DATA 572: Supervised Learning

Tutorial - Introduction to Python

2023W2

Shan Du

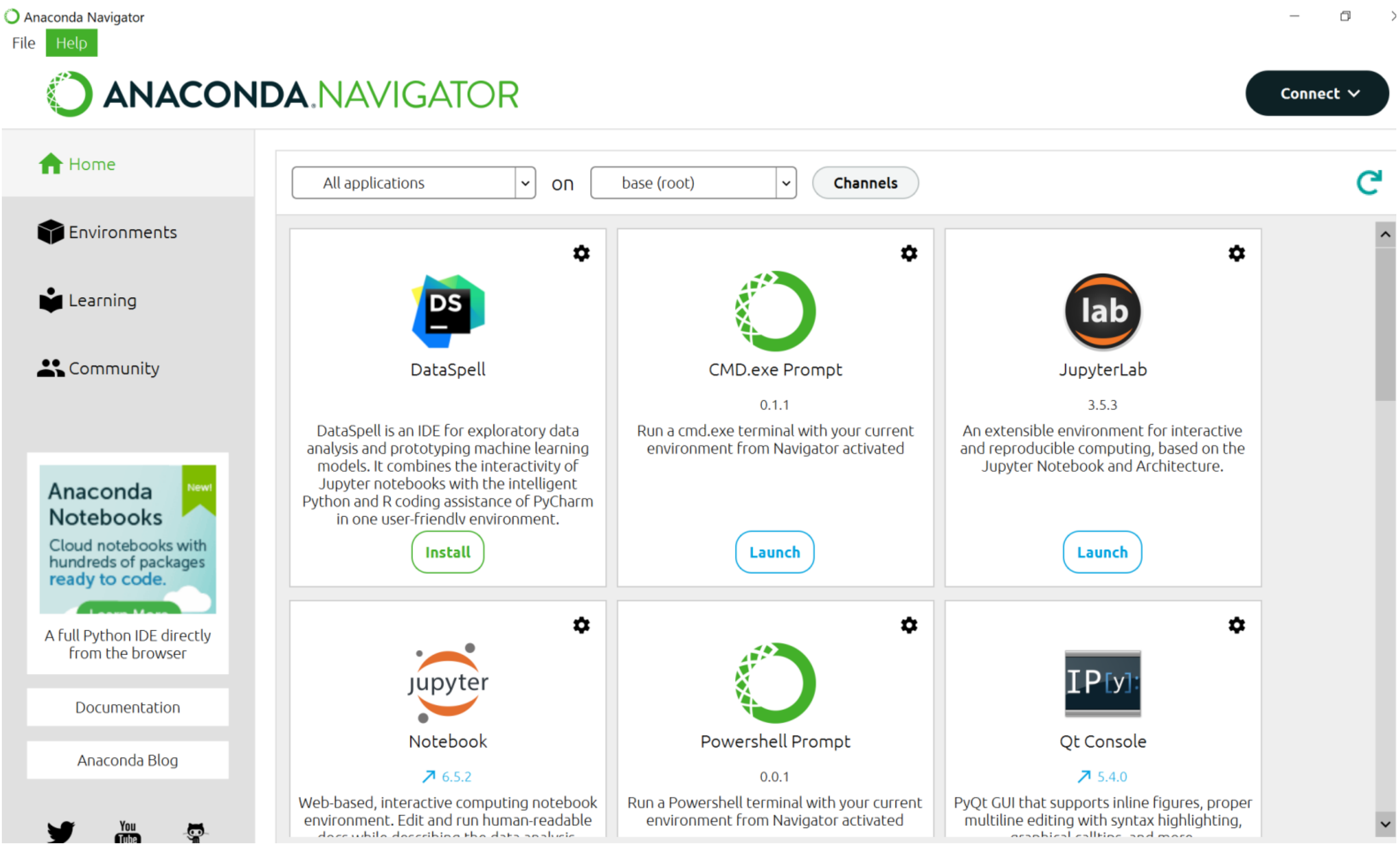
Installation

- To run the labs in this course, you will need two things:
  1. An installation of *Python3*, which is the specific version of Python used in the labs.
  2. Access to *Jupyter*, a very popular Python interface that runs code through a file called a *notebook*.

  You can download and install Python3 by following the instructions available at anaconda.com.

Installation

The Jupyter Notebook is also installed with Anaconda.

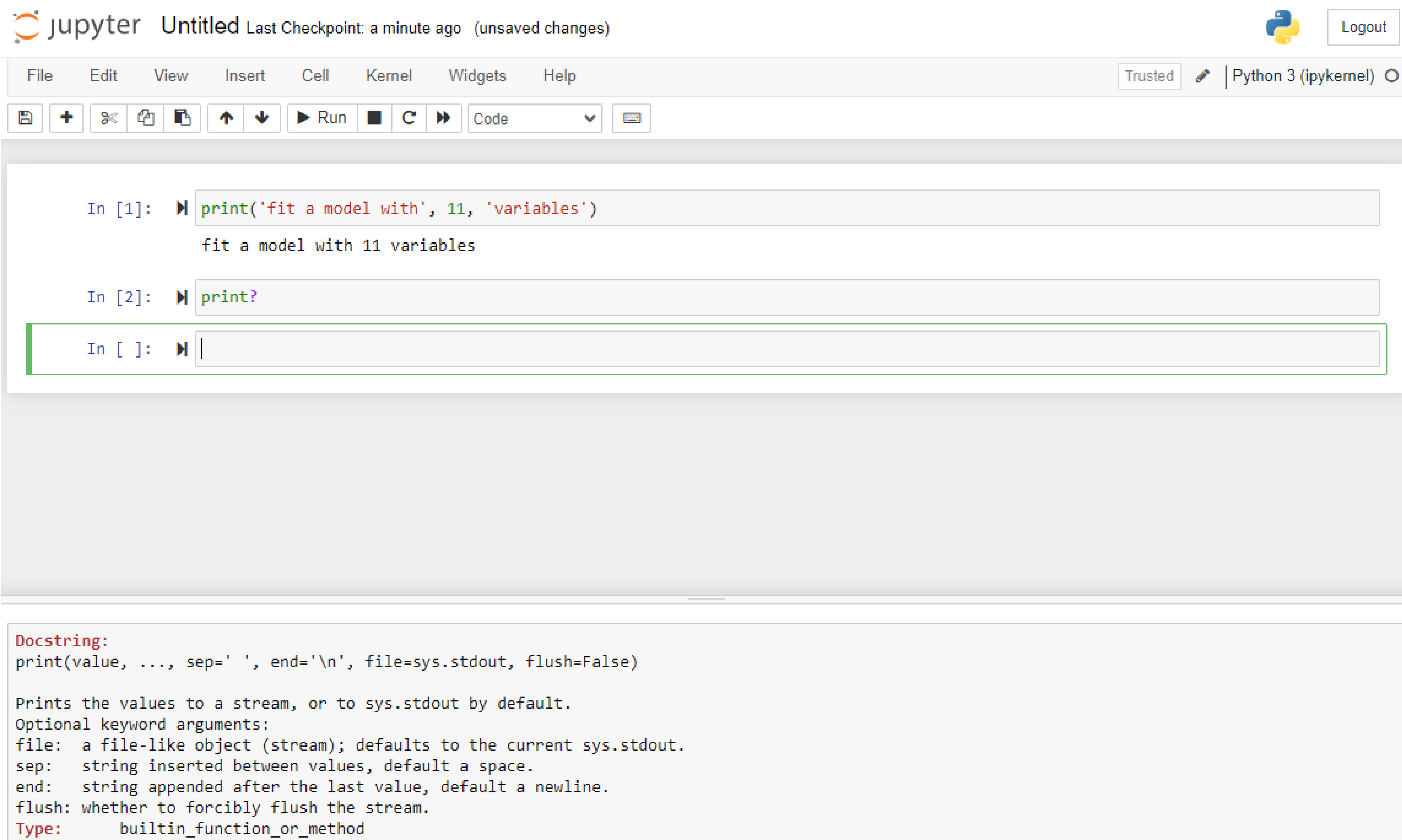


Basic Commands

- https://docs.python.org/3/tutorial/

- Like most programming languages, Python uses *functions* to perform operations. To run a function called *fun*, we type *fun(input1, input2)*, where the inputs (or arguments) *input1* and *input2* tell Python how to run the function. A function can have any number of inputs.

Basic Commands



Basic Commands

- Adding two integers

```
In [3]:  3 + 5
Out[3]: 8
```

- In Python, textual data is handled using *strings*. For instance, "hello" and 'hello' are strings.

```
In [4]:  "hello" + " " + "world"
Out[4]: 'hello world'

In [5]:  'hello' + ' ' + 'world'
Out[5]: 'hello world'
```

Basic Commands

- A string is actually a type of *sequence*: this is a generic term for an ordered list. The three most important types of sequences are lists, tuples, and strings.
- Example: Join together the numbers 3, 4, and 5, and to save them as a *list* named x.

```
In [7]:  x = [3, 4, 5]
         x
Out[7]: [3, 4, 5]
```

Note that we used the brackets [] to construct this list.

Basic Commands

In [7]:
```
x = [3, 4, 5]
x
```

Out[7]:
```
[3, 4, 5]
```

In [8]:
```
y = [4, 9, 7]
x + y
```

Basic Commands

```
In [7]:  x = [3, 4, 5]
         x

Out[7]:  [3, 4, 5]

In [8]:  y = [4, 9, 7]
         x + y

Out[8]:  [3, 4, 5, 4, 9, 7]
```

- In Python, lists hold arbitrary objects, and are added using *concatenation*. Much of Python's data-specific functionality comes from other packages, notably *numpy* and *pandas*.

Introduction to Numerical Python

- A package is a collection of modules that are not necessarily included in the base Python distribution.
- The *numpy library*, or *package* is an abbreviation for *numerical Python*.
- To access numpy, we must first *import* it.

```
In [9]:    import numpy as np
```

Introduction to Numerical Python

- In numpy, an *array* is a generic term for a multidimensional set of numbers. We use the *np.array()* function to define *x* and *y*, which are one-dimensional arrays, i.e., vectors.

```
In [10]:  x = np.array([3, 4, 5])
          y = np.array([4, 9, 7])

In [11]:  x + y

Out[11]:  array([ 7, 13, 12])
```

- The syntax *np.array()* indicates that the function being called is part of the *numpy* package, which we have abbreviated as *np*.

Introduction to Numerical Python

- In *numpy*, matrices are typically represented as two-dimensional arrays, and vectors as one-dimensional arrays. We can create a two-dimensional array as follows.

```
In [12]:  x = np.array([[1, 2], [3, 4]])
          x

Out[12]:  array([[1, 2],
                 [3, 4]])
```

Introduction to Numerical Python

- The object *x* has several attributes, or associated objects. To access an attribute of *x*, we type *x.attribute*, where we replace attribute with the name of the attribute. For instance, we can access the *ndim* attribute of *x* as follows.

```
In [13]:  x.ndim

Out[13]: 2
```

Introduction to Numerical Python

- The output indicates that *x* is a two-dimensional array. Similarly, *x.dtype* is the *data type* attribute of the object *x*. This indicates that *x* is comprised of 32-bit integers:

```
In [14]:  x.dtype

Out[14]:  dtype('int32')
```

- This is because we created *x* by passing in exclusively integers to the *np.array()* function.

Introduction to Numerical Python

- If we had passed in any decimals, then we would have obtained an array of *floating point numbers*.

```
In [15]:  np.array([[1, 2], [3.0, 4]]).dtype

Out[15]:  dtype('float64')
```

- We can create a floating point array by passing a *dtype* argument into *np.array().dtype*

```
In [17]:  np.array([[1, 2], [3, 4]], float).dtype

Out[17]:  dtype('float64')
```

Introduction to Numerical Python

- The array *x* is two-dimensional. We can find out the number of rows and columns by looking at its *shape* attribute.

```
In [18]:  x.shape

Out[18]: (2, 2)
```

- A *method* is a function that is associated with an object. For instance, given an array method *x*, the expression *x.sum()* sums all of its elements, using the *sum()* method for arrays. The call *x.sum()* automatically provides *x* as the first argument to its *sum()* method.

```
In [19]:  x = np.array([1, 2, 3, 4])
          x.sum()

Out[19]: 10
```

Introduction to Numerical Python

- We could also sum the elements of *x* by passing in *x* as an argument to the *np.sum()* function.

```
In [20]:  x = np.array([1, 2, 3, 4])
          np.sum(x)

Out[20]: 10
```

Introduction to Numerical Python

- The *reshape()* method returns a new array with the same elements as *x*, but a different shape. We do this by passing in a tuple in our call to *reshape()*, in this case (2, 3). This tuple specifies that we would like to create a two-dimensional array with 2 rows and 3 columns.

In [21]:
```
x = np.array([1, 2, 3, 4, 5, 6])
print('beginning x:\n', x)
x_reshape = x.reshape((2, 3))
print('reshaped x:\n', x_reshape)
```

```
beginning x:
 [1 2 3 4 5 6]
reshaped x:
 [[1 2 3]
 [4 5 6]]
```

Introduction to Numerical Python

- *numpy* arrays are specified as a sequence of *rows*. This is called *row-major ordering*, as opposed to *column-major ordering*.
- Python (and hence numpy) uses *0-based indexing*. This means that to access the top left element of *x_reshape*, we type in *x_reshape[0,0]*.

```
In [22]:  x_reshape[0, 0]

Out[22]: 1
```

Introduction to Numerical Python

- Modifying *x_reshape* also modified *x* because the two objects occupy the same space in memory.

In [23]:
```
print('x before we modify x_reshape:\n', x)
print('x_reshape before we modify x_reshape:\n', x_reshape)
x_reshape[0, 0] = 5
print('x_reshape after we modify its top left element:\n',
x_reshape)
print('x after we modify top left element of x_reshape:\n', x)
```

```
x before we modify x_reshape:
 [1 2 3 4 5 6]
x_reshape before we modify x_reshape:
 [[1 2 3]
 [4 5 6]]
x_reshape after we modify its top left element:
 [[5 2 3]
 [4 5 6]]
x after we modify top left element of x_reshape:
 [5 2 3 4 5 6]
```

Introduction to Numerical Python

```
In [25]: x_reshape.shape , x_reshape.ndim , x_reshape.T

Out[25]: ((2, 3),
          2,
          array([[5, 4],
                 [2, 5],
                 [3, 6]]))
```

```
In [26]: np.sqrt(x)

Out[26]: array([2.23606798, 1.41421356, 1.73205081, 2.        , 2.23606798,
                2.44948974])
```

```
In [27]: x**2

Out[27]: array([25,  4,  9, 16, 25, 36])
```

```
In [28]: x**0.5

Out[28]: array([2.23606798, 1.41421356, 1.73205081, 2.        , 2.23606798,
                2.44948974])
```

Introduction to Numerical Python

- Generate random data: *np.random.normal()* function generates a vector of random normal variables.

```
In [29]:  np.random.normal?
```

```
Docstring:
normal(loc=0.0, scale=1.0, size=None)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first
derived by De Moivre and 200 years later by both Gauss and Laplace
independently [2]_, is often called the bell curve because of
its characteristic shape (see the example below).

The normal distributions occurs often in nature.  For example, it
describes the commonly occurring distribution of samples influenced
by a large number of tiny, random disturbances, each with its own
```

Introduction to Numerical Python

- By default, this function will generate random normal variable(s) with mean (*loc*) 0 and standard deviation (*scale*) 1; furthermore, a single random variable will be generated unless the argument to *size* is changed.

```
In [30]:  x = np.random.normal(size=50)

In [31]:  x

Out[31]: array([ 1.016527  ,  0.29630926,  0.96134982,  0.31023133,  0.52177376,
                -0.91979143, -0.92215166, -1.14418682, -0.37095437, -1.09116511,
                -0.29749639, -0.35213256,  0.3245657 ,  1.0522413 ,  0.43113357,
                 0.48673004, -0.68270534, -0.19172473,  0.3020319 ,  0.18085425,
                -1.21985079,  0.29017371, -0.07931557,  0.84370524, -0.75146806,
                -1.06603679, -0.28243178, -0.73068695, -1.9812266 ,  0.47570591,
                 0.51699622, -0.37154697,  1.02783947,  0.82409953,  2.16533796,
                 0.77911666, -0.13074178,  0.20675057,  0.16049944, -0.43860181,
                 1.8335012 , -1.20391787, -0.86482857,  0.09080541, -0.52948043,
                 0.33561294,  0.34369016, -0.72590738, -0.14139   ,  1.38267245])
```

Introduction to Numerical Python

- We create an array *y* by adding an independent *N(50, 1)* random variable to each element of *x*.

```
In [34]: y = x + np.random.normal(loc=50, scale=1, size=50

In [35]: y

Out[35]: array([51.55267266, 50.76943981, 50.28593868, 50.77374953, 50.71122293,
                48.44601465, 48.43442275, 49.13620969, 50.01376573, 50.28511802,
                49.19032844, 50.00127415, 50.4109918 , 50.5895951 , 49.70425265,
                51.75148667, 48.88730397, 50.50671756, 50.3920225 , 49.87938496,
                49.42776672, 49.48997055, 50.75693897, 51.32385406, 49.26047168,
                46.28403247, 48.81168948, 49.33361401, 49.13870904, 50.61095999,
                50.4886592 , 50.05162636, 51.05819653, 49.63347628, 51.74098487,
                49.98854726, 49.60876822, 49.57216622, 50.23926914, 49.43007753,
                52.15660199, 48.46925995, 49.83051347, 48.80395311, 48.79012662,
                49.83480167, 50.46191154, 47.68905408, 49.55324324, 52.87029713])
```

Introduction to Numerical Python

- The *np.corrcoef()* function computes the correlation matrix between *x* and *y*. The off-diagonal elements give the correlation between *x* and *y*.

```
In [36]:  np.corrcoef(x, y)

Out[36]: array([[1.        , 0.74243579],
                [0.74243579, 1.        ]])
```

Introduction to Numerical Python

- In order to ensure that our code provides exactly the same results each time it is run, we can set a random seed using the *np.random.default_rng()* function. This function takes an arbitrary, user-specified integer argument. If we set a random seed before generating random data, then re-running our code will yield the same results.

Introduction to Numerical Python

- The *np.mean(), np.var(),* and *np.std()* functions can be used to compute the mean, variance, and standard deviation of arrays.

```
In [37]: rng = np.random.default_rng(3)
         y = rng.standard_normal(10)
         np.mean(y), y.mean()

Out[37]: (-0.1126795190952861, -0.1126795190952861)

In [38]: np.var(y), y.var(), np.mean((y - y.mean())**2)

Out[38]: (2.7243406406465125, 2.7243406406465125, 2.7243406406465125)
```

Introduction to Numerical Python

- The *np.mean(), np.var(),* and *np.std()* functions can also be applied to the rows and columns of a matrix. To see this, we construct a *10×3* matrix of *N(0, 1)* random variables, and consider computing its row sums.

```
In [39]:  X = rng.standard_normal((10, 3))
          X

Out[39]: array([[ 0.22578661, -0.35263079, -0.28128742],
                [-0.66804635, -1.05515055, -0.39080098],
                [ 0.48194539, -0.23855361,  0.9577587 ],
                [-0.19980213,  0.02425957,  1.54582085],
                [ 0.54510552, -0.50522874, -0.18283897],
                [ 0.54052513,  1.93508803, -0.26962033],
                [-0.24355868,  1.0023136 , -0.88645994],
                [-0.29172023,  0.88253897,  0.58035002],
                [ 0.0915167 ,  0.67010435, -2.82816231],
                [ 1.02130682, -0.95964476, -1.66861984]])
```

Introduction to Numerical Python

- Since arrays are row-major ordered, the first axis, i.e., *axis=0*, refers to its rows. We pass this argument into the mean() method for the object *X*.

```
In [40]:  X.mean(axis=0)

Out[40]: array([ 0.15030588,  0.14030961, -0.34238602])

In [41]:  X.mean(0)

Out[41]: array([ 0.15030588,  0.14030961, -0.34238602])
```

Graphics

- In Python, common practice is to use the library *matplotlib* for graphics. However, since Python was not written with data analysis in mind, the notion of plotting is not intrinsic to the language. We will use the *subplots()* function from *matplotlib.pyplot* to create a figure and the axes onto which we plot our data.

    matplotlib.org/stable/gallery/

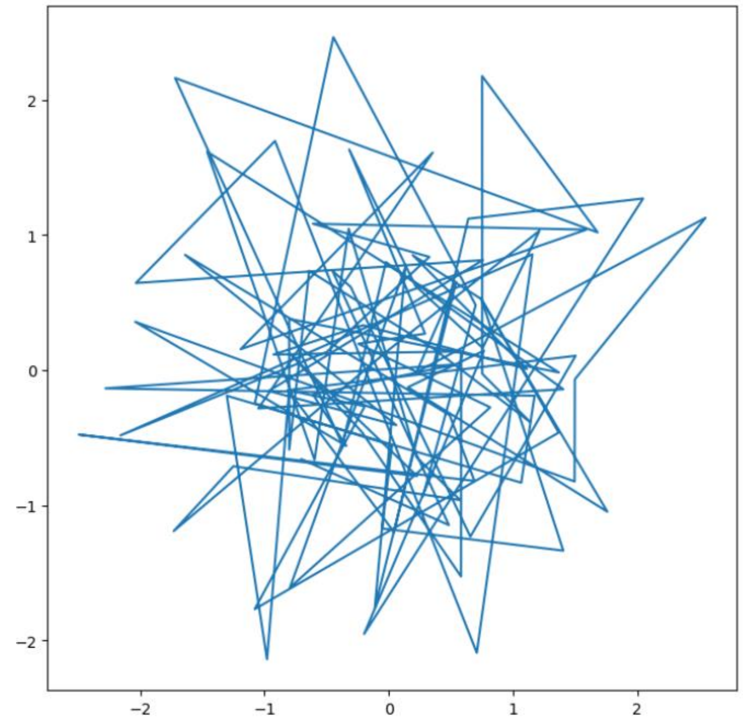
Graphics

- We begin by importing the *subplots()* function from *matplotlib*.
- The function returns a tuple of length two: a figure object as well as the relevant axes object.
- We will typically pass *figsize* as a keyword argument. Having created our axes, we attempt our first plot using its *plot()* method. To learn more about it, type *ax.plot?*.

Graphics

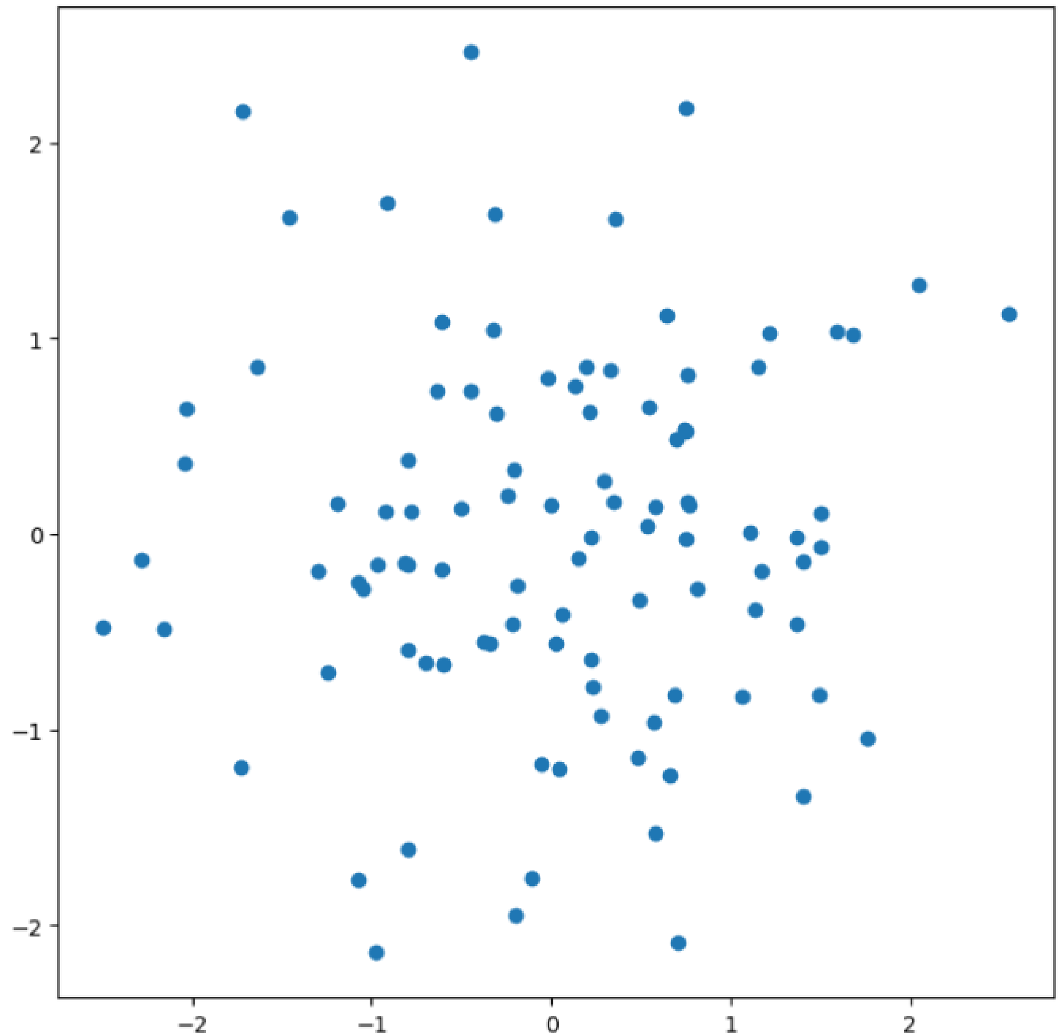
In [42]:

```
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);
```



Graphics

In [44]:
```
fig , ax = subplots(figsize=(8, 8))
ax.plot(x, y, 'o');
```

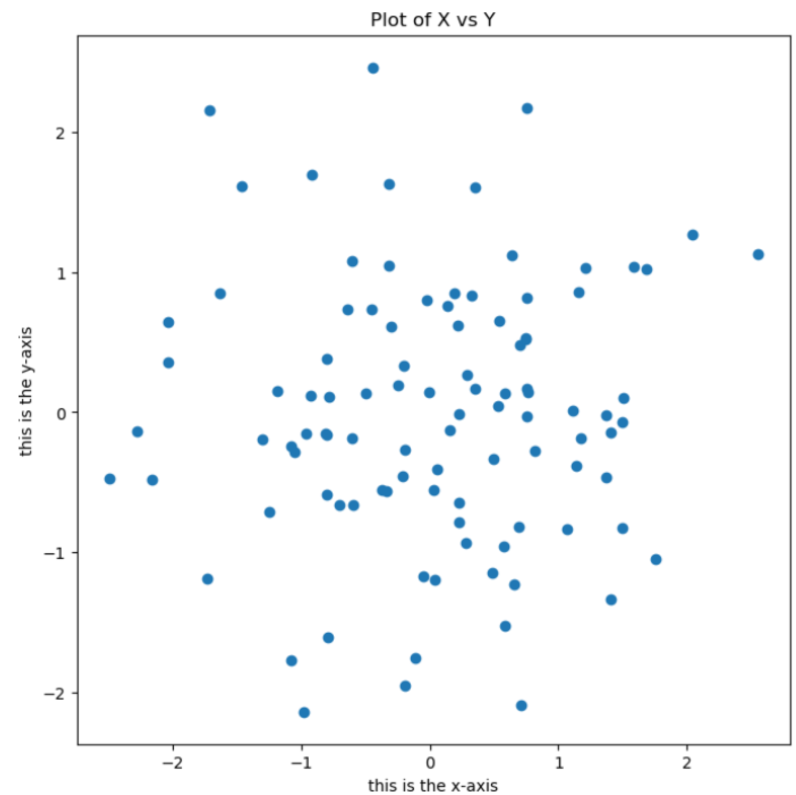


Graphics

- To label our plot, we make use of the *set_xlabel(), set_ylabel(),* and *set_title()* methods of *ax*.

In [47]:

```
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");
```

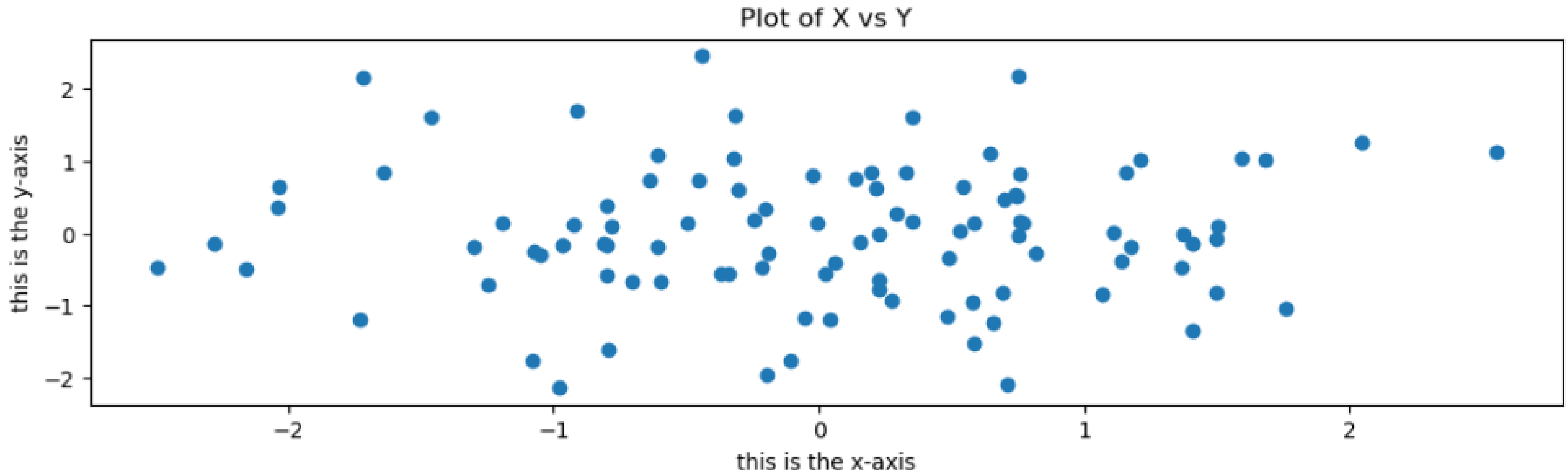


Graphics

- Having access to the figure object *fig* itself means that we can go in and change some aspects and then redisplay it. Here, we change the size from (8, 8) to (12, 3).

In [48]:
```
fig.set_size_inches (12,3)
fig
```

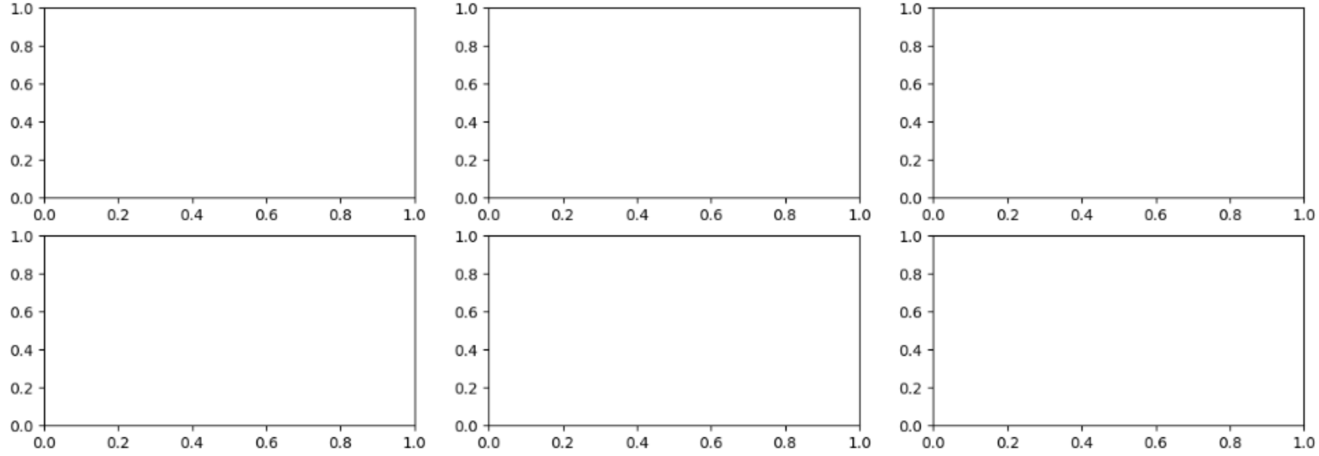
Out[48]:



Graphics

- If we want to create several plots within a figure. This can be achieved by passing additional arguments to *subplots()*. Below, we create a 2×3 grid of plots in a figure of size determined by the *figsize* argument.

In [49]:
```
fig , axes = subplots(nrows=2,
ncols=3,
figsize=(15, 5))
```

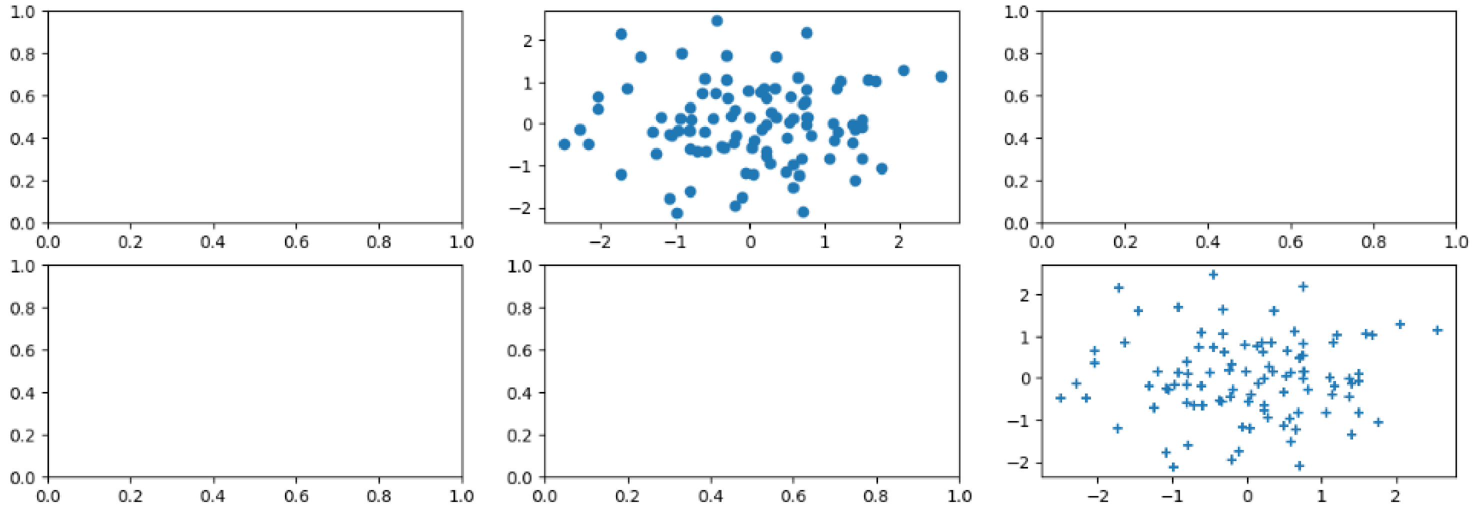


Graphics

- We now produce a scatter plot with 'o' in the second column of the first row and a scatter plot with '+' in the third column of the second row.

In [52]:
```
axes[0,1].plot(x, y, 'o')
axes[1,2].scatter(x, y, marker='+')
fig
```

Out[52]:



Graphics

- To save the output of *fig*, we call its *savefig()* method. The argument *dpi* is the dots per inch, used to determine how large the figure will be in pixels.

In [53]:

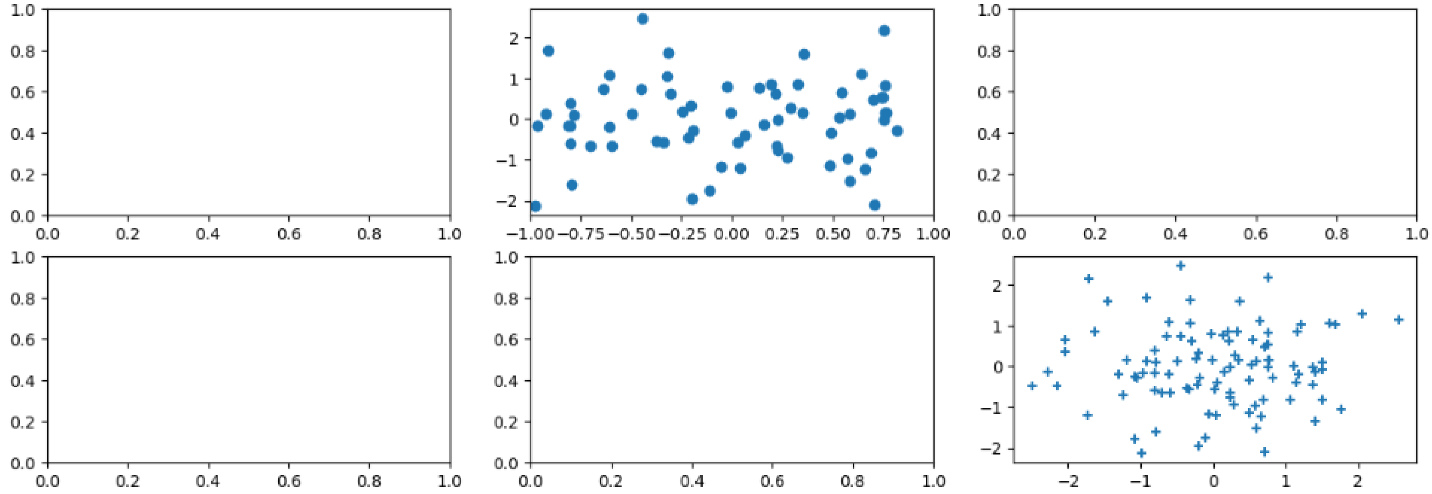
```
fig.savefig("Figure.png", dpi=400)
fig.savefig("Figure.pdf", dpi=200);
```

Graphics

- We can continue to modify *fig* using step-by-step updates; for example, we can modify the range of the x-axis, re-save the figure, and even re-display it.

In [54]:
```
axes[0,1].set_xlim([-1,1])
fig.savefig("Figure_updated.jpg")
fig
```

Out[54]:



Graphics

- The *ax.contour()* method produces a *contour plot* in order to represent three-dimensional data, similar to a topographical map. It takes three arguments:
    - A vector of *x* values (the first dimension),
    - A vector of *y* values (the second dimension),

and

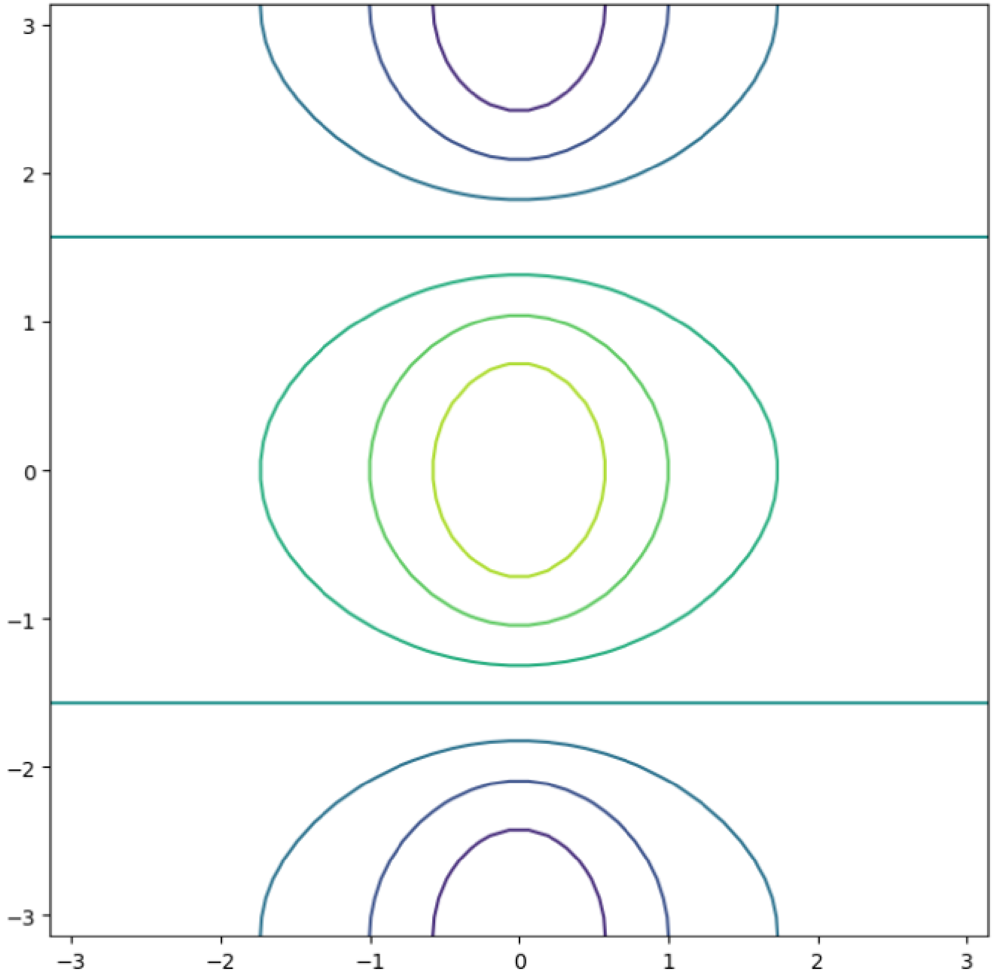
- A matrix whose elements correspond to the

z value (the third dimension) for each pair of (*x,y*) coordinates.

Graphics

In [55]:

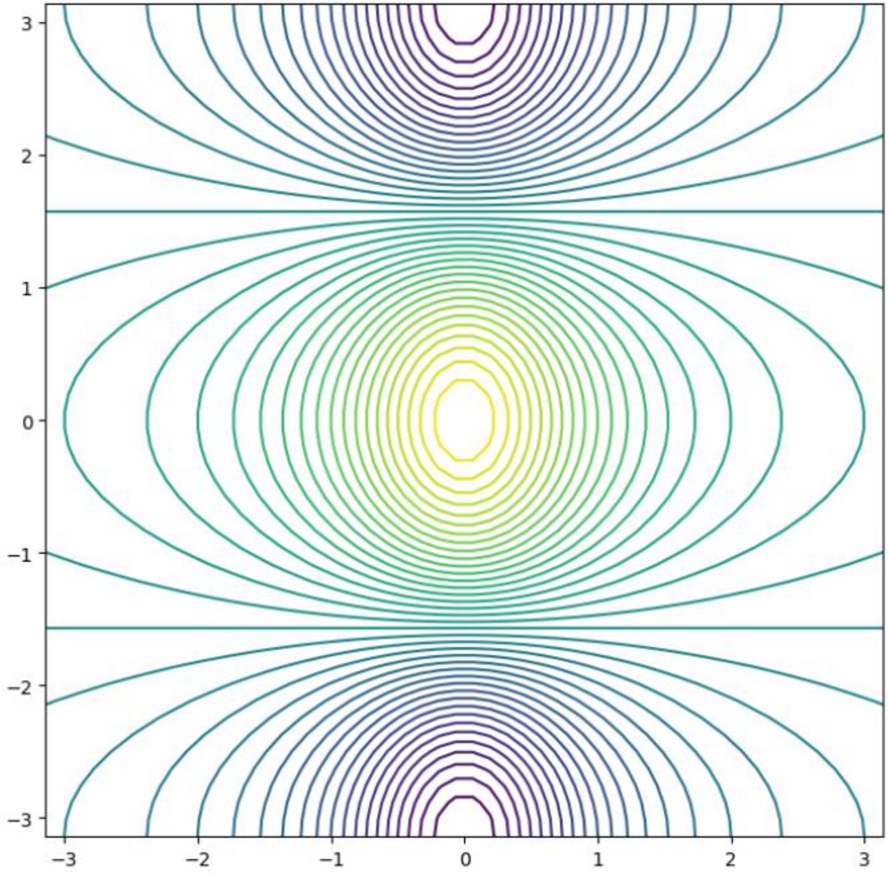
```
fig , ax = subplots(figsize=(8, 8))
x = np.linspace(-np.pi, np.pi, 50)
y = x
f = np.multiply.outer(np.cos(y), 1 / (1 + x**2))
ax.contour(x, y, f);
```



Graphics

- We can increase the resolution by adding more levels to the image.

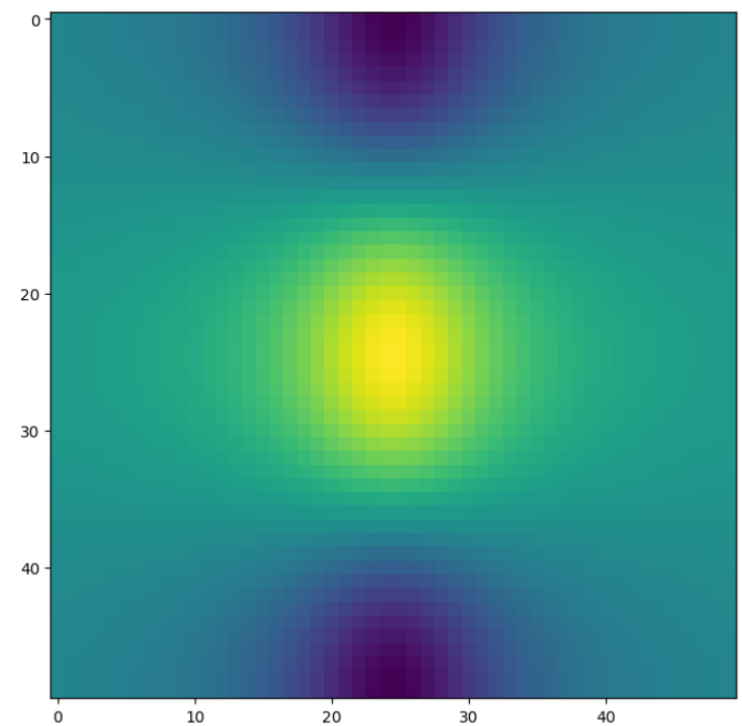
In [56]:
```
fig , ax = subplots(figsize=(8, 8))
ax.contour(x, y, f, levels=45);
```



Graphics

- The *ax.imshow()* method is similar to *ax.contour()*, except that it produces a color-coded plot whose colors depend on the *z* value. This is known as a *heatmap*, and is sometimes used to plot temperature in weather forecasts.

```
In [57]: fig , ax = subplots(figsize=(8, 8))
         ax.imshow(f);
```



Sequences and Slice Notation

- The function *np.linspace()* can be used to create a sequence of numbers.

```
In [53]: seq1 = np.linspace(0, 10, 11)
         seq1

Out[53]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

- The function *np.arange()* returns a sequence of numbers spaced out by *step*. If *step* is not specified, then a default value of 1 is used.

```
In [54]: seq2 = np.arange(0, 10)
         seq2

Out[54]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Sequences and Slice Notation

- Slice notation is used to index sequences such as lists, tuples and arrays.
- Suppose we want to retrieve the fourth through sixth (inclusive) entries of a string. We obtain a slice of the string using the indexing notation [3:6].

```
In [55]:  "hello world"[3:6]

Out[55]: 'lo '
```

which is the same as "hello world"[slice(3,6)]

Indexing Data

- Create a two-dimensional *numpy* array:

```
In [56]:  A = np.array(np.arange(16)).reshape((4, 4))
          A

Out[56]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])
```

- Typing A[1,2] retrieves the element corresponding to the second row and third column. (Python indexes from 0.)

```
In [58]:  A[1,2]

Out[58]: 6
```

Indexing Rows, Columns, and Submatrices

- To select multiple rows at a time, we can pass in a list specifying our selection. For instance, [1,3] will retrieve the second and fourth rows:

```
In [62]:  A[[1,3]]

Out[62]: array([[ 4,  5,  6,  7],
                [12, 13, 14, 15]])
```

- To select the first and third columns, we pass in [0,2] as the second argument in the square brackets. In this case we need to supply the first argument : which selects all rows.

```
In [63]:  A[:,[0,2]]

Out[63]: array([[ 0,  2],
                [ 4,  6],
                [ 8, 10],
                [12, 14]])
```

Indexing Rows, Columns, and Submatrices

- Suppose we want to select the submatrix made up of the second and fourth rows as well as the first and third columns.

```
In [64]:  A[[1,3],[0,2]]

Out[64]: array([ 4, 14])
```

- One easy way to do this is as follows. We first create a submatrix by subsetting the rows of A, and then on the fly we make a further submatrix by subsetting its columns.

```
In [65]:  A[[1,3]][:,[0,2]]

Out[65]: array([[ 4,  6],
                [12, 14]])
```

Indexing Rows, Columns, and Submatrices

- The *convenience* function *np.ix_()* allows us to extract a submatrix using lists, by creating an intermediate *mesh* object.

```
idx = np.ix_([1,3],[0,2,3])
A[idx]
```

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

- Alternatively, we can subset matrices efficiently using slices. The slice 1:4:2 captures the second and fourth items of a sequence, while the slice 0:3:2 captures the first and third items (the third element in a slice sequence is the step size).

```
A[1:4:2,0:3:2]
```

```
array([[ 4,  6],
       [12, 14]])
```

Boolean Indexing

- Boolean is a type that equals either *True* or *False* (also represented as Boolean 1 and 0, respectively).

```
keep_rows = np.zeros(A.shape[0], bool)
keep_rows
```

```
array([False,  False, False,  False])
```

- We now set two of the elements to *True*.

```
keep_rows[[1,3]] = True
keep_rows
```

```
array([False,  True, False,  True])
```

Boolean Indexing

- Note that the elements of *keep_rows*, when viewed as integers, are the same as the values of *np.array([0,1,0,1])*. We can use == to verify their equality.

```
np.all(keep_rows == np.array([0,1,0,1]))
```

```
True
```

- The function *np.all()* has checked whether all entries of an array are *True*. A similar function, *np.any()*, can be used to check whether any entries of an array are *True*.)

Loading Data

- Data sets often contain different types of data, and may have names associated with the rows or columns.
- For these reasons, they typically are best accommodated using a *data frame*. We can think of a data frame as a sequence of arrays of identical length; these are the columns. Entries in data frame the different arrays can be combined to form a row.
- The *pandas* library can be used to create and work with data frame objects.

Reading in a Data Set

- Before attempting to load a data set, we must make sure that *Python* knows where to find the file containing it.
- If the file is in the same location as this notebook file, then we are all set. Otherwise, the command *os.chdir()* can be used to change directory. (You will need to call import *os* before calling *os.chdir().*)

Reading in a Data Set

In [66]:
```python
import pandas as pd
Auto = pd.read_csv('Auto.csv')
Auto
```

Out[66]:

	mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	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
...	...	...	...	...	...	...	...	...	...
392	27.0	4	140.0	86	2790	15.6	82	1	ford mustang gl
393	44.0	4	97.0	52	2130	24.6	82	2	vw pickup
394	32.0	4	135.0	84	2295	11.6	82	1	dodge rampage
395	28.0	4	120.0	79	2625	18.6	82	1	ford ranger
396	31.0	4	119.0	82	2720	19.4	82	1	chevy s-10

397 rows × 9 columns

Reading in a Data Set

- The whitespace-delimited version of Auto.csv, called Auto.data can be read in using pd.read_csv():

```
In [68]:  Auto = pd.read_csv('Auto.data', delim_whitespace=True)
```

- We now take a look at the column of *Auto* corresponding to the variable *horsepower*:

```
In [69]:  Auto['horsepower']

Out[69]: 0      130.0
         1      165.0
         2      150.0
         3      150.0
         4      140.0
                ...
         392    86.00
         393    52.00
         394    84.00
         395    79.00
         396    82.00
         Name: horsepower, Length: 397, dtype: object
```

Reading in a Data Set

- We see that the *dtype* of this column is *object*. It turns out that all values of the *horsepower* column were interpreted as strings when reading in the data. The culprit is the value ? which is being used to encode missing values

```
In [70]:  np.unique(Auto['horsepower'])

Out[70]: array(['100.0', '102.0', '103.0', '105.0', '107.0', '108.0', '110.0',
                '112.0', '113.0', '115.0', '116.0', '120.0', '122.0', '125.0',
                '129.0', '130.0', '132.0', '133.0', '135.0', '137.0', '138.0',
                '139.0', '140.0', '142.0', '145.0', '148.0', '149.0', '150.0',
                '152.0', '153.0', '155.0', '158.0', '160.0', '165.0', '167.0',
                '170.0', '175.0', '180.0', '190.0', '193.0', '198.0', '200.0',
                '208.0', '210.0', '215.0', '220.0', '225.0', '230.0', '46.00',
                '48.00', '49.00', '52.00', '53.00', '54.00', '58.00', '60.00',
                '61.00', '62.00', '63.00', '64.00', '65.00', '66.00', '67.00',
                '68.00', '69.00', '70.00', '71.00', '72.00', '74.00', '75.00',
                '76.00', '77.00', '78.00', '79.00', '80.00', '81.00', '82.00',
                '83.00', '84.00', '85.00', '86.00', '87.00', '88.00', '89.00',
                '90.00', '91.00', '92.00', '93.00', '94.00', '95.00', '96.00',
                '97.00', '98.00', '?'], dtype=object)
```

Reading in a Data Set

- To fix the problem, we must provide *pd.read_csv()* with an argument called *na_values*. Now, each instance of *? (missing values)* in the file is replaced with the value *np.nan*, which means *not a number*:

```
In [71]:  Auto = pd.read_csv('Auto.data',
          na_values=['?'],
          delim_whitespace=True)
          Auto['horsepower'].sum()

Out[71]:  40952.0
```

Reading in a Data Set

- The *Auto.shape* attribute tells us that the data has 397 observations, or rows, and nine variables, or columns.

```
In [72]:  Auto.shape

Out[72]:  (397, 9)
```

- There are various ways to deal with missing data. In this case, since only five of the rows contain missing observations, we choose to use the *Auto.dropna()* method to simply remove these rows.

```
Auto_new = Auto.dropna()
Auto_new.shape
```

```
(392, 9)
```

Basics of Selecting Rows and Columns

- We can use *Auto.columns* to check the variable names.

```
Auto = Auto_new # overwrite the previous value
Auto.columns
```

```
Index(['mpg', 'cylinders', 'displacement', 'horsepower',
       'weight', 'acceleration', 'year', 'origin', 'name'],
       dtype='object')
```

- The first argument to the [ ] method is always applied to the rows of the array. Similarly, passing in a slice to the [ ] method creates a data frame whose rows are determined by the slice:

```
Auto[:3]
```

	mpg	cylinders	displacement	horsepower	weight	...
0	18.0	8	307.0	130.0	3504.0	...
1	15.0	8	350.0	165.0	3693.0	...
2	18.0	8	318.0	150.0	3436.0	...

Basics of Selecting Rows and Columns

- An array of Booleans can be used to subset the rows:

```
idx_80 = Auto['year'] > 80
Auto[idx_80]
```

- If we pass in a list of strings to the [ ] method, then we obtain a data frame containing the corresponding set of 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
```

Basics of Selecting Rows and Columns

- Since we did not specify an *index* column when we loaded our data frame, the rows are labeled using integers 0 to 396.

```
Auto.index
```

```
Int64Index([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,
            ...
            387, 388, 389, 390, 391, 392, 393, 394, 395, 396],
           dtype='int64', length=392)
```

- We can use the *set_index()* method to re-name the rows using the contents of Auto['name’]. The index has been set to *name*.

```
Auto_re = Auto.set_index('name')
Auto_re
```

```
                            mpg   cylinders   displacement   ...
                     name
chevrolet chevelle malibu   18.0          8          307.0   ...
        buick skylark 32    15.0          8          350.0   ...
      plymouth satellite    18.0          8          318.0   ...
           amc rebel sst    16.0          8          304.0   ...
```

```
Auto_re.columns
```

```
Index(['mpg', 'cylinders', 'displacement', 'horsepower',
       'weight', 'acceleration', 'year', 'origin'],
      dtype='object')
```

Basics of Selecting Rows and Columns

- We can access rows of the data frame by *name* using the *loc[]* method of *Auto*:

```
rows = ['amc rebel sst', 'ford torino']
Auto_re.loc[rows]
```

```
                mpg   cylinders   displacement  horsepower   ...
         name
amc rebel sst  16.0           8          304.0       150.0   ...
  ford torino  17.0           8          302.0       140.0   ...
```

- As an alternative to using the index name, we could retrieve rows and columns of *Auto* using the *iloc[]* method:

```
Auto_re.iloc[[3,4]]
```

```
Auto_re.iloc[:,[0,2,3]]
```

```
Auto_re.iloc[[3,4],[0,2,3]]
```

For Loops

```
total = 0
for value in [3,2,19]:
    total += value
print('Total is: {0}'.format(total))
```

```
Total is: 24
```

- The indented code beneath the line with the *for* statement is run for each value in the sequence specified in the *for* statement. The loop ends either when the cell ends or when code is indented at the same level as the original *for* statement. Loops can be nested by additional indentation.

```
total = 0
for value in [2,3,19]:
    for weight in [3, 2, 1]:
        total += value * weight
print('Total is: {0}'.format(total))
```

For Loops

- To compute the average value of a random variable that takes on possible values 2, 3 or 19 with probability 0.2, 0.3, 0.5 respectively we would compute the weighted sum. Tasks such as this can often be accomplished using the *zip()* function that loops over a sequence of tuples.

```
total = 0
for value, weight in zip([2,3,19],
                         [0.2,0.3,0.5]):
    total += weight * value
print('Weighted average is: {0}'.format(total))
```

```
Weighted average is: 10.8
```

String Formatting

- Inserting the value of something into a string is a common task.
- For example, we may want to loop over the columns of a data frame and print the percent missing in each column.
- Let's create a data frame *D* with columns in which 20% of the entries are missing, i.e., set to *np.nan*. We'll create the values in *D* from a normal distribution with mean 0 and variance 1 using *rng.standard_normal*() and then overwrite some random entries using *rng.choice*().

String Formatting

```
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'])
D[:3]
```

	food	bar	pickle	snack	popcorn
0	0.345584	0.821618	0.330437	-1.303157	NaN
1	NaN	-0.536953	0.581118	0.364572	0.294132
2	NaN	0.546713	NaN	-0.162910	-0.482119

```
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
```

Additional Graphical and Numerical Summaries

- We can use the *ax.plot()* or *ax.scatter()* functions to display the quantitative variables. However, Python does not know to look in the *Auto* data set for those variables.

```
fig, ax = subplots(figsize=(8, 8))
ax.plot(horsepower, mpg, 'o');
```

```
NameError: name 'horsepower' is not defined
```

- We can address this by accessing the columns directly:

```
fig, ax = subplots(figsize=(8, 8))
ax.plot(Auto['horsepower'], Auto['mpg'], 'o');
```

Additional Graphical and Numerical Summaries

- Alternatively, we can use the *plot()* method with the call *Auto.plot().* Using this method, the variables can be accessed by name.

```
ax = Auto.plot.scatter('horsepower', 'mpg');
ax.set_title('Horsepower vs. MPG')
```

- The columns of a data frame can be accessed as attributes: try typing in *Auto.horsepower.*

Additional Graphical and Numerical Summaries

- *Auto.cylinders.dtype* reveals that it is being treated as a quantitative variable. However, since there is only a small number of possible values for this variable, we may wish to treat it as qualitative.
- Below, we replace the cylinders column with a categorical version of *Auto.cylinders*.

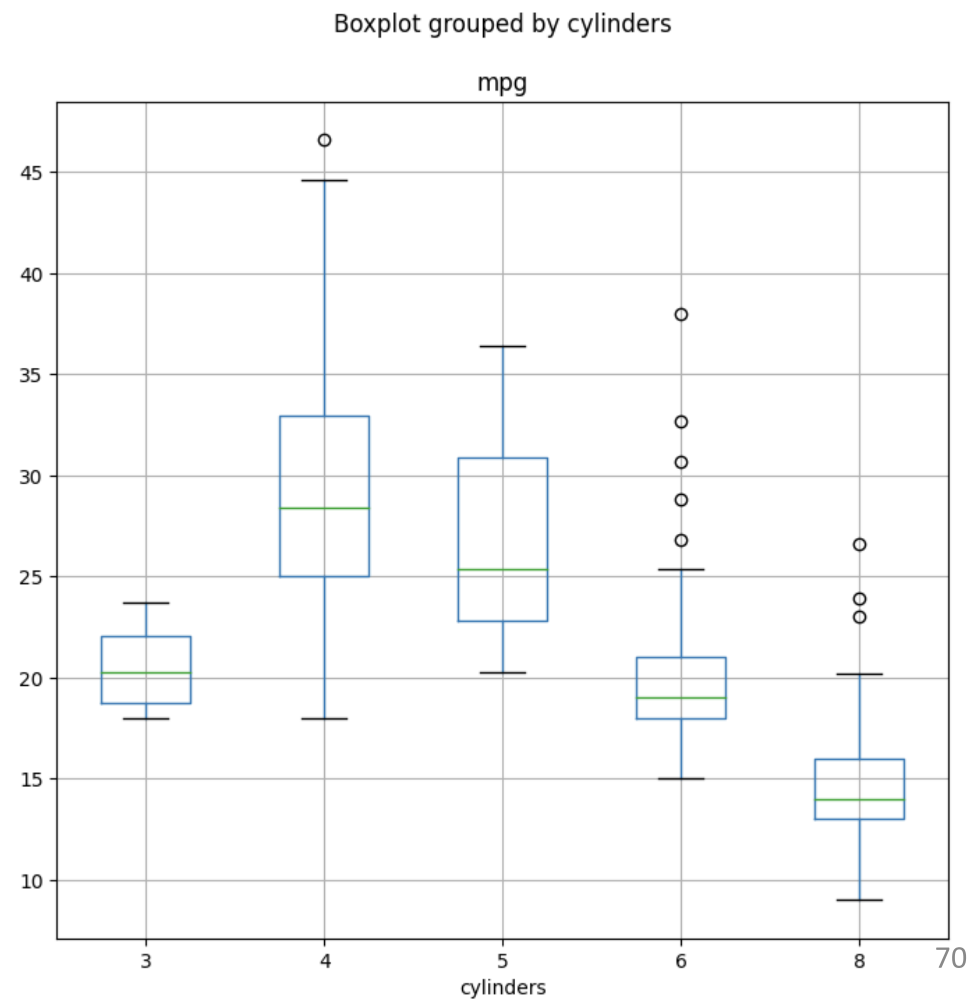
```
Auto.cylinders = pd.Series(Auto.cylinders, dtype='category')
Auto.cylinders.dtype
```

```
CategoricalDtype(categories=[3, 4, 5, 6, 8], ordered=False)
```

Additional Graphical and Numerical Summaries

- Now that cylinders is qualitative, we can display it using the *boxplot()* method.

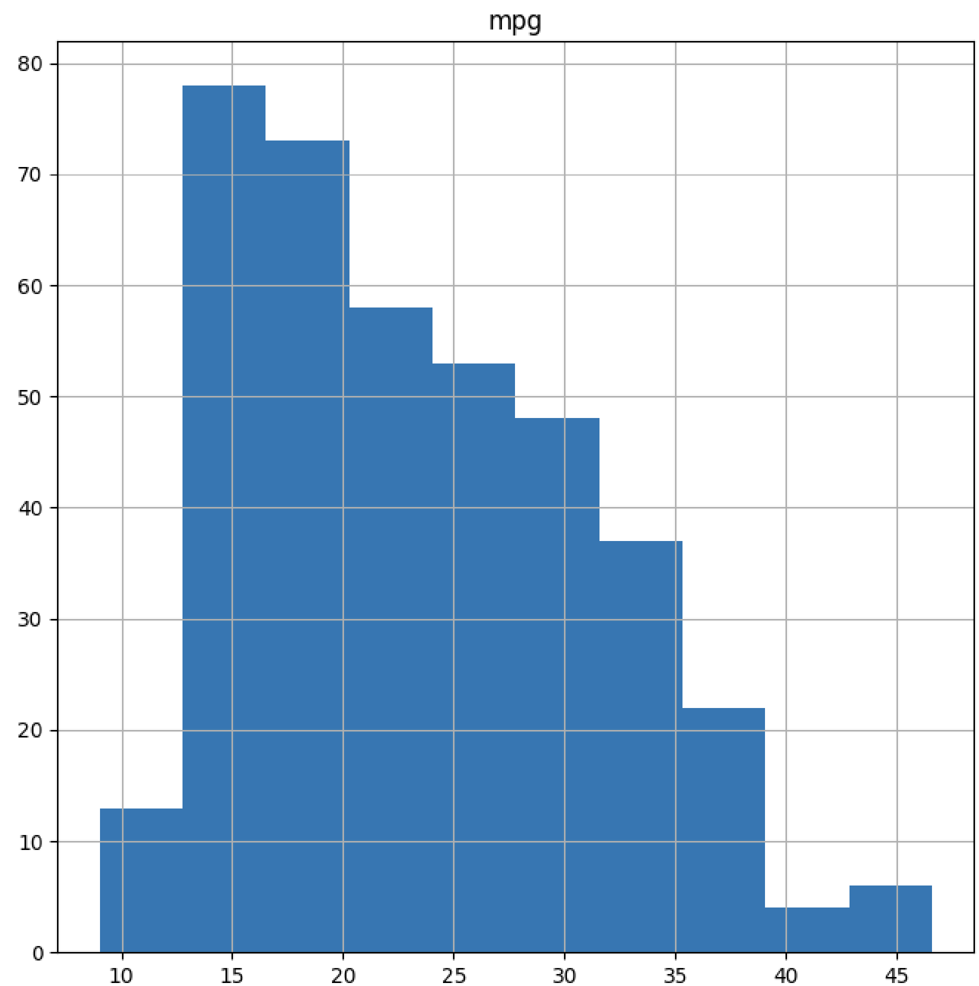
```
fig, ax = subplots(figsize=(8, 8))
Auto.boxplot('mpg', by='cylinders', ax=ax);
```



Additional Graphical and Numerical Summaries

- The *hist()* method can be used to plot a histogram.

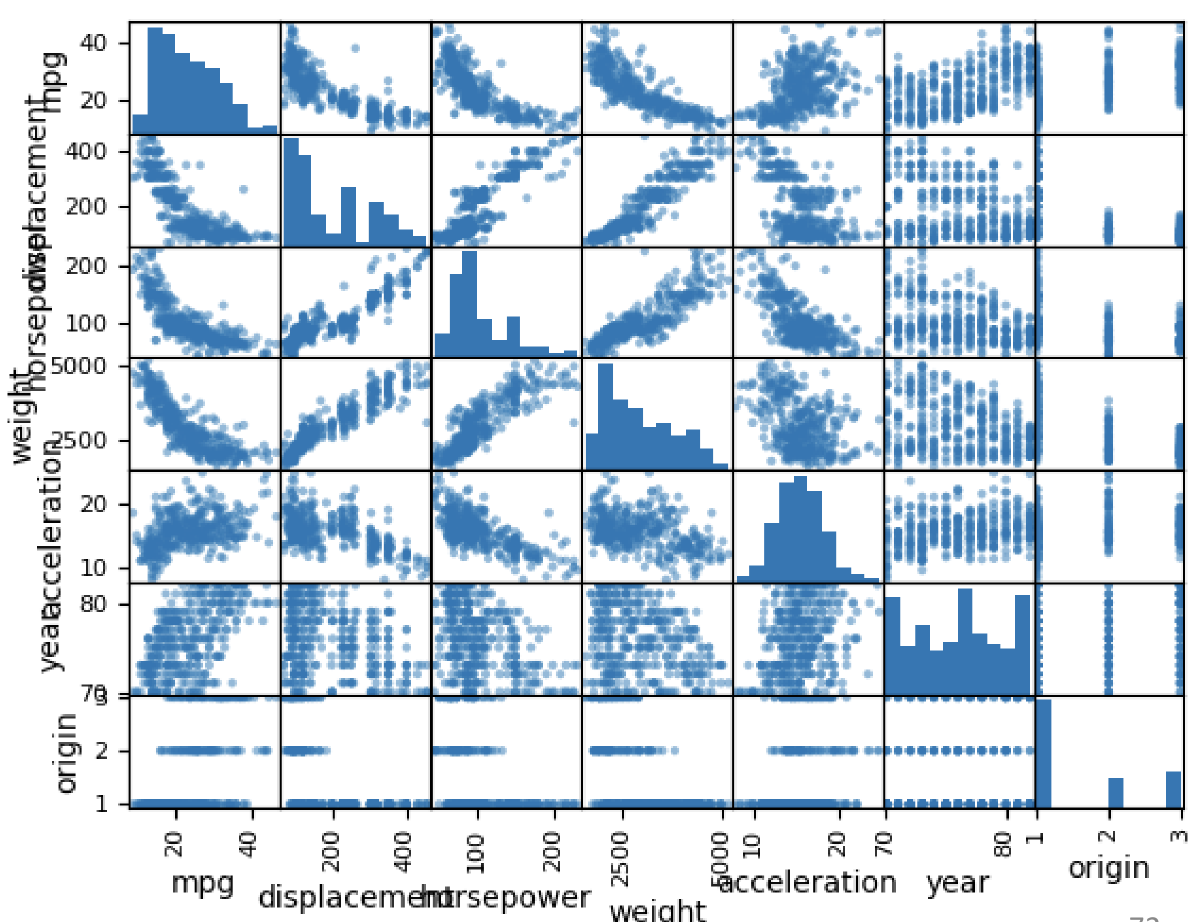
```
fig, ax = subplots(figsize=(8, 8))
Auto.hist('mpg', ax=ax);
```



Additional Graphical and Numerical Summaries

- We can use the *pd.plotting.scatter_matrix()* function to create a scatterplot matrix to visualize all of the pairwise relationships between the columns in a data frame.

```
pd.plotting.scatter_matrix(Auto);
```



Additional Graphical and Numerical Summaries

- The describe() method produces a numerical summary of each column in a data frame.

```
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