int'> <class 'float'> <class 'float'> <class 'str'> <class 'list'>
<class 'list'>
int64
float64
```

[ ]:
```
# We can also reshape our data if it makes sense
x = np.array(np.arange(100))
print(x)
y = x.reshape(5,20)
print(y)
# Now we need a new index to grab this new dimension
# To grab the first element of this reshaped array
print(y[0,0])
# To grab the last element of this reshpaed array
print(y[4,18])
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99]
[[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59]
 [60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79]
```

```
 [80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99]]
0
98
```

[ ]:

```
# Interestingly enough if we mutate our reshaped array it will also mutate the␣
 ↪original array
from scipy import pi
x = np.array(np.arange(100))
y = x.reshape(5,20)
print(x)
y[0,0] = pi
print(x)
# Note that even if we mutated our data with a float it gets cast as an integer!
 ↪!
print(y.dtype)
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99]
[ 3  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95
 96 97 98 99]
int64
```

0.1.2 Lists and Tuples

Lists are like tuples, they both represent sequences of objects. Note that there are some differences, mainly that we cannot mutate a tuple, ie we can not modify any of is elements.

[ ]:

```
# If we try and mutate the following tuple it will error
tuple = (1,2,3,4)
array  = [1,2,3,4]
array[0] = 3;
try:
    tuple[0] = 3;
except Exception as e:
    print(e)

print(tuple)
print(array)
```

```
'tuple' object does not support item assignment
(1, 2, 3, 4)
[3, 2, 3, 4]
```

[ ]:
```
# There are built in functions that can be invoked on arrays
x = np.array([pi, 2*pi, 3*pi])
# Shape
print(x.shape)
# ndim
print(x.ndim)
# Transpose
print(x.T)
print(x.T.shape)
```

```
(3,)
1
[3.14159265 6.28318531 9.42477796]
(3,)
```

[ ]:
```
# We also can apply functions to arrays ex square root or power
y = np.sqrt(x)
z = x**3
print(y)
print(z)
```

```
[1.77245385 2.50662827 3.06998012]
[ 31.00627668 248.05021344 837.16947037]
```

Random Data It is useful to be able to generate random data. We can use *keyword* arguments to be passed to the function,ie parameters. We can use positional arguments and named arguments. The order matters for positional arguments, but we can use named arguments in any order.

An important element of generating random data is setting the seed.

[ ]:
```
# Consider generating 50 elements from a normal distribution with the parameter␣
↪50
z = np.random.normal(size = 5)
print(z, z.dtype)
# We can add arrays together and generate different data
y = np.random.normal(loc = -1, size =5)
print(y)
print(z+y)
# We can find the correlation matrix by
np.corrcoef(z,y)
```

```
[ 0.19877954  0.59797598 -1.61923133  0.71815191  0.67650412] float64
[-2.16316721 -2.31174807  0.12891709 -2.07006803 -0.60068586]
[-1.96438767 -1.71377209 -1.49031423 -1.35191611  0.07581826]
```

[ ]:
```
array([[ 1.        , -0.71368782],
       [-0.71368782,  1.        ]])
```

```
[ ]: import numpy as np
     # Random seeds, this will ensure that the data is consistent by generating␣
      ↪random data over and over again
     # The default seed is:
     np.random.default_rng()
```

```
[ ]: Generator(PCG64) at 0x104EFB060
```

Mean, Variance, Standard Deviation The package numpy provides these functions to calculate statistics on data.

```
[ ]: x = np.array([1,2,3,4])
     # Mean
     print(np.mean(x))
     # Variance
     print(np.var(x))
     # Standard Deviation Note we divide by n not n-1, which means this is a biased␣
      ↪estimator
     print(np.std(x))

     # We can also apply these to matrices
     rng = np.random.default_rng()
     y = rng.standard_normal((10,3))
     # The first axis is referenced by 0 and is the rows, and the second is the␣
      ↪columns which is 1
     print(y)
     print('\n\nMeans\n')
     # Rows
     print(y.mean(axis=0))
     # Columns
     print(y.mean(axis=1))
```

```
2.5
1.25
1.118033988749895
[[-0.80956883  0.50634283  1.02460713]
 [ 0.70021202  0.26572454 -1.15872546]
 [ 1.03523017  1.51296515 -0.59310847]
 [ 1.04532458  1.64303525  0.03509235]
 [-0.29235404 -0.71556559  0.78805657]
 [-1.54693877 -1.54423376 -0.30895584]
 [-1.17251022 -1.19139436 -1.5931905 ]
 [-0.12266961  0.35570405  1.0243101 ]
 [ 1.06520762  0.22053755 -0.93313849]
 [-0.94543101 -0.74741569  3.17528774]]
```

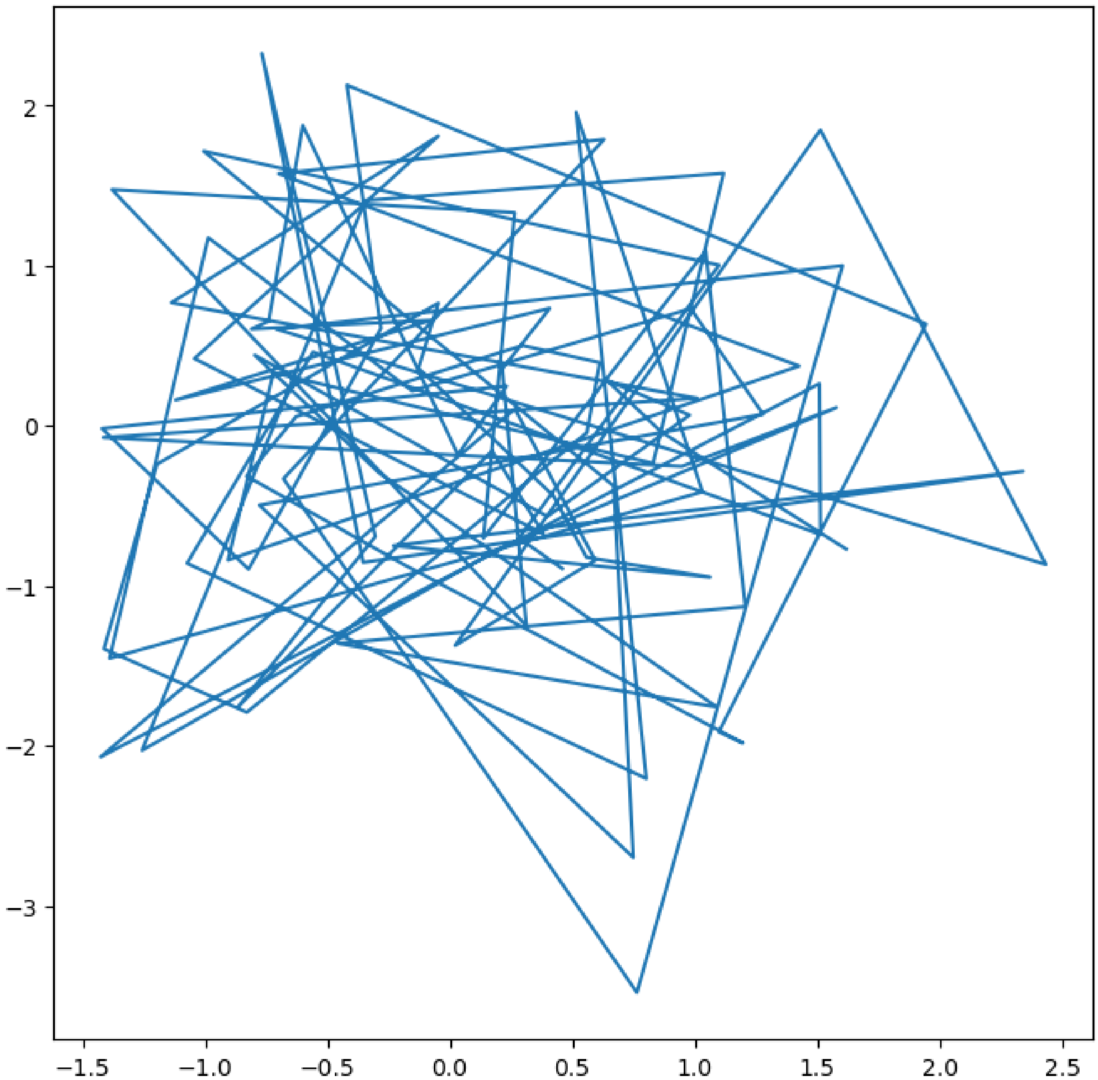
```
Means

[-0.10434981  0.03057     0.14602351]
[ 0.24046038 -0.06426297  0.65169562  0.90781739 -0.07328768 -1.13337613
 -1.3190317   0.41911485  0.11753556  0.49414701]
```

[ ]:
```
y.mean?
```

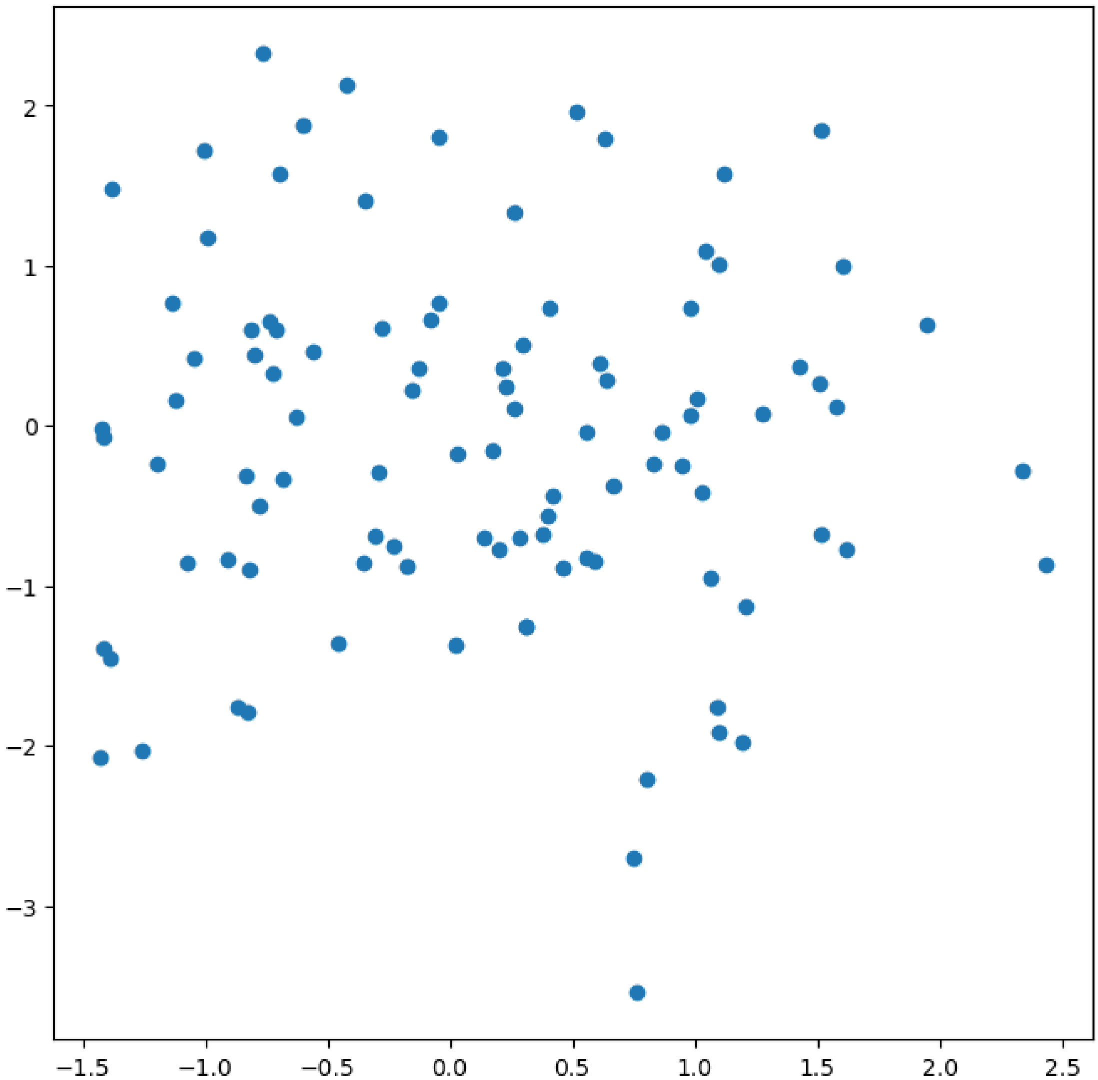
[ ]:
```
import numpy as np
rng = np.random.default_rng()
y = rng.standard_normal(10)
```

Graphics

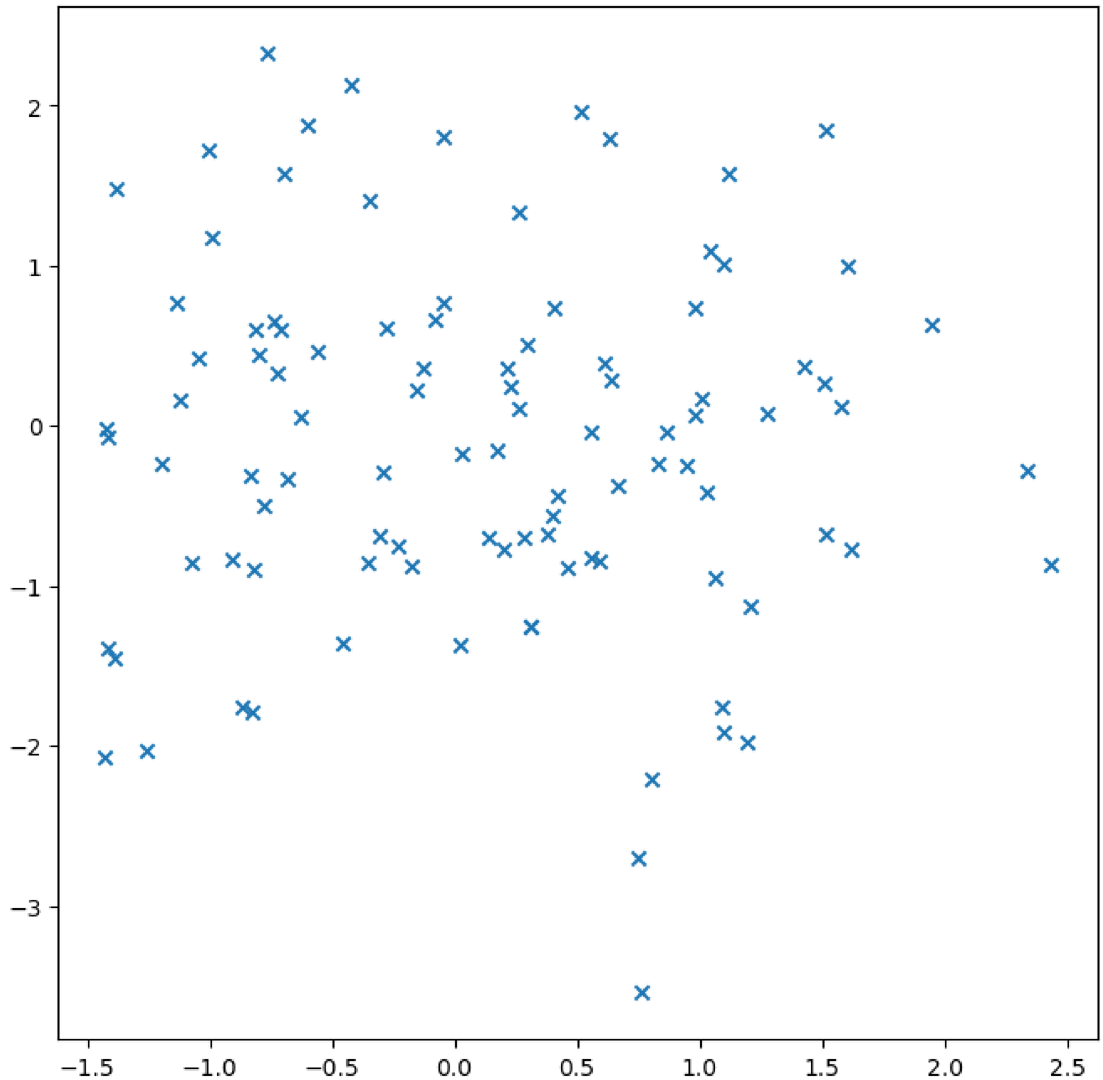
[ ]:
```
from matplotlib.pyplot import subplots
fig, ax = subplots(figsize=(8, 8))
x = rng.standard_normal(100)
y = rng.standard_normal(100)
ax.plot(x, y);
```



[ ]:
```
# To create a scatter plot we can provide an additional argument to ax.plot
# Note that we could also do this by using ax.scatter()
fig, ax = subplots(figsize=(8, 8))
ax.plot(x, y, 'o');
```

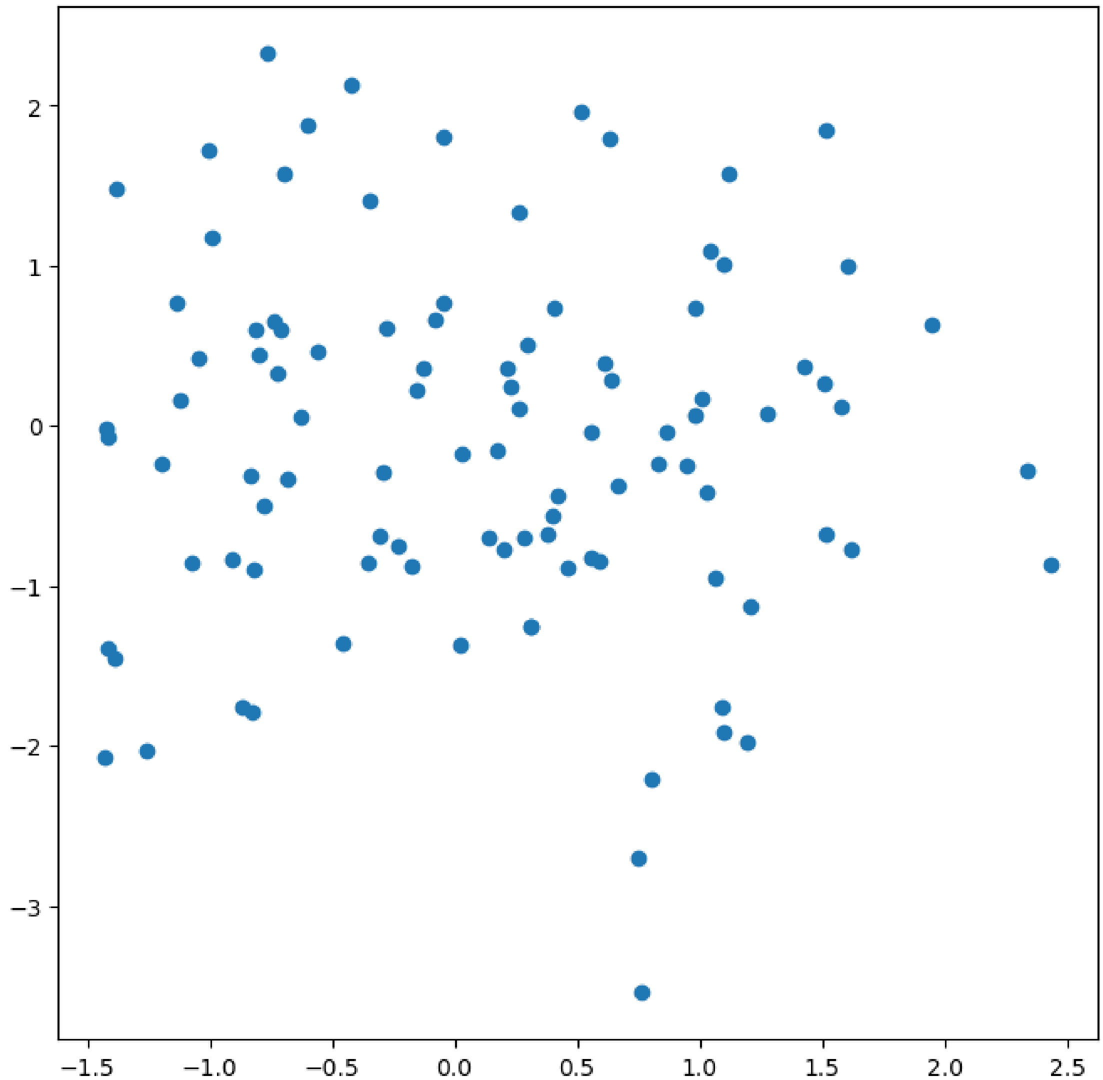


[ ]:
```
# The values are plotted as xs instead of circles.
fig, ax = subplots(figsize=(8, 8))
ax.scatter(x, y, marker='x');
```

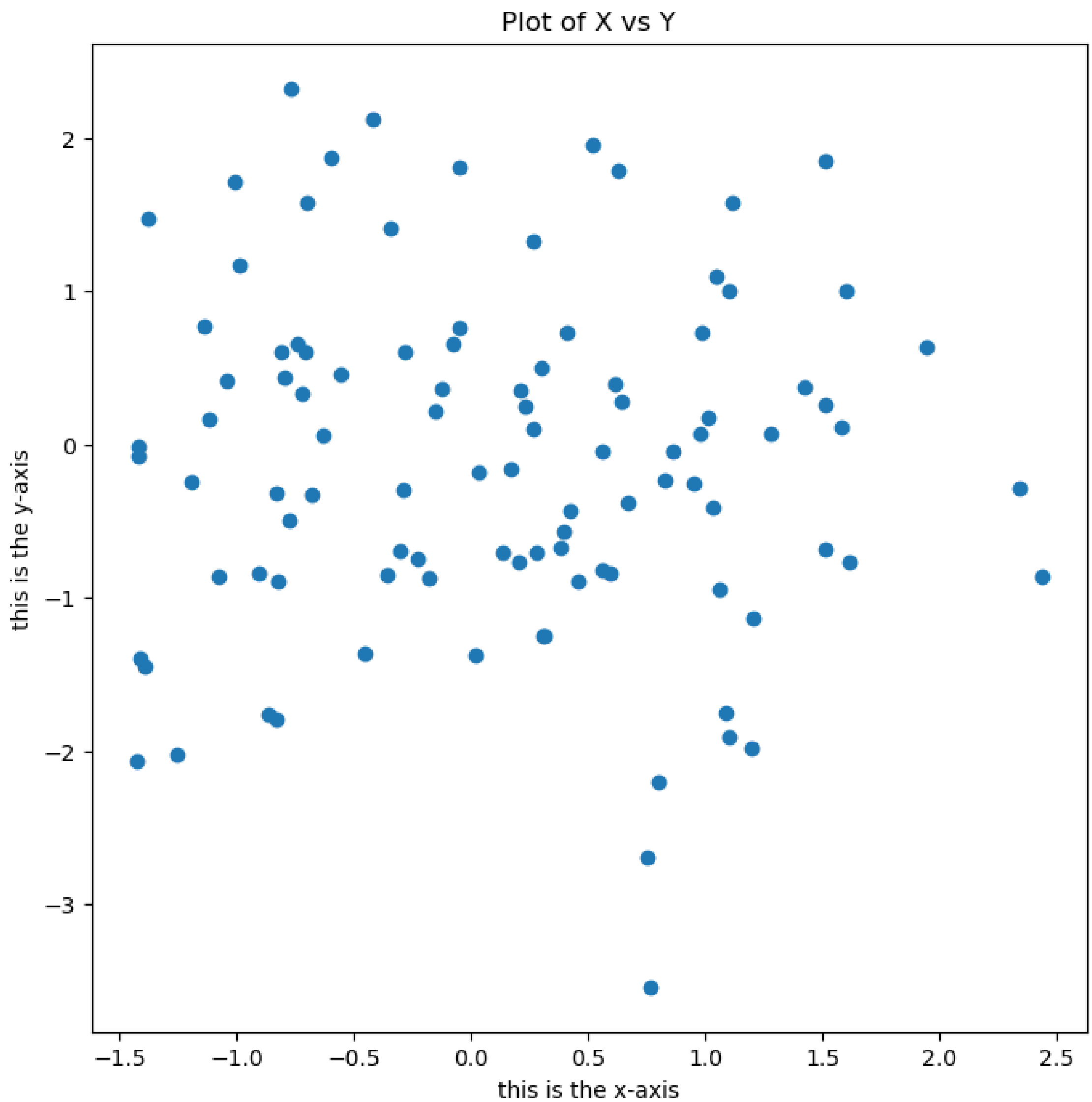


Note If we end a code block with a semicolon. This will prevent ax.plot() from printing the text to the notebook. Right now this doesn't seem to be working in jupyter lab for me or in pycharm.

[ ]:
```
fig, ax = subplots(figsize=(8, 8))
ax.scatter(x, y, marker='o');
#The semicolon doesn't seem to stop the figure from being rendered.
```

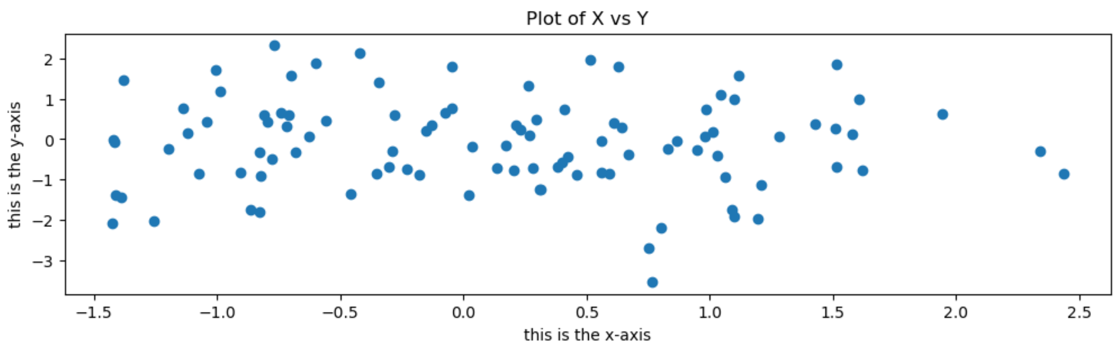


[ ]:
```
fig, ax = subplots(figsize=(8, 8))
ax.scatter(x, y, marker='o')
ax.set_xlabel("this is the x-axis")
ax.set_ylabel("this is the y-axis")
ax.set_title("Plot of X vs Y");
```

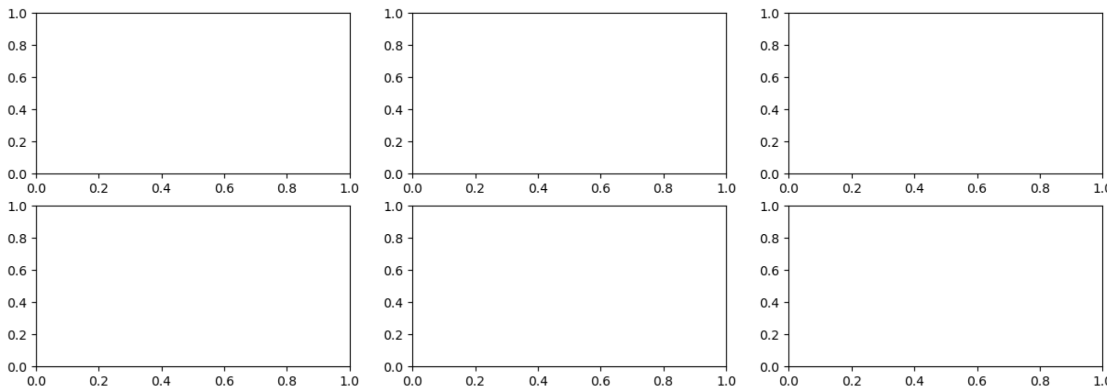


[ ]:
```
#With the fig object we can set the attributes of the display.
fig.set_size_inches(12,3)
fig
```

[ ]:

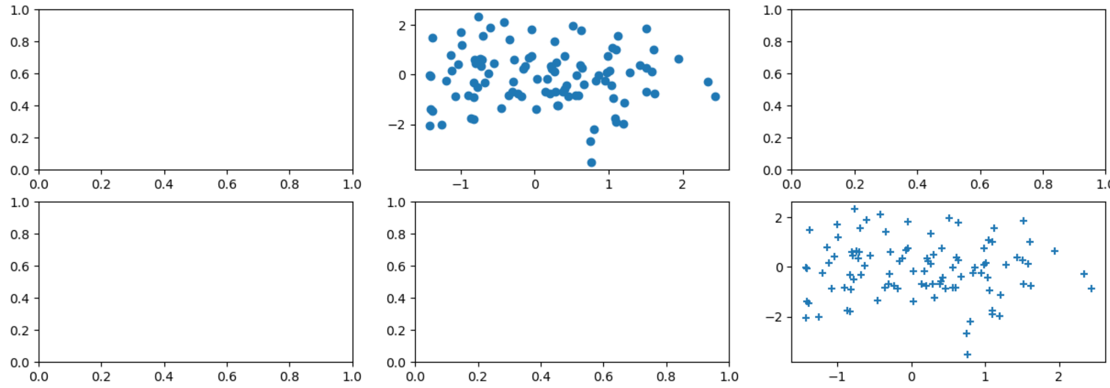



```
[ ]: # To create several objects within the same display we
     # can use sharex=True
     fig, axes = subplots(nrows=2, ncols=3,
     figsize=(15, 5))
```



```
[ ]: # We can also set some of the plots to be scatter plots for example making the␣
      ↪second column of the first row a scatter plot and the third column of the␣
      ↪second row with different markers
     axes[0,1].plot(x, y, 'o')
     axes[1,2].scatter(x, y, marker='+')
     fig
```

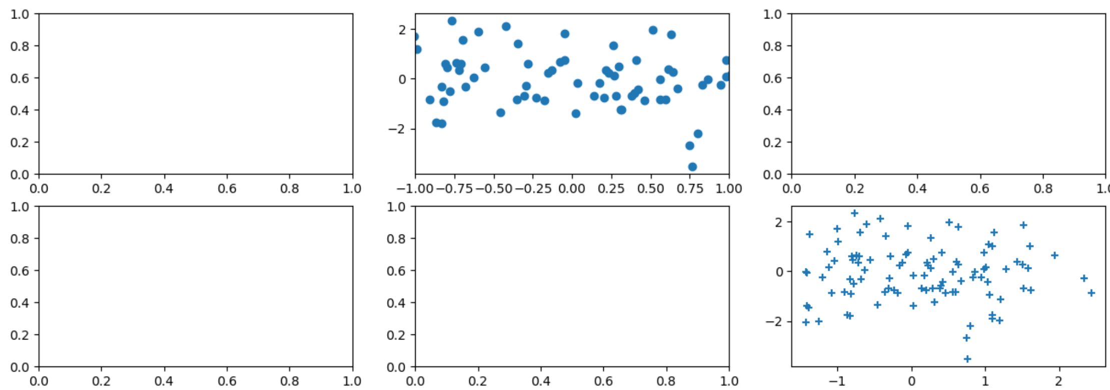
```
[ ]:
```



```
[ ]: # To save these plots we can use fig.savfig with the desired resolution.
     fig.savefig("Figure.png", dpi=400)
     fig.savefig("Figure.pdf", dpi=200);
```

```
[ ]: # We can still update these
     axes[0,1].set_xlim([-1,1])
     fig.savefig("Figure_updated.jpg")
     fig
```

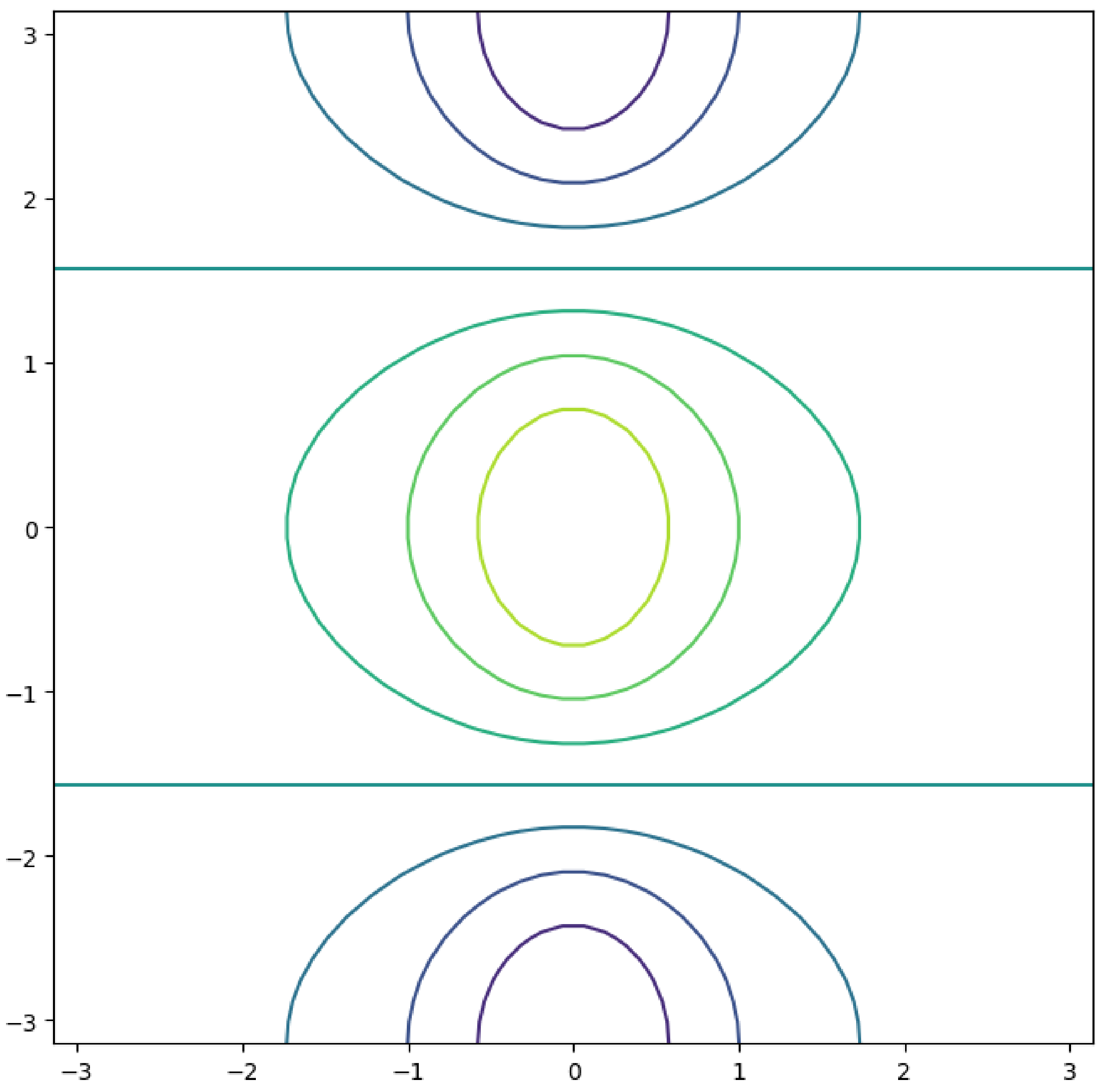
[ ]:



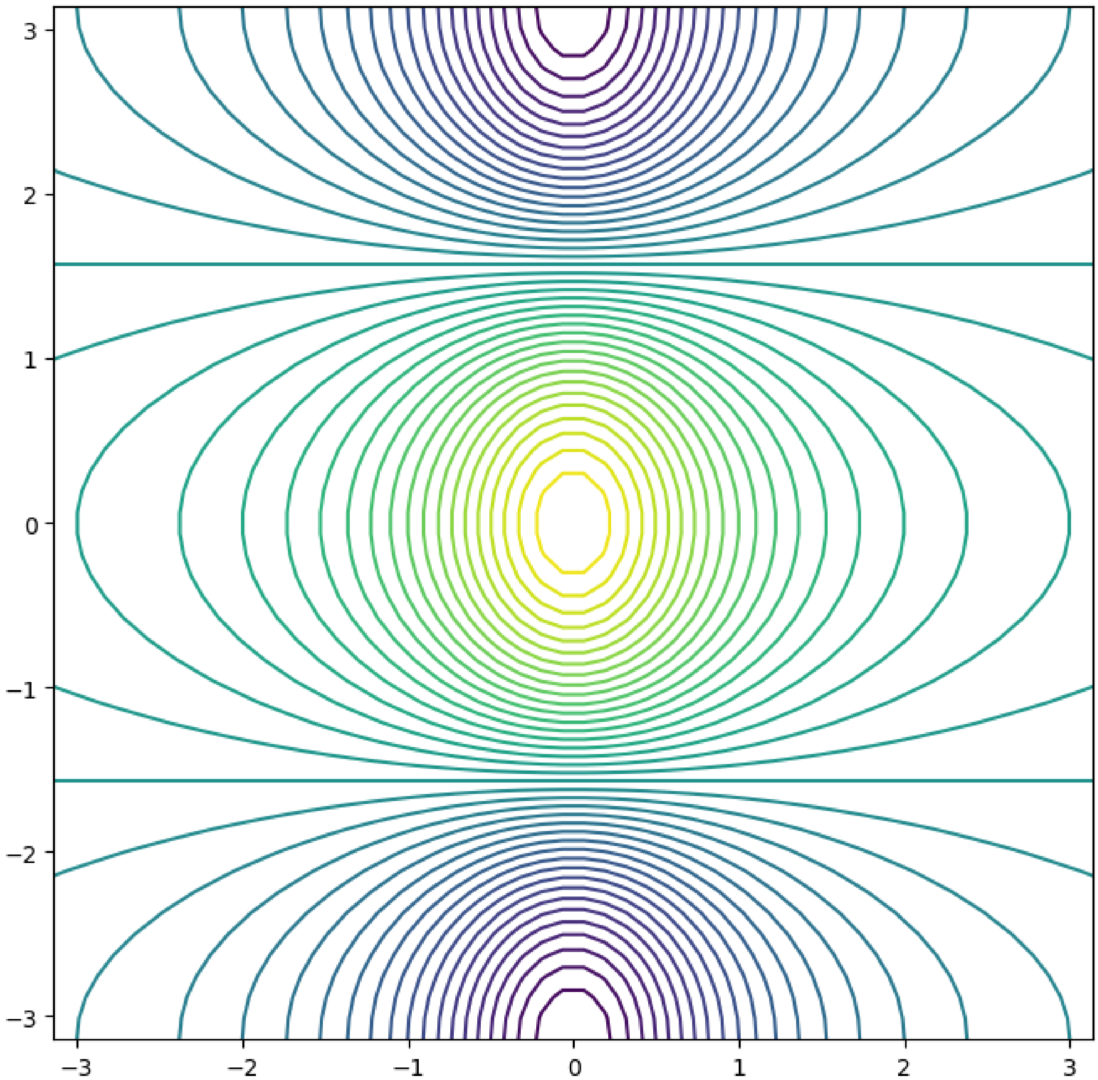
0.1.3 Contour Plots

We can create more complex plots using contour plots. We can take in `x` values to denote the first dimension and `y` values to denote the second dimension. The third matrix is the `z` value.

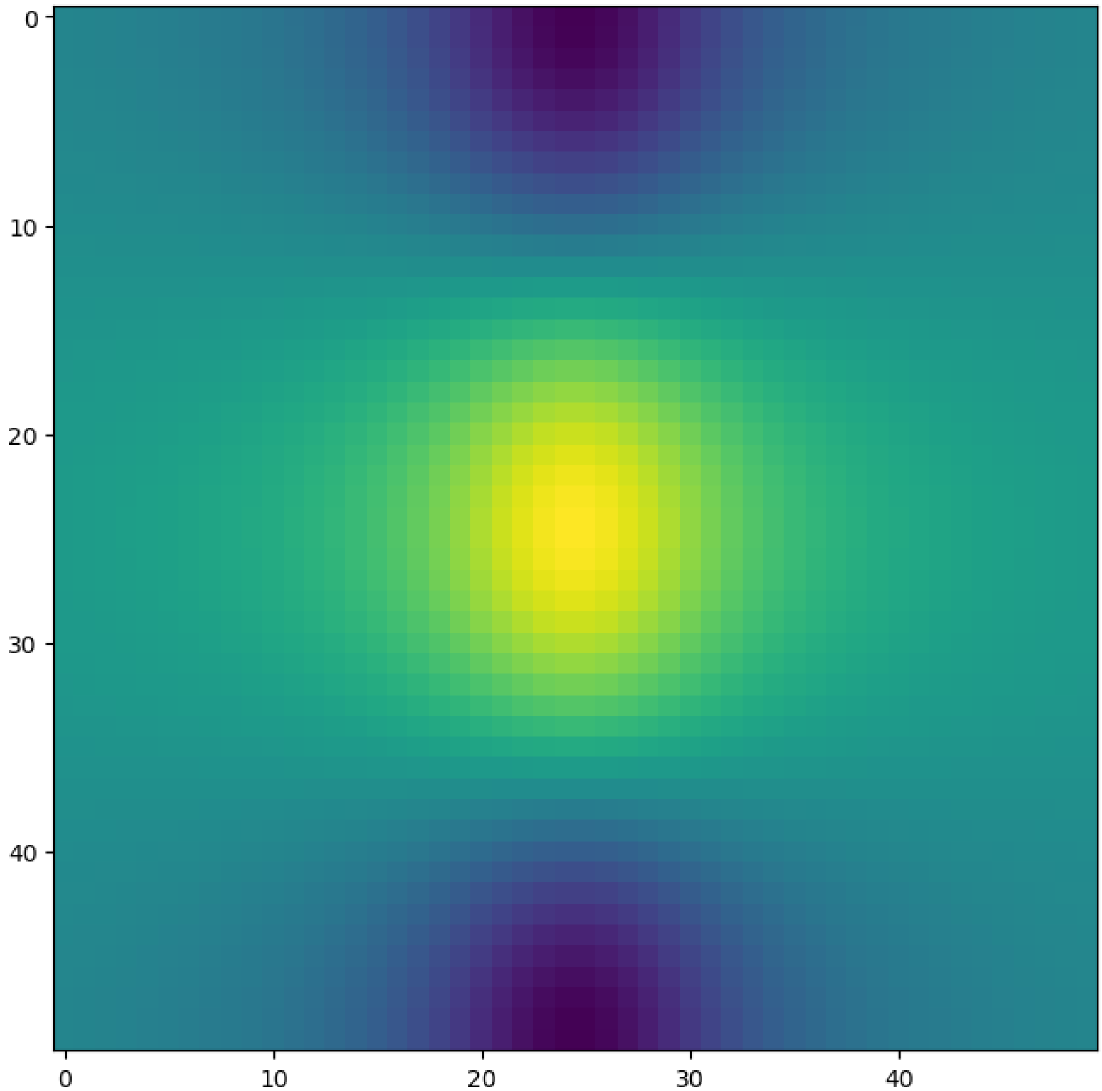
```
[ ]: fig, ax = subplots(figsize=(8, 8))
     x = np.linspace(-np.pi, np.pi, 50)
     y=x
     f = np.multiply.outer(np.cos(y), 1 / (1 + x**2))
     ax.contour(x, y, f);
```



[ ]:
```
# To increase the resolution
fig, ax = subplots(figsize=(8, 8))
ax.contour(x, y, f, levels=45);
```




```
[ ]: #Instead of a contour plots we can use heatmaps. This is false color image␣
     ↪generated by the z value.
     fig, ax = subplots(figsize=(8, 8))
     ax.imshow(f);
```



0.2 Sequences and Slice Notation

We can access values in our arrays using slice notation.

[ ]:
```
# Create a sequence of numbers from 0 to 10 containing 11 values
seq = np.linspace(0,10,11)

seq2 = np.linspace(0,10,20)

print('Sequence 1:\n', seq)

print('Sequence 2:\n', seq2)
```

```
Sequence 1:
 [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
Sequence 2:
 [ 0.          0.52631579  1.05263158  1.57894737  2.10526316  2.63157895
  3.15789474  3.68421053  4.21052632  4.73684211  5.26315789  5.78947368
  6.31578947  6.84210526  7.36842105  7.89473684  8.42105263  8.94736842
  9.47368421 10.        ]
```

[ ]:
```
# If step is not specified then the default value of 1 is used.
default  = np.arange(0,10)
# 10 is not in the output due to how Python slices arrays, lists, and tuples.␣
↪We go up to but not include the upper range.
default
```

[ ]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[ ]:
```
# Consider slicing the following string
"hello world"[3:6]
# This is shorthand for
"hello world"[slice(3,6)]
```

[ ]:
```
'lo '
```

[ ]:
```
# Slice is of class slice.
slice?
print(slice(1,10))
print(type(slice(1,10)))
```

```
slice(1, 10, None)
<class 'slice'>
```

0.2.1 Indexing Data

We need a way to access our data in single or multidimensional arrays

[ ]:
```
A = np.array(np.arange(16)).reshape((4,4))
A
```

[ ]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

[ ]:
```
# Grab the 2nd row and 3rd column.
A[1,2]

# To select multiple rows at the same time
# Retrieve the second and fourth rows
```

```
A[[1,3]]
# Retrieve the first and third columns with all the rows
A[:,[0,2]]
```

```
[ ]: array([[ 0,  2],
            [ 4,  6],
            [ 8, 10],
            [12, 14]])
```

```
[ ]: print(A)
# Note this grabs two elements from the matrix - not rows and columns
print(A[[1,3],[0,2]])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[ 4 14]
```

```
[ ]: # Consequently, naturally this will fail
try:
    A[[1,3],[0,2,3]]
except Exception as e:
    print('Error: \n{}'.format(e))
```

```
Error:
shape mismatch: indexing arrays could not be broadcast together with shapes (2,)
(3,)
```

```
[ ]: # To do this we need to build our sub matrix sequentially, fist we grab the␣
 ↪rows then the columns.
A[[1,3]][:,[0,2]]

# We can also use the ix_() method which creates a mesh object to efficiently␣
 ↪slice sub-matrices.
#This grabs the 2nd and 3rd rows and 1st, 3rd, and 4th columns
idx = np.ix_([1,3],[0,2,3])
print(A)
print(A[idx])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[[ 4  6  7]
 [12 14 15]]
```

[ ]:
```
# We can also use slice objects to access our data
print(A)
print(A[slice(1,4,3),slice(0,3,2)])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[[4 6]]
```

0.2.2 Boolean Indexing

Another useful way to access data is using boolean indexing. Boolean is a type that is either True or False, which is equivalent 0 or 1, but will not return the same subset! Using integers will actually grab the subset by index (1st and 2nd index).

[ ]:
```
keep_rows = np.zeros(A.shape[0], bool)
keep_rows
# Set two rows to true
keep_rows[[1,3]] = True
keep_rows
# This is also equivalent of using 0s and 1s. But be careful they will not␣
↪return the same data!
np.all(keep_rows == np.array([0,1,0,1]))
```

[ ]:
```
True
```

[ ]:
```
# Original Matrix
print(A)
# This will grab the second and fourth columns
print(A[keep_rows])
# This will actually grab the rows by index.
print(A[np.array([0,1,0,1])])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[[ 4  5  6  7]
 [12 13 14 15]]
[[0 1 2 3]
 [4 5 6 7]
 [0 1 2 3]
 [4 5 6 7]]
```

[ ]:
```
# Like before we can use a mesh object. We grab the second and fourth rows and␣
↪the first, third, and fourth columns.
keep_cols = np.zeros(A.shape[1], bool)
```

```
keep_cols[[0, 2, 3]] = True
idx_bool = np.ix_(keep_rows, keep_cols)
print(A)
print(np.zeros(A.shape[1], bool))
print(keep_cols)
print(A[idx_bool])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[False False False False]
[ True False  True  True]
[[ 4  6  7]
 [12 14 15]]
```

[ ]:
```
# We can mix and match booleans and indices when sub-setting data
idx_mixed = np.ix_([1,3], keep_cols)
A[idx_mixed]
```

[ ]:
```
array([[ 4,  6,  7],
       [12, 14, 15]])
```

0.2.3 Loading Data

We can use csv files or .data files to read in data. In `pandas` we can read in whitespace-delimited versions with the parameter `delim_whitespace=True`.

[ ]:
```
import pandas as pd
Auto = pd.read_csv("../data/Auto.csv")
Auto_data = pd.read_csv("../data/Auto.data", delim_whitespace=True)
```

[ ]:
```
# We can access columns like so
Auto['horsepower']
# Note that the data type of the column is actually object. Every value was␣
 ↪interpreted as a string.

# To understand why we can look at the unique values:
np.unique(Auto['horsepower'])
#Every instance of ? is not a number
```

[ ]:
```
array(['100', '102', '103', '105', '107', '108', '110', '112', '113',
       '115', '116', '120', '122', '125', '129', '130', '132', '133',
       '135', '137', '138', '139', '140', '142', '145', '148', '149',
       '150', '152', '153', '155', '158', '160', '165', '167', '170',
       '175', '180', '190', '193', '198', '200', '208', '210', '215',
       '220', '225', '230', '46', '48', '49', '52', '53', '54', '58',
```

```
       '60', '61', '62', '63', '64', '65', '66', '67', '68', '69', '70',
       '71', '72', '74', '75', '76', '77', '78', '79', '80', '81', '82',
       '83', '84', '85', '86', '87', '88', '89', '90', '91', '92', '93',
       '94', '95', '96', '97', '98', '?'], dtype=object)
```

[ ]:
```python
#To coerce the values ? into NANs we can read in the data like so

Auto = pd.read_csv('../data/Auto.data', na_values=['?'],
                   delim_whitespace=True)
#We can ask questions like the shape of the object:
Auto.shape

#Instead of changing the NA values we can drop them
Auto_sanitized = Auto.dropna()
print("Auto Shape:\n", Auto.shape)

print("Auto Sanitized Shape:\n",Auto_sanitized.shape)
```

```
Auto Shape:
 (397, 9)
Auto Sanitized Shape:
 (392, 9)
```

Selecting Rows and Columns We can use dot notation to also grab the columns and rows of a data set.

[ ]:
```python
Auto = Auto_sanitized
Auto.columns

#Simlar to accessing values in nested arrays we can slice data
#Subset Columns
Auto[:3]
#Subset Rows
index_of_75 = Auto.year > 75
Auto[index_of_75]
#If we pass in an array of strings we can get multiple columns
Auto[['mpg','horsepower']]
```

[ ]:

	mpg	horsepower
0	18.0	130.0
1	15.0	165.0
2	18.0	150.0
3	16.0	150.0
4	17.0	140.0
..	…	…
392	27.0	86.0
393	44.0	52.0

```
394  32.0         84.0
395  28.0         79.0
396  31.0         82.0

[392 rows x 2 columns]
```

Indexes By default, the indices of the rows in our data frame are ordered integers, but we can set the index using `set_index()`

This specifies how the rows are named. You can grab rows by the indices.

[ ]:
```
print(Auto.index)
Auto_re_indexed = Auto.set_index('name')
print(Auto_re_indexed.index)
```

```
Index([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,
       …
       387, 388, 389, 390, 391, 392, 393, 394, 395, 396],
      dtype='int64', length=392)
Index(['chevrolet chevelle malibu', 'buick skylark 320', 'plymouth satellite',
       'amc rebel sst', 'ford torino', 'ford galaxie 500', 'chevrolet impala',
       'plymouth fury iii', 'pontiac catalina', 'amc ambassador dpl',
       …
       'chrysler lebaron medallion', 'ford granada l', 'toyota celica gt',
       'dodge charger 2.2', 'chevrolet camaro', 'ford mustang gl', 'vw pickup',
       'dodge rampage', 'ford ranger', 'chevy s-10'],
      dtype='object', name='name', length=392)
```

[ ]:
```
# Now we can grab rows by .loc[]
rows = ['amc rebel sst', 'ford torino']
Auto_re_indexed.loc[rows]
```

[ ]:
```
                mpg  cylinders  displacement  horsepower  weight  \
name
amc rebel sst  16.0          8         304.0       150.0  3433.0
ford torino    17.0          8         302.0       140.0  3449.0

               acceleration  year  origin
name
amc rebel sst          12.0    70       1
ford torino            10.5    70       1
```

[ ]:
```
# We can still use the .iloc method to find values using row and column
↪indices.

#Grabs the 2nd and 3rd rows.
Auto_re_indexed.iloc[[1,2]]
```

```
#Grabs the 2nd and 3rd columns
Auto_re_indexed.iloc[:,[1,2]]

#Grab the 2nd and 3rd column of the 2nd and 3rd row
Auto_re_indexed.iloc[[1,2],[1,2]]
```

[ ]:

```
                    cylinders  displacement
name
buick skylark 320           8         350.0
plymouth satellite          8         318.0
```

[ ]:

```
#Index entries do not have to be unique
Auto_re_indexed.loc['ford galaxie 500', ['mpg', 'origin']]
```

[ ]:

```
                   mpg  origin
name
ford galaxie 500  15.0       1
ford galaxie 500  14.0       1
ford galaxie 500  14.0       1
```

Conditional Subset We can select rows and columns based off of boolean conditions

& - the and operator | - or operator "<" - less than operator ">" - greater than operator == equal to operator

lambdas are small useful inline anonymous functions we can call when we subset a dataframe.

[ ]:

```
year_80 = Auto_re_indexed['year'] > 80
print('Boolean on rows greater than year 80\n',year_80)
Auto_re_indexed.loc[year_80, ['weight', 'origin']]
```

```
Boolean on rows greater than year 80
 name
chevrolet chevelle malibu    False
buick skylark 320            False
plymouth satellite           False
amc rebel sst                False
ford torino                  False
                             …
ford mustang gl               True
vw pickup                     True
dodge rampage                 True
ford ranger                   True
chevy s-10                    True
Name: year, Length: 392, dtype: bool
```

```
[ ]:                              weight  origin
     name
     plymouth reliant             2490.0       1
     buick skylark                2635.0       1
     dodge aries wagon (sw)       2620.0       1
     chevrolet citation           2725.0       1
     plymouth reliant             2385.0       1
     toyota starlet               1755.0       3
     plymouth champ               1875.0       1
     honda civic 1300             1760.0       3
     subaru                       2065.0       3
     datsun 210 mpg               1975.0       3
     toyota tercel                2050.0       3
     mazda glc 4                  1985.0       3
     plymouth horizon 4           2215.0       1
     ford escort 4w               2045.0       1
     ford escort 2h               2380.0       1
     volkswagen jetta             2190.0       2
     honda prelude                2210.0       3
     toyota corolla               2350.0       3
     datsun 200sx                 2615.0       3
     mazda 626                    2635.0       3
     peugeot 505s turbo diesel    3230.0       2
     volvo diesel                 3160.0       2
     toyota cressida              2900.0       3
     datsun 810 maxima            2930.0       3
     buick century                3415.0       1
     oldsmobile cutlass ls        3725.0       1
     ford granada gl              3060.0       1
     chrysler lebaron salon       3465.0       1
     chevrolet cavalier           2605.0       1
     chevrolet cavalier wagon     2640.0       1
     chevrolet cavalier 2-door    2395.0       1
     pontiac j2000 se hatchback   2575.0       1
     dodge aries se               2525.0       1
     pontiac phoenix              2735.0       1
     ford fairmont futura         2865.0       1
     volkswagen rabbit l          1980.0       2
     mazda glc custom l           2025.0       3
     mazda glc custom             1970.0       3
     plymouth horizon miser       2125.0       1
     mercury lynx l               2125.0       1
     nissan stanza xe             2160.0       3
     honda accord                 2205.0       3
     toyota corolla               2245.0       3
     honda civic                  1965.0       3
     honda civic (auto)           1965.0       3
```

```
datsun 310 gx                        1995.0       3
buick century limited                2945.0       1
oldsmobile cutlass ciera (diesel)    3015.0       1
chrysler lebaron medallion           2585.0       1
ford granada l                       2835.0       1
toyota celica gt                     2665.0       3
dodge charger 2.2                    2370.0       1
chevrolet camaro                     2950.0       1
ford mustang gl                      2790.0       1
vw pickup                            2130.0       2
dodge rampage                        2295.0       1
ford ranger                          2625.0       1
chevy s-10                           2720.0       1
```

[ ]:
```
# We can also do this using lambdas
Auto_re_indexed.loc[lambda df: df['year'] > 80, ['weight', 'origin']]

# This checks if the car has a displacement or if the car brand is either ford␣
 ↪or datsun. Then we get those cars weight and origin.
Auto_re_indexed.loc[lambda df: (df['displacement'] < 300)
& (df.index.str.contains('ford')
| df.index.str.contains('datsun')), ['weight', 'origin']
]
```

[ ]:
```
                       weight  origin
name
ford maverick          2587.0       1
datsun pl510           2130.0       3
datsun pl510           2130.0       3
ford torino 500        3302.0       1
ford mustang           3139.0       1
datsun 1200            1613.0       3
ford pinto runabout    2226.0       1
ford pinto (sw)        2395.0       1
datsun 510 (sw)        2288.0       3
ford maverick          3021.0       1
datsun 610             2379.0       3
ford pinto             2310.0       1
datsun b210            1950.0       3
ford pinto             2451.0       1
datsun 710             2003.0       3
ford maverick          3158.0       1
ford pinto             2639.0       1
datsun 710             2545.0       3
ford pinto             2984.0       1
ford maverick          3012.0       1
ford granada ghia      3574.0       1
```

```
datsun b-210            1990.0       3
ford pinto              2565.0       1
datsun f-10 hatchback   1945.0       3
ford granada            3525.0       1
ford mustang ii 2+2     2755.0       1
datsun 810              2815.0       3
ford fiesta             1800.0       1
datsun b210 gx          2070.0       3
ford fairmont (auto)    2965.0       1
ford fairmont (man)     2720.0       1
datsun 510              2300.0       3
datsun 200-sx           2405.0       3
ford fairmont 4         2890.0       1
datsun 210              2020.0       3
datsun 310              2019.0       3
ford fairmont           2870.0       1
datsun 510 hatchback    2434.0       3
datsun 210              2110.0       3
datsun 280-zx           2910.0       3
datsun 210 mpg          1975.0       3
ford escort 4w          2045.0       1
ford escort 2h          2380.0       1
datsun 200sx            2615.0       3
datsun 810 maxima       2930.0       3
ford granada gl         3060.0       1
ford fairmont futura    2865.0       1
datsun 310 gx           1995.0       3
ford granada l          2835.0       1
ford mustang gl         2790.0       1
ford ranger             2625.0       1
```

For Loops Sometimes we want to repeatedly evaluate a chunk of code. We can do this using for loops.

[ ]:
```
# Add up all the numbers in an an array
total = 0
for value in [3,2,19]:
    total += value
print('Total is: {0}'.format(total))

#To sum over multiple arrays we can create a tuple of objects and access them␣
↪in our for loop
weighted_total = 0
for value, weight in zip([2,3,19], [0.2,0.3,0.5]):
    weighted_total += weight * value
print('Weighted average is: {0}'.format(weighted_total))
```

```
Total is: 24
Weighted average is: 10.8
```

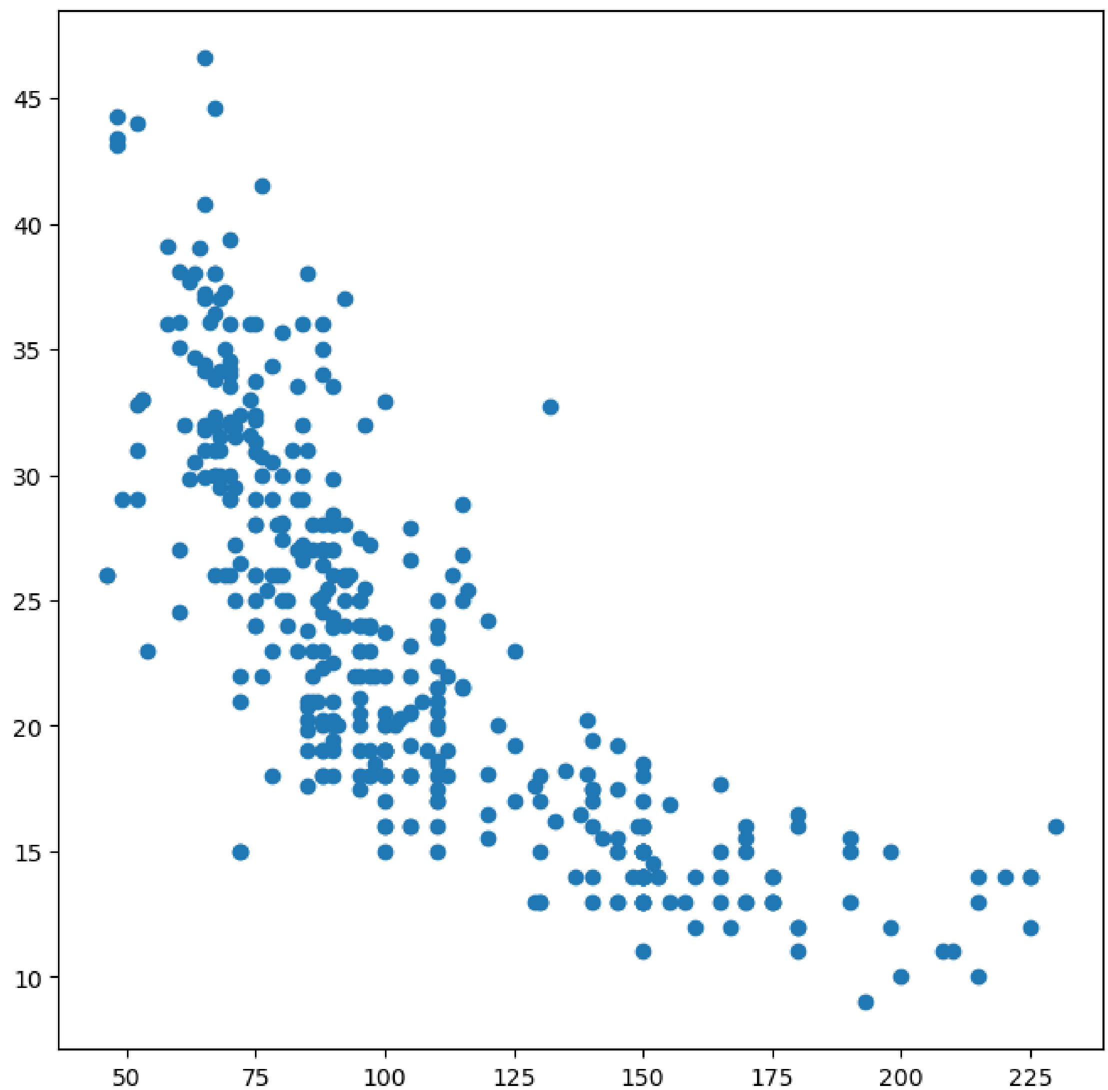
0.2.4 String Formatting

A powerful tool to display our data usefully is to use string formatting.

[ ]:
```
# Generate some random data and use for loops and string formatting to display␣
↪revalent summary statistics.

rng = np.random.default_rng(1)
A = rng.standard_normal((127, 5))
M = rng.choice([0, np.nan], p=[0.8,0.2], size=A.shape)
A += M

D = pd.DataFrame(A, columns=['food',
'bar', 'pickle', 'snack', 'popcorn'])

# We use the template formatting language to specify where our values should␣
↪print and how to format them. {1:2%} means that the second argument should␣
↪be expressed as a percent with two decimal digits.
for col in D.columns:
    template = 'Column "{0}" has {1:.2%} missing values'
    print(template.format(col, np.isnan(D[col]).mean()))
```

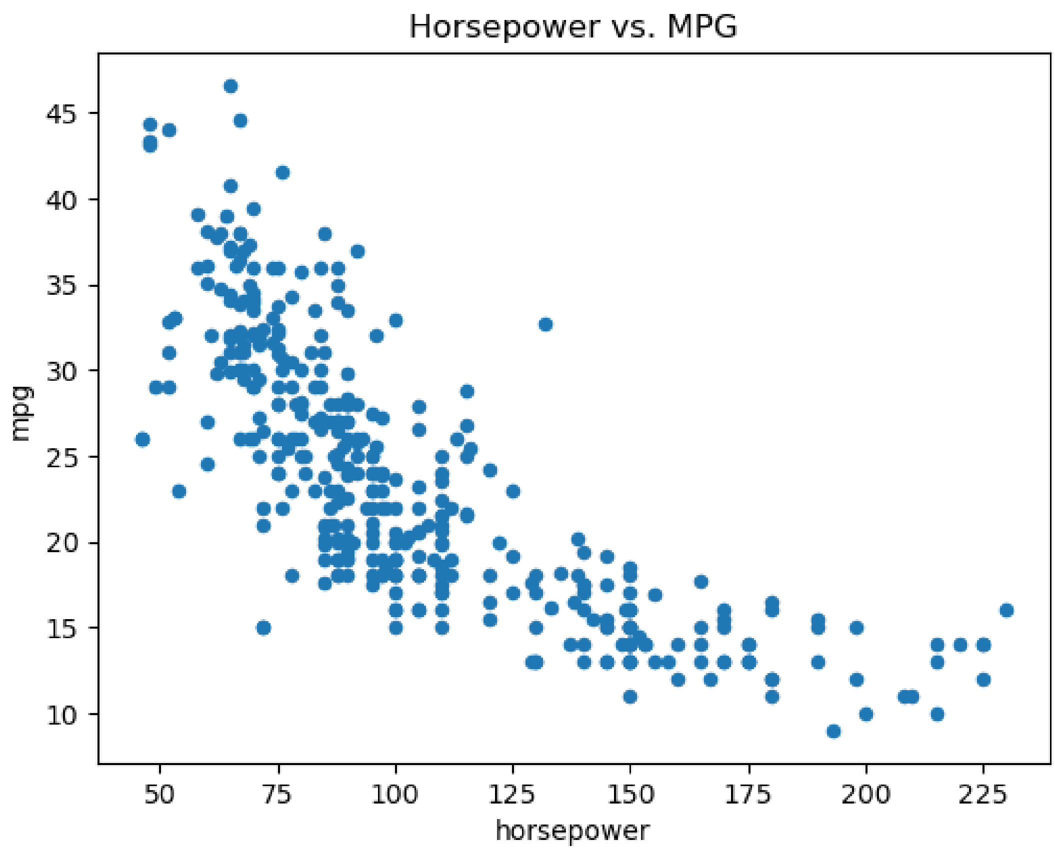
```
Column "food" has 16.54% missing values
Column "bar" has 25.98% missing values
Column "pickle" has 29.13% missing values
Column "snack" has 21.26% missing values
Column "popcorn" has 22.83% missing values
```

[ ]:
```
# To plot our imported data we need to specify
fig, ax = subplots(figsize=(8, 8))
ax.plot(Auto['horsepower'], Auto['mpg'], 'o');
```



[ ]:
```
# We can also call plot directly on Auto
ax = Auto.plot.scatter('horsepower', 'mpg');
ax.set_title('Horsepower vs. MPG')
```

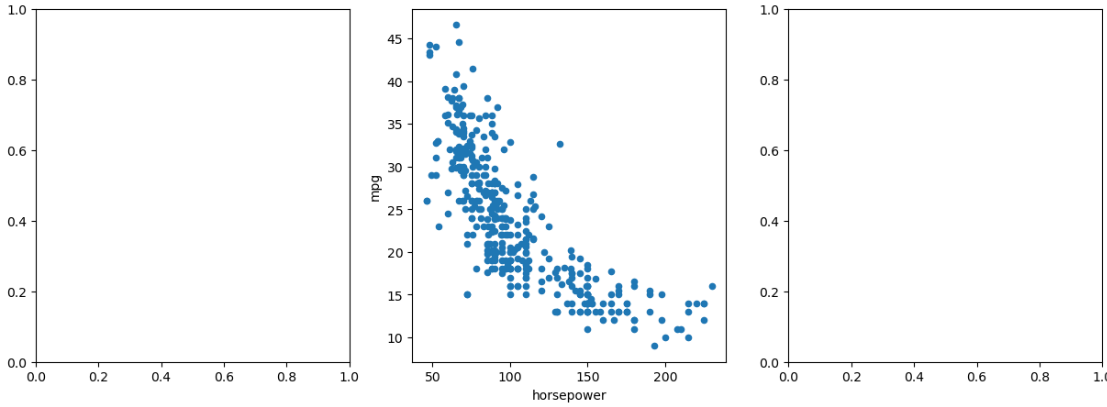
[ ]:
```
Text(0.5, 1.0, 'Horsepower vs. MPG')
```



To save the figure with the given axes we can use the `figure` attribute.

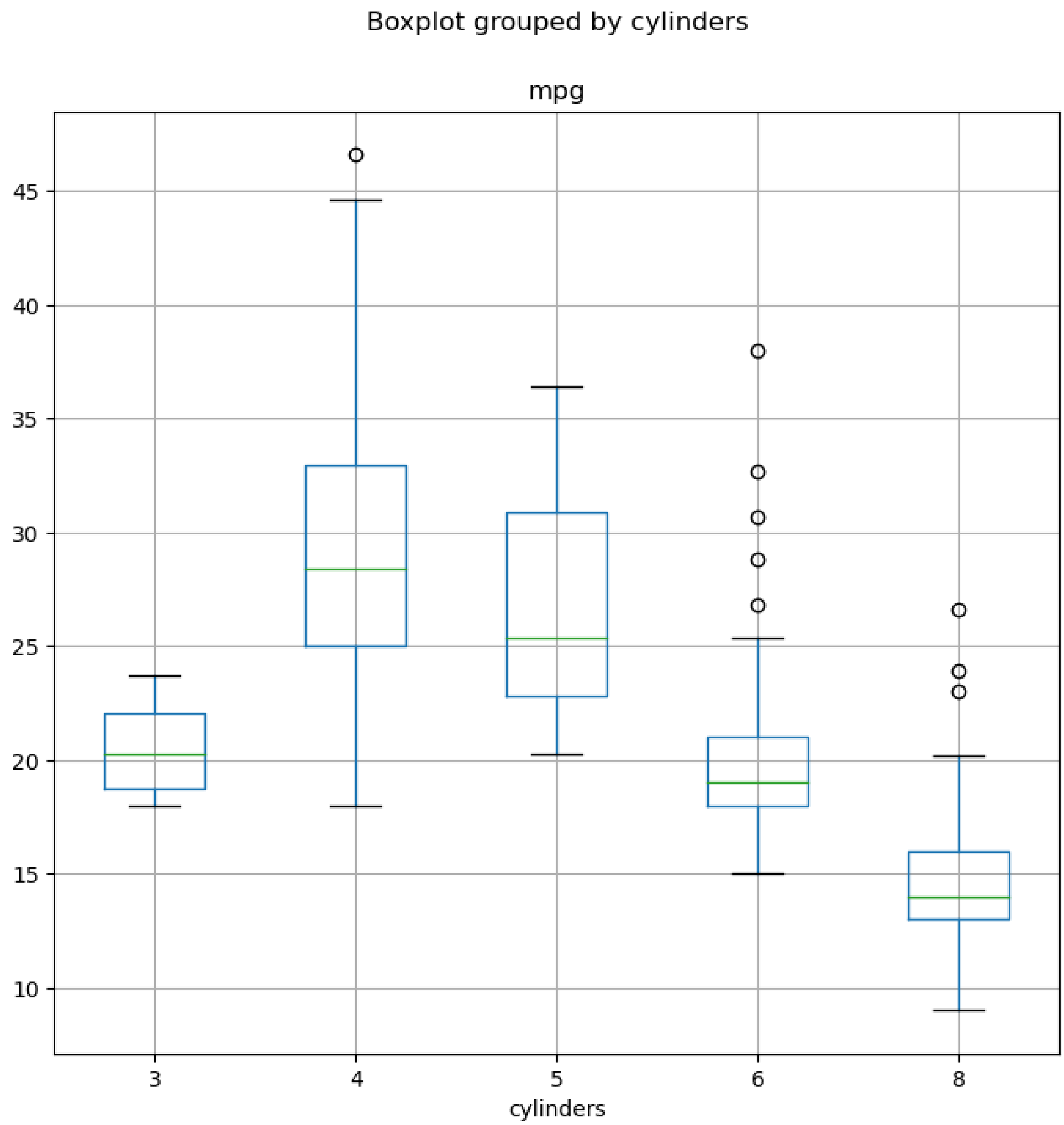
```
[ ]: fig = ax.figure
     fig.savefig('horsepower_mpg.png');
```

```
[ ]: # Getting fancier we can request a one-dimensional grid of plots and place the␣
     ↪scatterplot in the middle of the plot of a row of three plots withen the␣
     ↪figure.
     fig, axes = subplots(ncols=3, figsize=(15, 5))
     Auto.plot.scatter('horsepower', 'mpg', ax=axes[1]);

     # We could have also used the dot notation to access the columns.
```



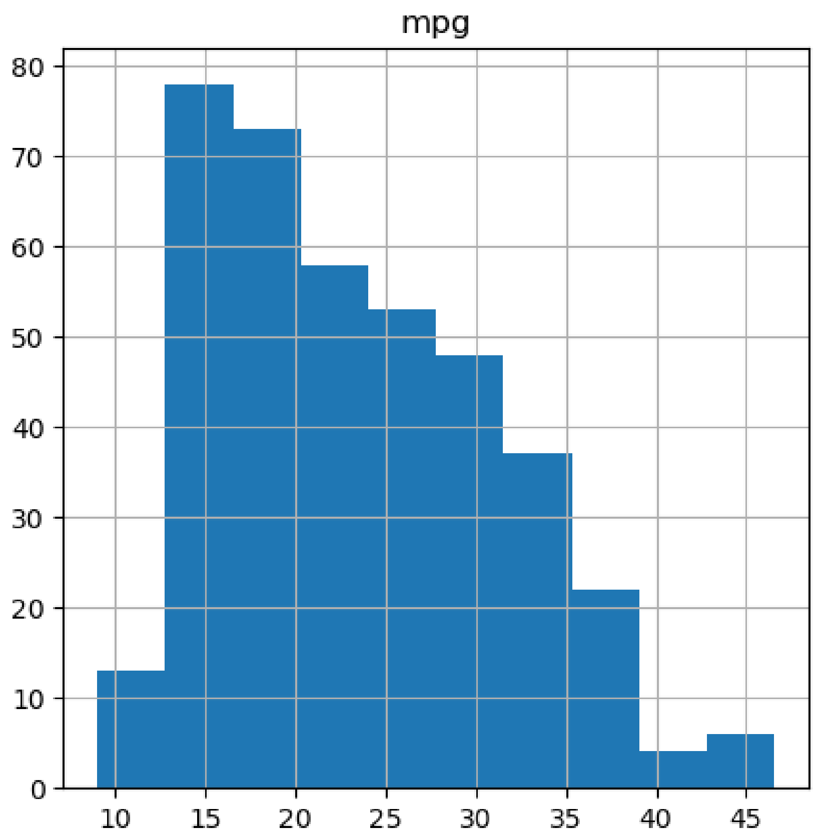
[ ]:
```python
# We can coerce quantitative data into qualitative data using dtype='category'

Auto.cylinders = pd.Series(Auto.cylinders, dtype='category')
print('Auto Cylinders Datatype:\n',Auto.cylinders.dtype)

# And now we can use boxplot with it
fig, ax = subplots(figsize=(8, 8))
Auto.boxplot('mpg', by='cylinders', ax=ax);
```

```
Auto Cylinders Datatype:
 category
```



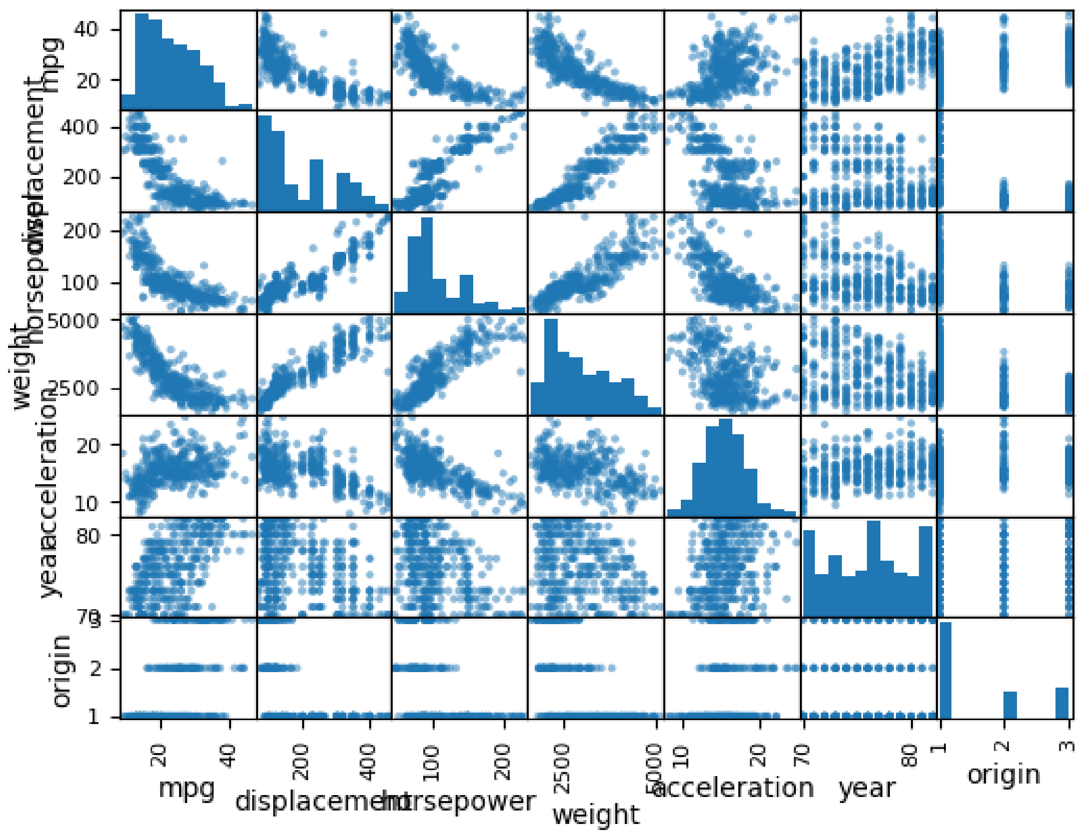

```
[ ]: # figsize tells us how to specify the bins and height of the plot
     fig, ax = subplots(figsize=(5,5))

     Auto.hist('mpg', ax =ax)
```

```
[ ]: array([<Axes: title={'center': 'mpg'}>], dtype=object)
```

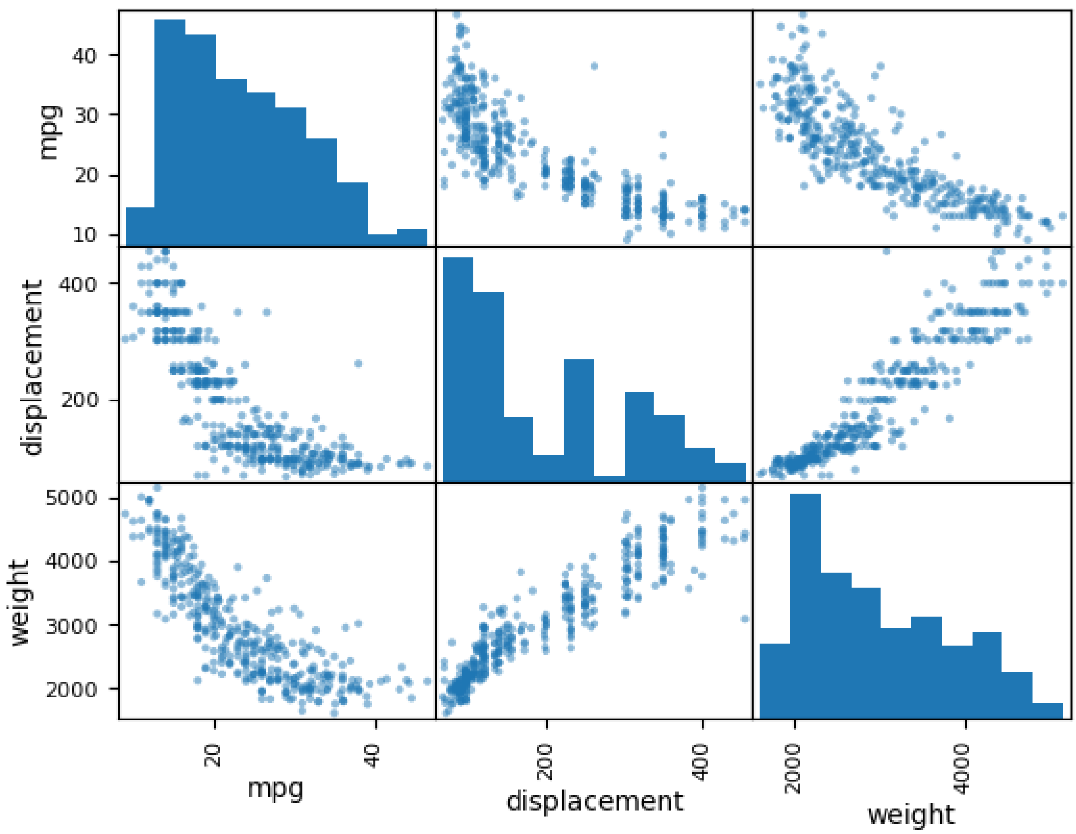



```
[ ]: # For more information
     Auto.hist?
```

```
[ ]: # To visualize all the pairwise relationships between variables we can do
     pd.plotting.scatter_matrix(Auto);
```



[ ]:
```
# And for a subset of the variables
pd.plotting.scatter_matrix(Auto[['mpg', 'displacement',
'weight']]);
```



0.2.5 Numerical Summaries

Sometimes we want to actually know the summary statistics of a data set. We can use `.describe()` function on a data frame to do so. This will return the count, mean, standard deviations as other statistics.

[ ]:
```
# Summary of multiple columns
Auto[['mpg', 'weight']].describe()
```

[ ]:

	mpg	weight
count	392.000000	392.000000
mean	23.445918	2977.584184
std	7.805007	849.402560
min	9.000000	1613.000000
25%	17.000000	2225.250000
50%	22.750000	2803.500000
75%	29.000000	3614.750000
max	46.600000	5140.000000

[ ]:

```
#Single Column
Auto['cylinders'].describe()
Auto['mpg'].describe()
```

[ ]:

```
count    392.000000
mean      23.445918
std        7.805007
min        9.000000
25%       17.000000
50%       22.750000
75%       29.000000
max       46.600000
Name: mpg, dtype: float64
```