Introduction to basic tools (Numpy, Pandas, MatplotLib, Seaborn)

In this tutorial we will look into the Numpy library: [http://www.numpy.org/ (http://www.numpy.org/)](http://www.numpy.org/)

Numpy is a very important library for numerical computations and matrix manipulation. It has a lot of the functionality of Matlab, and some of the functionality of Pandas

In [1]:

```
import numpy as np
import pandas as pd
```

Why Numpy?

Example: generate two large vectors (lists, arrays, etc) and add them up.

In [2]:

```
import time

def trad_version():
    t1 = time.time()
    X = range(10000000)
    Y = range(10000000)
    Z = [x+y for x,y in zip(X,Y)] #loop over tuples like (x1, y1), (x2, y2) ...
    return time.time() - t1

def naive_numpy_version():
    t1 = time.time()
    X = np.arange(10000000)
    Y = np.arange(10000000)
    Z = np.zeros(10000000)
    for i in range(10000000):
        Z[i] = X[i]+Y[i]
    return time.time() - t1

def numpy_version():
    t1 = time.time()
    X = np.arange(10000000)
    Y = np.arange(10000000)
    Z = X + Y
    return time.time() - t1


traditional_time = trad_version()
naive_numpy_time = naive_numpy_version()
numpy_time = numpy_version()
print ("Traditional time = "+ str(traditional_time))
print ("Naive numpy time       = "+ str(naive_numpy_time))
print ("Numpy time       = "+ str(numpy_time))
```

```
Traditional time = 1.4511222839355469
Naive numpy time       = 4.2815563678741455
Numpy time       = 0.030846357345581055
```

Arrays

Creating Arrays

In Numpy, data is organized into arrays. There are many different ways to create a numpy array.

For the following we will use the random library of Numpy: [http://docs.scipy.org/doc/numpy-1.10.0/reference/routines.random.html (http://docs.scipy.org/doc/numpy-1.10.0/reference/routines.random.html)](http://docs.scipy.org/doc/numpy-1.10.0/reference/routines.random.html)

Creating arrays from lists

In [3]:

```
#1-dimensional arrays
x = np.array([2,5,18,14,4])
print ("\n Deterministic 1-dimensional array \n")
print (x)

#2-dimensional arrays
x = np.array([[2,5,18,14,4], [12,15,1,2,8]])
print ("\n Deterministic 2-dimensional array \n")
print (x)
```

```
 Deterministic 1-dimensional array

[ 2  5 18 14  4]

 Deterministic 2-dimensional array

[[ 2  5 18 14  4]
 [12 15  1  2  8]]
```

We can also create Numpy arrays from Pandas DataFrames

In [4]:

```
d = {'A':[1., 2., 3., 4.],
     'B':[4., 3., 2., 1.]}
df = pd.DataFrame(d)
x = np.array(df)
print(x)
```

```
[[1. 4.]
 [2. 3.]
 [3. 2.]
 [4. 1.]]
```

Creating random arrays

In [5]:

```
#1-dimensional arrays
x = np.random.rand(5)
print ("\n Random 1-dimensional array \n")
print (x)

#2-dimensional arrays

x = np.random.rand(5,5)
print ("\n Random 5x5 2-dimensional array \n")
print (x)

x = np.random.randint(10,size=(2,3))
print("\n Random 2x3 array with integers")
print(x)
```

```
 Random 1-dimensional array 

[0.42487286 0.41176468 0.59465439 0.89526189 0.39724261]

 Random 5x5 2-dimensional array 

[[0.51399166 0.8430621  0.76417979 0.54055998 0.247367  ]
 [0.13134887 0.25637572 0.28903523 0.47818357 0.49434342]
 [0.58024633 0.17615738 0.35838108 0.65282912 0.11935115]
 [0.75712299 0.41293248 0.13174633 0.13968171 0.07826283]
 [0.0479161  0.69505621 0.16493388 0.14320333 0.44227487]]

 Random 2x3 array with integers
[[1 8 2]
 [4 7 9]]
```

Special Arrays

In [7]:

```python
x = np.zeros((4,4))
print ("\n 4x4 array with zeros \n")
print(x)

x = np.ones((4,4))
print ("\n 4x4 array with ones \n")
print (x)

x = np.eye(4)
print ("\n Identity matrix of size 4\n")
print(x)

x = np.diag([1,2,3])
print ("\n Diagonal matrix\n")
print(x)
```

```
 4x4 array with zeros 

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

 4x4 array with ones 

[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]

 Identity matrix of size 4

[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]

 Diagonal matrix

[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

In [8]:

```python
A = np.random.randint(10,size=(2,3))
A
```

Out[8]:

```
array([[3, 6, 1],
       [3, 6, 1]])
```

In [9]:

```python
v = np.array([2,3])
D = np.diag(v) #create a diagonal matrix, with [2,3]
print(D@A) #matrix multiplication
```

```
[[ 6 12  2]
 [ 9 18  3]]
```

Operations on arrays.

These are very similar to what we did with Pandas

In [10]:

```
x = np.random.randint(10, size = (2,4))
print (x)
print('\n mean value of all elements')
print (np.mean(x))
print('\n vector of mean values for columns')
print (np.mean(x,0)) #0 signifies the dimension meaning columns
print('\n vector of mean values for rows')
print (np.mean(x,1)) #1 signifies the dimension meaning rows
```

```
[[8 8 0 8]
 [8 9 7 8]]

 mean value of all elements
7.0

 vector of mean values for columns
[8.  8.5 3.5 8. ]

 vector of mean values for rows
[6. 8.]
```

In [11]:
```python
print('\n standard deviation of all elements')
print (np.std(x))
print('\n vector of std values for rows')
print (np.std(x,1)) #1 signifies the dimension meaning rows
print('\n median value of all elements')
print (np.median(x))
print('\n vector of median values for rows')
print (np.median(x,1))
print('\n sum of all elements')
print (np.sum(x))
print('\n vector of column sums')
print (np.sum(x,0))
print('\n product of all elements')
print (np.prod(x))
print('\n vector of row products')
print (np.prod(x,1))
```

```
 standard deviation of all elements
2.692582403567252

 vector of std values for rows
[3.46410162 0.70710678]

 median value of all elements
8.0

 vector of median values for rows
[8.  8.]

 sum of all elements
56

 vector of column sums
[16 17  7 16]

 product of all elements
0

 vector of row products
[   0 4032]
```

Manipulating arrays

Accessing and Slicing

In [12]:

```
x = np.random.rand(4,3)
print(x)
print("\n element\n")
print(x[1,2])
print("\n row zero \n")
print(x[0,:])
print("\n column 2 \n")
print(x[:,2])
print("\n submatrix \n")
print(x[1:3,0:2])
print("\n entries > 0.5 \n")
print(x[x>0.5])
```

```
[[0.62405124 0.40513119 0.627672  ]
 [0.65656526 0.05848209 0.51303353]
 [0.64900435 0.5646028  0.25781207]
 [0.88894767 0.64988613 0.43416638]]

 element

0.513033527094123

 row zero

[0.62405124 0.40513119 0.627672  ]

 column 2

[0.627672   0.51303353 0.25781207 0.43416638]

 submatrix

[[0.65656526 0.05848209]
 [0.64900435 0.5646028 ]]

 entries > 0.5

[0.62405124 0.627672   0.65656526 0.51303353 0.64900435 0.5646028
 0.88894767 0.64988613]
```

Changing entries

In [13]:
```
x = np.random.rand(4,3)
print(x)

x[1,2] = -5 #change an entry
x[0:2,:] += 1 #change a set of rows
x[2:4,1:3] = 0.5 #change a block
print(x)

print('\n Set entries > 0.5 to zero')
x[x>0.5] = 0
print(x)
```

```
[[0.40275828 0.79754803 0.28586192]
 [0.56442874 0.05009653 0.95618195]
 [0.13050762 0.26012062 0.10066745]
 [0.30332523 0.87777386 0.51247311]]
[[ 1.40275828  1.79754803  1.28586192]
 [ 1.56442874  1.05009653 -4.        ]
 [ 0.13050762  0.5         0.5       ]
 [ 0.30332523  0.5         0.5       ]]

 Set entries > 0.5 to zero
[[ 0.          0.          0.        ]
 [ 0.          0.         -4.        ]
 [ 0.13050762  0.5         0.5       ]
 [ 0.30332523  0.5         0.5       ]]
```

In [14]:
```
print('\n Diagonal \n')
x = np.random.rand(4,4)
print(x)
print('\n Read Diagonal \n')
print(x.diagonal())
print('\n Fill Diagonal with 1s \n')
np.fill_diagonal(x,1)
print(x)
print('\n Fill Diagonal with vector \n')
x[np.diag_indices_from(x)] = [1,2,3,4]
print(x)
```

```

 Diagonal

[[0.91067147 0.91647418 0.69379541 0.56329488]
 [0.68510886 0.50546766 0.97590963 0.89036623]
 [0.48548505 0.26001958 0.48234241 0.8280419 ]
 [0.74031693 0.34437138 0.63228524 0.8239999 ]]

 Read Diagonal

[0.91067147 0.50546766 0.48234241 0.8239999 ]

 Fill Diagonal with 1s

[[1.         0.91647418 0.69379541 0.56329488]
 [0.68510886 1.         0.97590963 0.89036623]
 [0.48548505 0.26001958 1.         0.8280419 ]
 [0.74031693 0.34437138 0.63228524 1.        ]]
```

Operations with Arrays

Multiplication and addition with scalar

In [15]:
```
x = np.random.rand(4,3)
print(x)

#multiplication and addition with scalar value
print("\n Matrix 2x+1 \n")
print(2*x+1)
```

```
[[0.75062586 0.57459431 0.7964259 ]
 [0.95027688 0.16033685 0.25221051]
 [0.44475711 0.08340624 0.98814788]
 [0.65135975 0.12626252 0.80915897]]

 Matrix 2x+1

[[2.50125172 2.14918863 2.5928518 ]
 [2.90055376 1.32067369 1.50442102]
 [1.88951422 1.16681248 2.97629576]
 [2.30271951 1.25252504 2.61831793]]
```

Vector-vector dot product

There are three ways to get the dot product of two vectors:

- Using the method .dot of an array
- Using the method dot of the numpy library
- Using the '@' operator

In [16]:
```
y = np.array([2,-1,3])
z = np.array([-1,2,2])
print('\n y:',y)
print('  z:',z)
print('\n vector-vector dot product')
print(y.dot(z))
print(np.dot(y,z))
print(y@z)
```

```
 y: [ 2 -1  3]
 z: [-1  2  2]

 vector-vector dot product
2
2
2
```

External product

The external product between two vectors x,y of size (n,1) and (m,1) results in a matrix M of size (n,m) with entries M(i,j) = x(i)*y(j)

In [17]:
```
print('\n y:',y)
print(' z:',z)
print('\n vector-vector external product')
print(np.outer(y,z))
```

```
 y: [ 2 -1  3]
 z: [-1  2  2]

 vector-vector external product
[[-2  4  4]
 [ 1 -2 -2]
 [-3  6  6]]
```

Element-wise operations

In [18]:
```
print('\n y:',y)
print(' z:',z)
print('\n element-wise addition')
print(y+z)
print('\n element-wise product')
print(y*z)
print('\n element-wise division')
print(y/z)
```

```
 y: [ 2 -1  3]
 z: [-1  2  2]

 element-wise addition
[1 1 5]

 element-wise product
[-2 -2  6]

 element-wise division
[-2.  -0.5  1.5]
```

Matrix-Vector multiplication

Again we can do the multiplication either using the dot method or the '@' operator

In [19]:
```
X = np.random.randint(10, size = (4,3))
print('Matrix X:\n',X)
y = np.array([1,0,0])
print("\n Matrix-vector right multiplication with",y,"\n") #treated as column vec
print(X.dot(y))
print(np.dot(X,y))
print(X@y)
y = np.array([1,0,1,0])
print("\n Matrix-vector left multiplication with",y,"\n") #treated as row vector
print(y.dot(X))
print(np.dot(y,X))
print(y@X)
```

```
Matrix X:
 [[5 2 7]
 [1 1 1]
 [1 7 0]
 [8 1 7]]

 Matrix-vector right multiplication with [1 0 0]

[5 1 1 8]
[5 1 1 8]
[5 1 1 8]

 Matrix-vector left multiplication with [1 0 1 0]

[6 9 7]
[6 9 7]
[6 9 7]
```

Matrix-Matrix multiplication

Same for the matrix-matrix operation

In [20]:
```python
Y = np.random.randint(10, size=(3,2))
print("\n Matrix-matrix multiplication\n")
print('Matrix X:\n',X)
print('Matrix Y:\n',Y)
print('Product:\n',X.dot(Y))
print('Product:\n',X@Y)
```

```
 Matrix-matrix multiplication

Matrix X:
 [[5 2 7]
 [1 1 1]
 [1 7 0]
 [8 1 7]]
Matrix Y:
 [[4 1]
 [4 7]
 [5 1]]
Product:
 [[63 26]
 [13  9]
 [32 50]
 [71 22]]
Product:
 [[63 26]
 [13  9]
 [32 50]
 [71 22]]
```

Matrix-Matrix element-wise operations

In [21]:

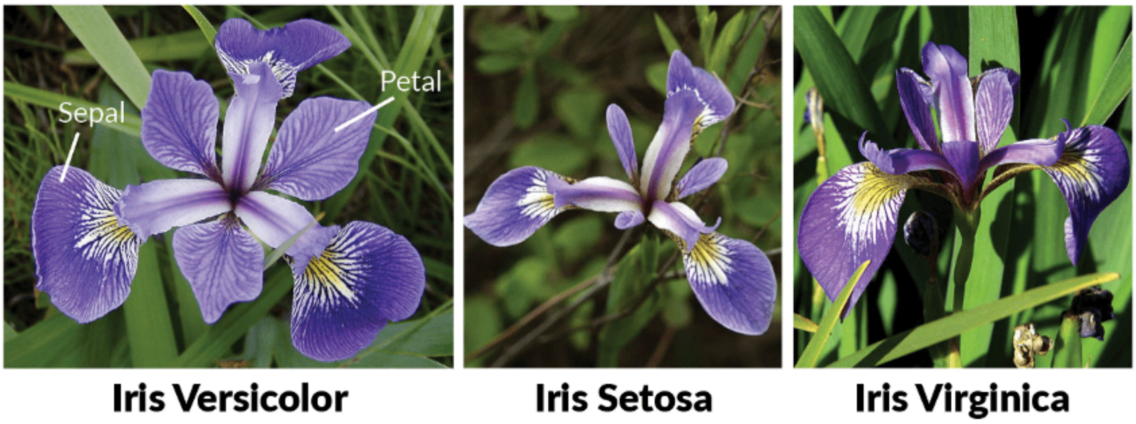
```python
Z = np.random.randint(10, size=(3,2))+1
print('Matrix Y:\n',Y)
print('Matrix Z:\n',Z)
print("\n Matrix-matrix element-wise addition\n")
print(Y+Z)
print("\n Matrix-matrix element-wise multiplication\n")
print(Y*Z)
print("\n Matrix-matrix element-wise division\n")
print(Y/Z)
```

```
Matrix Y:
 [[4 1]
 [4 7]
 [5 1]]
Matrix Z:
 [[7 9]
 [7 6]
 [6 4]]

 Matrix-matrix element-wise addition

[[11 10]
 [11 13]
 [11  5]]

 Matrix-matrix element-wise multiplication

[[28  9]
 [28 42]
 [30  4]]

 Matrix-matrix element-wise division

[[0.57142857 0.11111111]
 [0.57142857 1.16666667]
 [0.83333333 0.25      ]]
```

Example: Exploring the Iris Dataset

Iris

There are 3 types of Iris in the dataset.



In [22]:
```python
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from pandas.plotting import parallel_coordinates
#from sklearn import metrics
from sklearn import datasets
from sklearn.model_selection import train_test_split
```

Load the dataset from sklearn

In [23]:
```python
#iris = datasets.load_iris()
iris = datasets.load_iris()
```

In [24]:
```python
type(iris)
```

Out[24]: sklearn.utils.Bunch

Create a Pandas dataframe from the dataset.

In [25]:
```python
df = pd.DataFrame(data=iris.data, columns=iris.feature_names)
```

In [26]:
```python
df['species'] = pd.Categorical.from_codes(iris.target, iris.target_names)
```

Take a look at the first couple of rows.

In [27]:
```python
df.head(5)
```

Out[27]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Sepal: 萼片 Petal: 花瓣

First, let's look at a numerical summary of each attribute through describe:

In [28]:
```
df.describe()
```

Out[28]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333
std	0.828066	0.435866	1.765298	0.762238
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

We can also check the class distribution using groupby and size:

In [29]:
```
df.groupby('species').size()
```

Out[29]:
```
species
setosa        50
versicolor    50
virginica     50
dtype: int64
```

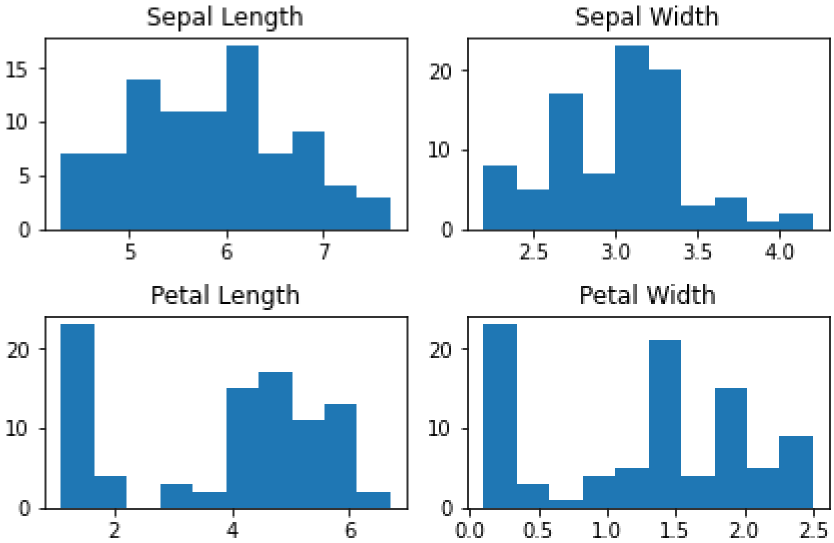
Now, we can split the dataset into a training set and a test set. In general, we should also have a validation set, which is used to evaluate the performance of each classifier and fine-tune the model parameters in order to determine the best model. The test set is mainly used for reporting purposes. However, due to the small size of this dataset, we can simplify this process by using the test set to serve the purpose of the validation set.

In [30]:
```
train, test = train_test_split(df, test_size = 0.4, random_state = 42)
```

Let's first create some univariate plots, through a histogram for each feature:

In [31]:

```
n_bins = 10
fig, axs = plt.subplots(2, 2)
axs[0,0].hist(train['sepal length (cm)'], bins = n_bins);
axs[0,0].set_title('Sepal Length');
axs[0,1].hist(train['sepal width (cm)'], bins = n_bins);
axs[0,1].set_title('Sepal Width');
axs[1,0].hist(train['petal length (cm)'], bins = n_bins);
axs[1,0].set_title('Petal Length');
axs[1,1].hist(train['petal width (cm)'], bins = n_bins);
axs[1,1].set_title('Petal Width');
# add some spacing between subplots
fig.tight_layout(pad=1.0);
```

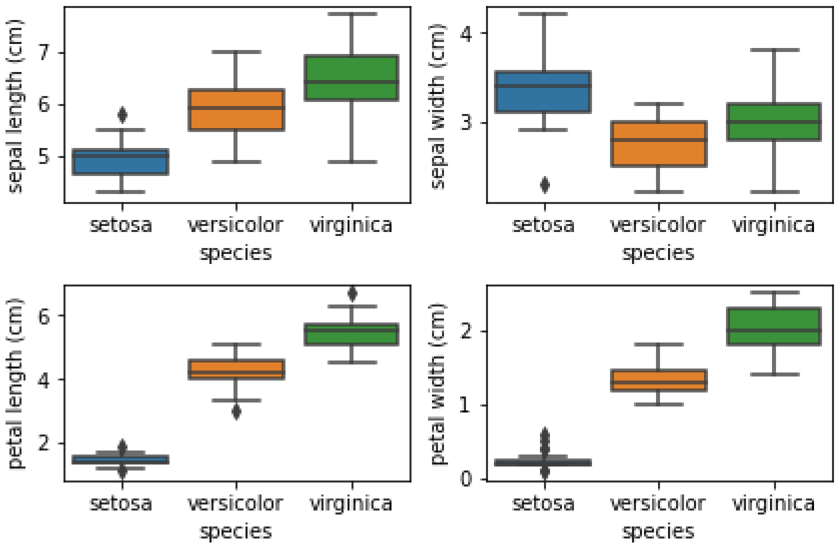


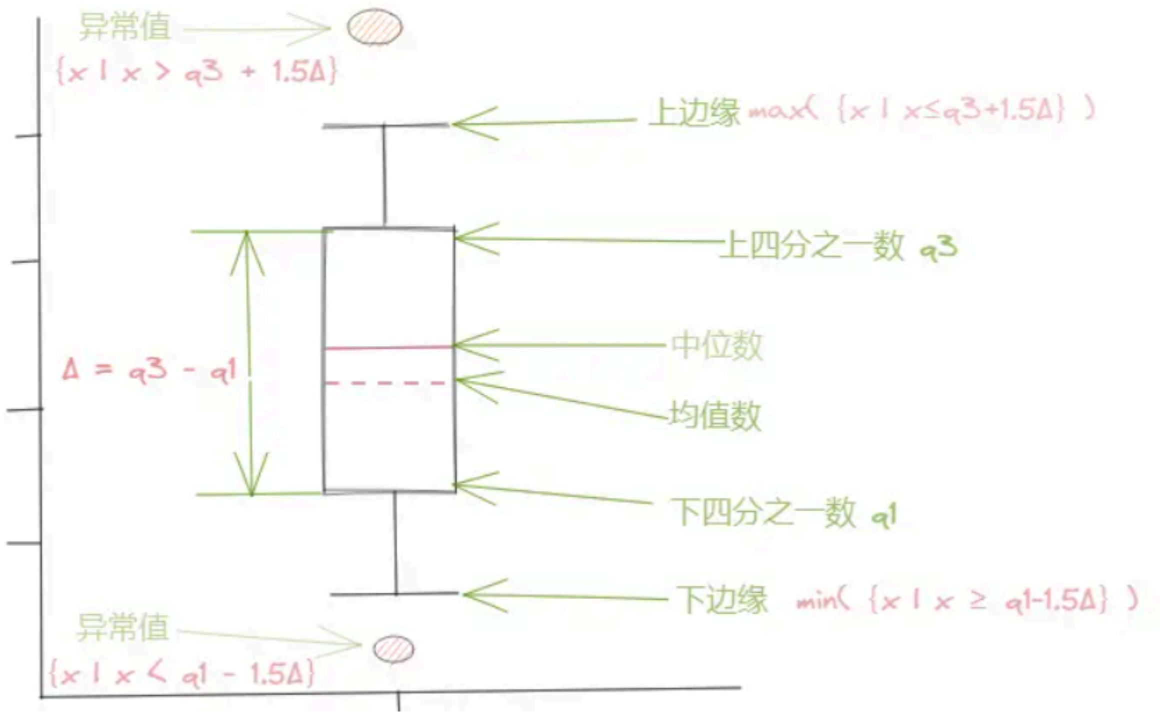
Note that for both **petal_length** and **petal_width**, there seems to be a group of data points that have smaller values than the others, suggesting that there might be different groups in this data.

Next, let's try some side-by-side box plots:

In [32]:

```
fig, axs = plt.subplots(2, 2)
#fn = ["sepal_length", "sepal_width", "petal_length", "petal_width"]
cn = ['setosa', 'versicolor', 'virginica']
sns.boxplot(x = 'species', y = 'sepal length (cm)', data = train, order = cn, ax
sns.boxplot(x = 'species', y = 'sepal width (cm)', data = train, order = cn, ax =
sns.boxplot(x = 'species', y = 'petal length (cm)', data = train, order = cn, ax
sns.boxplot(x = 'species', y = 'petal width (cm)', data = train,  order = cn, ax
# add some spacing between subplots
fig.tight_layout(pad=1.0)
```





https://zh.wikipedia.org/wiki/%E7%AE%B1%E5%BD%A2%E5%9C%96
(https://zh.wikipedia.org/wiki/%E7%AE%B1%E5%BD%A2%E5%9C%96)

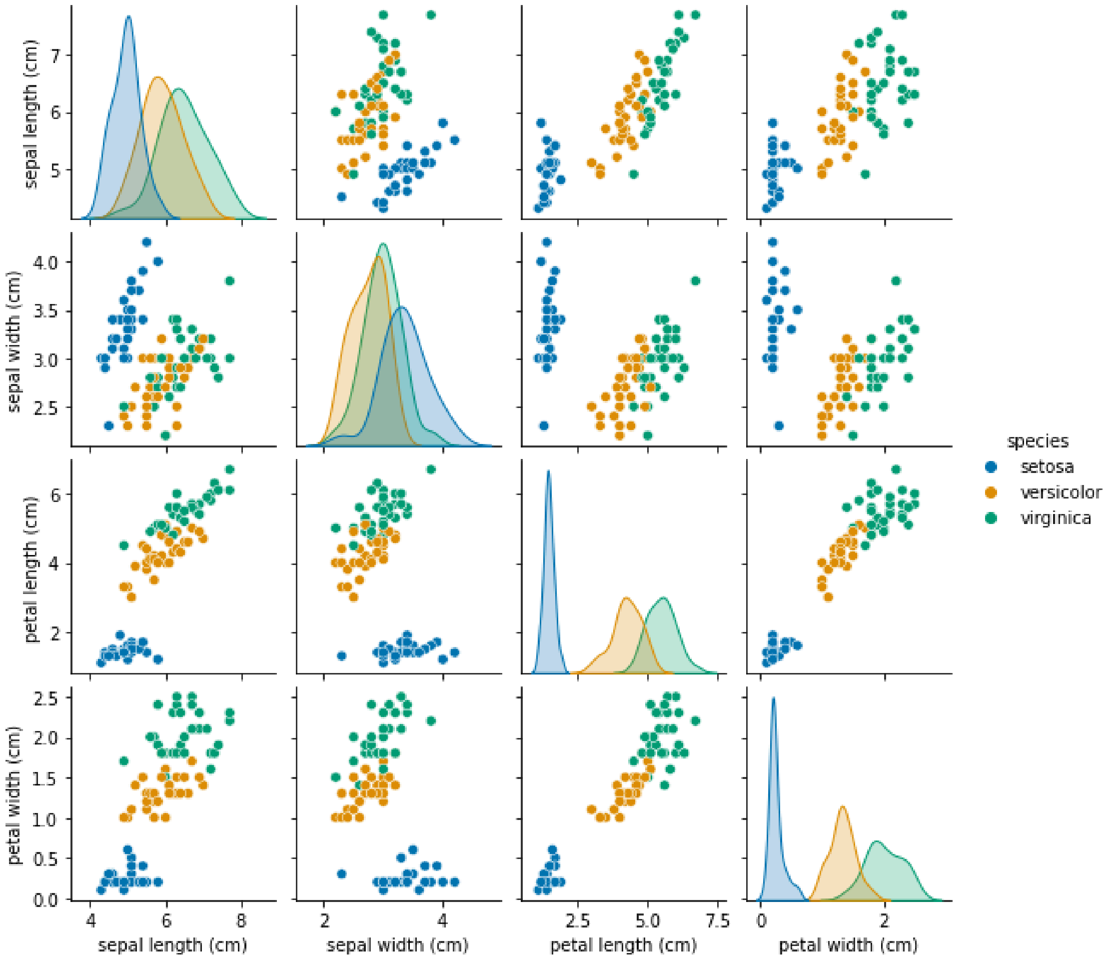
Q1，第1四分位数，即25百分位数；Q3，第3四分位数，即75百分位数。

四分位间距（interquartile range，简称IQR）=Q3-Q1;当有数值与第1四分位数与第3四分位数的范围差距1.5×IQR以上时，该值为离群值（outlier）

Now we can make scatterplots of all-paired attributes by using seaborn’s pairplot function:

```
In [33]: sns.pairplot(train, hue="species", height = 2, palette = 'colorblind')
```

Out[33]: <seaborn.axisgrid.PairGrid at 0x224355ef910>



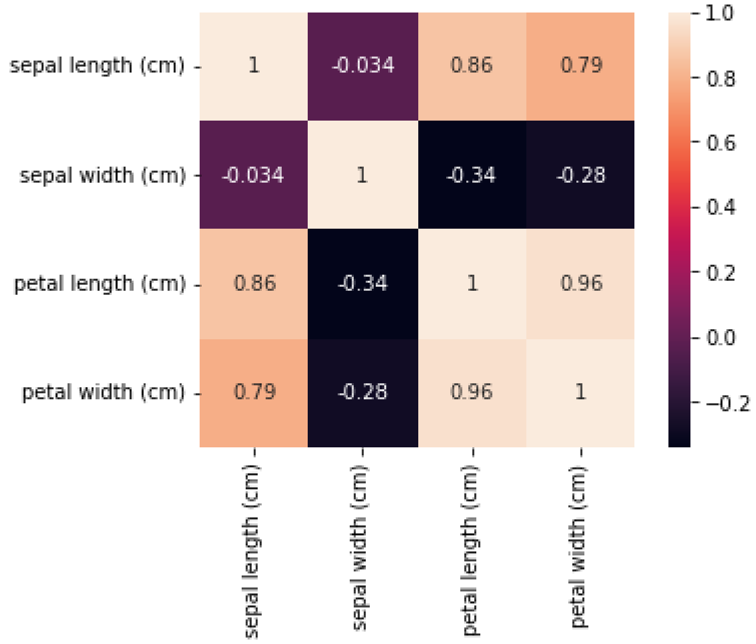
Note that **some variables seem to be highly correlated, e.g., petal_length and petal_width**.

In addition, the **petal measurements** separate the different species better than the sepal ones.

Next, let's make a **correlation matrix** to quantitatively examine the relationship between variables:

In [34]:
```python
corrmat = train.corr()
sns.heatmap(corrmat, annot = True, square = True)
```

Out[34]: <AxesSubplot:>



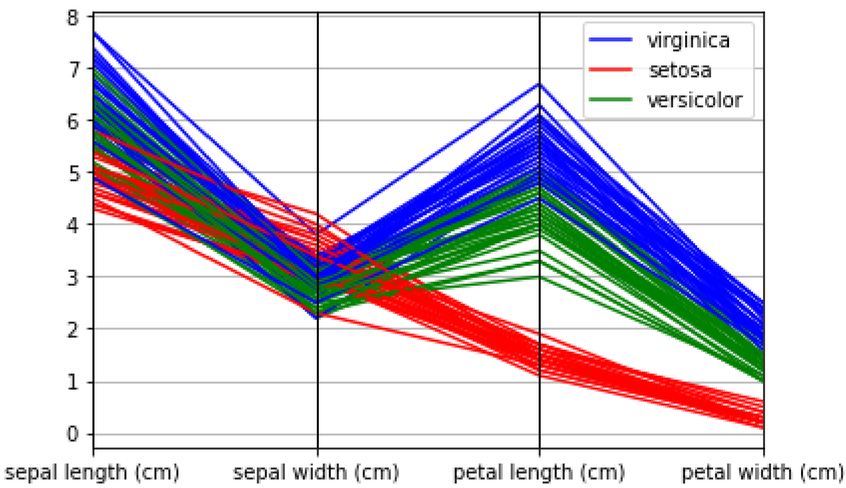
The main takeaway is that the **petal measurements have highly positive correlation** (0.96), while the sepal ones are uncorrelated (-0.034).

Note that the **petal features** also have relatively high **correlation with sepal_length (upper-right corner 0.86 and 0.79), but not with sepal_width (-0.34 and -0.28)**.

Another cool visualization tool is **parallel coordinate** plot, which represents each sample as a line.

In [35]:
```python
parallel_coordinates(train, "species", color = ['blue', 'red', 'green'])
```

Out[35]: <AxesSubplot:>



As we have seen before, **petal measurements can separate species better** than the sepal ones.