Chapter 2 Numeric and Data Tools

Steve Elston

March 27, 2020

Introduction to NumPy and Pandas

Data manipulation and numerical linear algebra are core skills in for anyone performing computational statistics. Not only that, but this background will be essential for you as your study of the rest of this book. Thus, this topic is a natural starting point.

In this chapter we will first explore numerical linear algebra using the Python NumPy package. This discussion will review some basics of linear algebra. The focus, however, is on the use of the Numpy package for numerical linear algebra. There are many excellent texts which treat linear algebra in depth, including Strang (2016) and Lay, Lay, and McDonald (2015).

Second, this chapter introduces the basics of data manipulation with the Pandas data frame package and NumPy. Even though data manipulation is not the subject of this book, we will be using Pandas throughout this book. Wes McKinney (2017) presents a broad overview of data manipulation and management using Pandas and NumPy.

Linear Algebra and NumPy

In this section we will review basic computational linear algebra. An understanding of computational linear algebra is essential to practicing computational statistics. Our focus will be numerical computing with the NumPy package (Oliphant 1970). An in depth discussion of the theory of computational linear algebra can be found in Golub and Loan (2012).

In this book we only review the basics of linear algebra with NumPy. You can find more details in the NumPy tutorial.

> **Programming Note:** NumPy is widely used for compuational statistics. NumPy computations are generally optimized. However, NumPy does not take full advantage of today's massively parallel cluster computing. Consequently, platforms like Spark (Zaharia et al. 2010), TensorFlow (Abadi et al. 2015), and Torch (Paszke et al. 2017) take full advantage of today's massive scale custer computing environments. We do not address use of these platforms in this book. However, the principles of numerical linear algebra are the same.

Arrays

Linear algebra is the algebra of **arrays**. An array is a regular arrangement of numbers. An array can have any number of dimensions, at least in principle.

A single numeric value is a **scaler**. A scalar has dimensional of 0.

A **vector** is a one dimensional array. As an example a vector of length n, and the ith value or element expressed as x_i, is written, $\mathbf{x} = [x_1, x_2, \ldots, x_n]$. We can also express a **column** vector:

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

A **matrix** is a 2 dimensional array. As an example, consider a matrix with dimension n rows by m columns, $\mathbf{A}$. An element of this matrix for the ith row and jth column can be written, $x_{i,j}$. In general, this matrix can be expressed:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,m} \\ a_{2,1} & a_{2,2} & \ldots & a_{2,m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \ldots & a_{n,m} \end{bmatrix}$$

As has already been mentioned, arrays can have any number of dimensions. We will focus on vectors and matrices, as these are used most often in computational statistics.

Element-Wise Operations

We will start with simple element-wise arithmetic operations which can be performed on arrays. These operations include addition, subtraction, and multiplication.

Element-wise operations require both arrays to be **conformable**. By conformable arrays we mean arrays which have dimensions that conform to the shape required for the operation being performed. For element-wise operations, conformable arrays must have the exactly the same dimensions so that each element in each array has a corresponding element in the other array. There is an exception, when one array is a scalar, therefore having dimension 0.

As a first step we will start an example by creating two vectors of length 3. In the code below, the NumPy array function is used to create these vectors. The vector is printed along with its type.

```
import pandas as pd
import numpy as np
import numpy.random as nr
import numpy.linalg as npla
import io
import requests
from math import sqrt, acos
import seaborn as sns
import matplotlib as plt
#%matplotlib inline

y = np.array([2]*3)
print('Array y = {}, with type {}'.format(y,type(y)))
```

```
## Array y = [2 2 2], with type <class 'numpy.ndarray'>
```

```
x = np.arange(1, 4)
print('Array x = {} with type {}'.format(x,type(x)))
```

```
## Array x = [1 2 3] with type <class 'numpy.ndarray'>
```

We can perform element-wise operations on these array with a scalar. Some examples are shown below. Notice that the usual Python arithmetic operators are used for element-wise operations.

```
a_scalar = 1.0
print(a_scalar + y)
```

```
## [3. 3. 3.]
```

```
print(a_scalar - x)
```

```
## [ 0. -1. -2.]
```

The code below applies some element-wise operations to the arrays.

```
print(y + x)
```

```
## [3 4 5]
```

```
print(y - x)
```

```
## [ 1  0 -1]
```

```
print(y * x)
```

```
## [2 4 6]
```

We can also perform element-wise operations on matrices. The code below creates a 4x3 (four rows x three columns) array all with the same value, 2.0, using the NumPy full function and prints the result.

```
A = np.full((4,3), 2.0)
print('A = \n{}'.format(A))
```

```
## A =
## [[2. 2. 2.]
##  [2. 2. 2.]
##  [2. 2. 2.]
##  [2. 2. 2.]]
```

Notice how this array is displayed. The rows are shown in order from the first row to the last. The values of each column for each row are shown within the row vectors. This is an important property of NumPy arrays, known as **row major order**. Other programming languages may use the alternative column major order.

The code below creates another 4x3 array containing a sequence of numbers from 1 to 12, with the following steps: 1. The NumPy arange function creates a 1-dimensional 1x12 array (vector) of the values. 2. The shape attribute is changed to 4x3 using the reshape method.
3. The `shape` attribute is then accessed and printed. Notice there is a difference between changing the shape attribute with the `reshape` function and accessing the shape attribute of the array.

```
B = np.arange(1,13).reshape((4,3))
print('Matrix B with shape {} \n{}'.format(B.shape, B))
```

```
## Matrix B with shape (4, 3)
## [[ 1  2  3]
##  [ 4  5  6]
##  [ 7  8  9]
##  [10 11 12]]
```

We can add a scalar to the matrix as shown in the code below.

```
print('1.0 + A = \n{}'.format(a_scalar + A))
```

```
## 1.0 + A =
## [[3. 3. 3.]
##  [3. 3. 3.]
##  [3. 3. 3.]
##  [3. 3. 3.]]
```

And, we can add the two conformable arrays as shown in the code below.

```
print('A + B = \n{}'.format(A + B))
```

```
## A + B =
## [[ 3.  4.  5.]
##  [ 6.  7.  8.]
##  [ 9. 10. 11.]
##  [12. 13. 14.]]
```

> **Exercise 2-1:** In the foregoing code example two conformal matrices are summed element-wise. Now, you will create conformable matrix with all the elements with a value of 2.0. Then, multiply this matrix by the matrix B. Ensure the result is correct?

The dot product, a fundamental array operation

After the foregoing introduction to the element-wise operations we move on to other array operations. We will start with perhaps the most fundamental multi-element operations, the **inner product** also known as the **dot product** or **scalar project**. The dot product between two vectors is defined:

$$dot\ product = \Sigma_i^n a_i \cdot b_i$$

A useful property is that the **Euclidian norm** (length), or **L2 norm**, of a vector is the square root of the dot product with itself. This can be expressed:

$$\| a \| = length\ of\ vector\ a = \sqrt{a \cdot a}$$

But, how can you interpret the dot product? The dot product is the **projection** of one vector on another. This concept can be expressed mathematically:

$$a \cdot b = \| a \| \| b \| cos(\theta)$$

Rearranging terms we can find the **cosine distance** between two vectors as:

$$cos(\theta) = \frac{a \cdot b}{\| a \| \| b \|}$$

Notice that the dot product of orthogonal (perpendicular) vectors is 0. In this case there is no projection of either vector on the other. Whereas, for parallel vectors the dot product is at a maximum.

These concepts are illustrated in the figure below. The dot product is the projection of one vector on another. The angle between the two vectors, θ, determines the projection. You can see that if $\theta = \frac{\pi}{2}$ the projection will be 0. If $\theta = 0$ the projection is maximized.

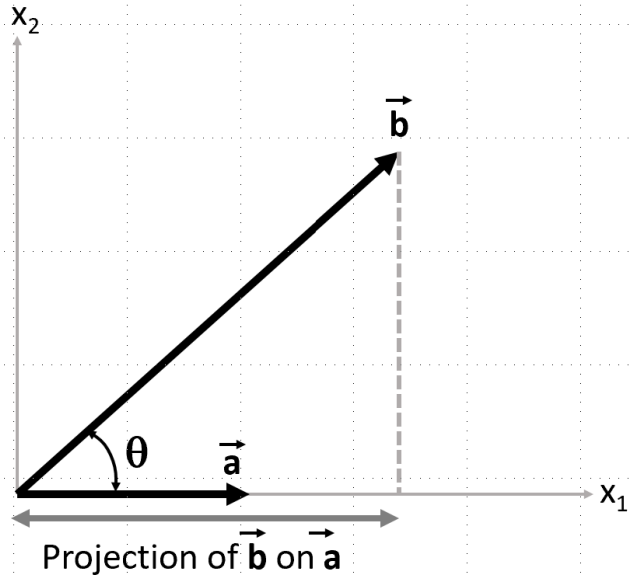

Figure 1: Slicing an 8 x 6 NumPy array

The code in the cell below computes the dot product of the two vectors, a and b using the NumPy dot function.

```
np.dot(x,y)
```

```
## 12
```

The result is the expected $12 = 2 + 4 + 6$.

As has been mentioned, the dot product is the projection of one vector on another. The code below demonstrates this point by taking the dot product of a vector with 3 times the same vector.

```
np.dot(y, np.transpose(np.dot(3,y)))
```

```
## 36
```

This dot project is 3 times the dot product of a with itself. Demonstrating that the projection scales with the length of the vectors.

Orthogonal vectors should have a dot product of 0. Intuitively, orthogonal vectors have no common directions. This useful property can be used to the test if vectors are orthogonal. This concept is demonstrated by the code below which takes the dot product of two orthogonal vectors.

```
w = np.array([1.0,1.0,0.0])
z = np.array([0.0,0.0,1.0])
np.dot(w, np.transpose(z))
```

```
## 0.0
```

The result is 0, as expected.

As already stated, dot or inner product can be used to compute the L2 norm of a vector. The code below uses the `norm` function from the `numpy.linalg` package to compute the norm of a vector.

```
npla.norm(y)
```

```
## 3.4641016151377544
```

The answer is the expected square root of 12.

Finally, the code in the cell below computes the cosign of the angle between the vectors a and b.

```
np.dot(x,y)/(npla.norm(x) * npla.norm(y))
```

```
## 0.9258200997725515
```

The matrix transpose

In many cases, we need the **transpose** of a matrix. The transpose is found by interchanging the row and column indices of a matrix.

Consider a 1-dimension vector of dimension $1xn$. The transpose of this array is a column vector of dimension $nx1$.

The elements of a 2-dimensional matrix are indexed first by a row index and then a column index, or $A_{i,j}$. The transpose is then, $A_{j,i}$, with the indices reversed. For example, using the notation #BˆT# for the transpose, we can write:

$$B_{ji} = B_{ij}^T$$

where,
B has dimensions n x m.
B^T has dimensions m x n.

To illustrate this idea, the code below uses the NumPy transpose function to find the transpose of a matrix.

```
np.transpose(B)
```

```
## array([[ 1,  4,  7, 10],
##        [ 2,  5,  8, 11],
##        [ 3,  6,  9, 12]])
```

The matrix product

Can we take other products of arrays? The answer is yes. We can multiply a vector times a matrix, or a matrix times a matrix. As you will see all of these operations are built with dot products. This is why the dot product is so important in computational statistics.

To start, consider how to compute the product of a matrix and a vector. You can think of this operation as a series of dot products between the rows of the matrix and the column vector. More specifically, an mxn matrix, $\mathbf{A}$, is multiplied by a vector, $\mathbf{x}$, of length n by taking the m dot products. The result is a vector of length m. This concept is illustrated in the relationship below.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{row\ 1} \cdot \mathbf{x} \\ \mathbf{A}_{row\ 2} \cdot \mathbf{x} \\ \vdots \\ \mathbf{A}_{row\ n} \cdot \mathbf{x} \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

We can summarize the above for the ith element of the resulting vector as:

$$y_i = \Sigma_j^m A_{ij} \cdot x_j$$

To demonstrate this concept, the code in the cell below uses the NumPy `dot` function to compute the product of a matrix and a vector.

```
np.dot(B, x)
```

```
## array([14, 32, 50, 68])
```

How do we extend the above computation to compute the product of two matrices? The answer is simple and intuitive. Each element of the result is the dot product between a row of the first matrix and a column of the second matrix. This is known as the row by column or **RC rule**. The product of an nxm matrix with an mxn matrix is an mxm dimensional array, computed by taking m^2 dot products. For example, the element Y_{ij} of the result matrix is computed as follows:

$$Y_{ij} = \Sigma_j^m A_{ij} \cdot B_{ji}$$

Notice that to be conformable the number of columns, m, of the first matrix must equal the number of rows of second matrix. And, that the number of rows, n of the first matrix must equal the number of columns of the second matrix.

The code in the cell below computes a matrix product. As both matrices are of dimension $4x3$, it is necessary to take the transpose of one or the other.

> **Programming Note:** For this example, we could use the Numpy `dot` function. However, here we use the `matmul` function. For 1 and 2 dimensional arrays, the results will be the same. But, `matmul` works with arrays of higher dimenstions, but not with scalars. The arrays must be conformable for either function to work.

```
np.matmul(np.transpose(A) , B)
```

```
## array([[44., 52., 60.],
##        [44., 52., 60.],
##        [44., 52., 60.]])
```

The above operation resulted in a $3x3$ matrix. Taking the transpose of the second matrix results in a conformable operation, but the result will have different dimensions.

```
np.matmul(A, np.transpose(B))
```

```
## array([[12., 30., 48., 66.],
##        [12., 30., 48., 66.],
##        [12., 30., 48., 66.],
##        [12., 30., 48., 66.]])
```

Now, the product is a $4x4$ matrix.

> **Programming Note:** In the foregoing you have seen how matrix products are built from dot products. If the dot product can be computed efficiently, then so to can matrix products be computed efficiently. Not only that, but each of the dot products required for these operations can be computed in parallel, as there is no dependency of one on another. These properties are at the basis of many of today's high performance computing environments used for machine learning and AI.

> **Exercise 2-2:** Perhaps, you are curious about what happens if you try to multiply non-conformable arrays. To find out, multiply the matrix A by the matrix B, but without the transpose.

The identity matrix and the inverse

Is it possible to divide one matrix by another? Yes in a certain way, but not in a direct manner. Instead, this operation is performed by finding the **inverse** of a matrix and multiplying by that inverse by the other matrix.

Before explaining the inverse, we need to examine a matrix with special properties, the **identity matrix**. An identity matrix is a square nxn matrix with 1s on the diagonal and 0s everywhere else. We can write the identity matrix as follows:

$$I = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

The code in the cell below creates a $3x3$ identity matrix using the NumPy eye function.

```
I3 = np.eye(3)
I3
```

```
## array([[1., 0., 0.],
##        [0., 1., 0.],
##        [0., 0., 1.]])
```

What will be the result if we multiply the identity by another square matrix? To understand the result, consider that multiplying by an identity matrix in linear algebra is the same as multiplying by a 1 in ordinary algebra.

The identity multiplied by any matrix gives that same matrix. If A is a square matrix (conformal) then:

$$A = I \cdot A = A \cdot I$$

Let's illustrate this concept with an example. First, the code in the cell below creates a $3x3$ numeric matrix.

```
C = np.array([[1,3,6],
              [2,2,1],
              [3,1,4]])
C
```

```
## array([[1, 3, 6],
##        [2, 2, 1],
##        [3, 1, 4]])
```

Now, we compute the product of the above matrix with the identity matrix

```
np.dot(C,I3)
```

```
## array([[1., 3., 6.],
##        [2., 2., 1.],
##        [3., 1., 4.]])
```

The result is the same as the original matrix.

What happens if the product is computed the other way? The code below performs this calculation.

```
np.dot(I3,C)
```

```
## array([[1., 3., 6.],
##        [2., 2., 1.],
##        [3., 1., 4.]])
```

Again, the result is the original matrix. This example demonstrates that multiplying a square matrix by an identity matrix of the same dimension is the same as multiplying by 1 in regular algebra.

Let's get back to the inverse of a matrix. In principle we can compute an inverse of a square matrix so that the following relationships hold:

$$\begin{aligned} A &= A \\ A &= AI \\ A^{-1}A &= I \end{aligned} \tag{1}$$

Here, A^{-1} is the inverse of A.

To illustrate this concept the code in the cell below computes the inverse of the matrix using the inv from the NumPy linalg package. The inverse is then multiplied by the original matrix.

```
inv_C = npla.inv(C)
np.dot(inv_C,C)
```

```
## array([[ 1.00000000e+00, -1.11022302e-16,  2.22044605e-16],
##        [ 1.66533454e-16,  1.00000000e+00,  2.22044605e-16],
##        [-1.38777878e-17,  6.93889390e-17,  1.00000000e+00]])
```

The result is an identity matrix, as it should be.

> **Exercise 2-3:** The identity matrix has a special property, that it is its own inverse. To demonstate this property, create a $4x4$ identity matrix and take its inverse. Is the result the same?

Slicing NumPy arrays

When using NumPy, you will very often need a subset of an array for some operations. Fortunately, NumPy arrays are easy to subset, an operation known as **slicing** or **indexing**. For a NumPy array, the slicing operator are the square brackets, `[]`. For a 2-dimensional matrix the syntax for this operator is:

$$slice = array[row\_start : row\_end,\ column\_start, column\_end]$$

Where, keeping in mind that NumPy indices are zero based,
row_start = the index of the first row in the slice.
row_start = the index of the first row not in the slice.
row_start = the index of the first column in the slice.
row_start = the index of the first column not in the slice.

The figure below shows an example of slicing a 2-dimensional array, or matrix. The matrix has dimension $8x6$, and a slice of dimension $4x3$ is created. Notice the syntax of the row and column slices.

There are some other ways to specify slices that use implied starts or ends, or a specific list of indices.

Required Operation	Syntax
All rows or all columns	`:`
first n rows or columns	`:n`
Rows or columns after the nth	`n:`
All but the last n rows or columns	`:-n`
Specific rows or columns	List of indices
Specific rows or columns	List of logicals

To illustrate these operations, let's try some examples. The code below creates a slice of columns 1 and 2 a NumPy matrix, keeping all rows:

```
B[:, 1:3]
```

```
## array([[ 2,  3],
##        [ 5,  6],
```

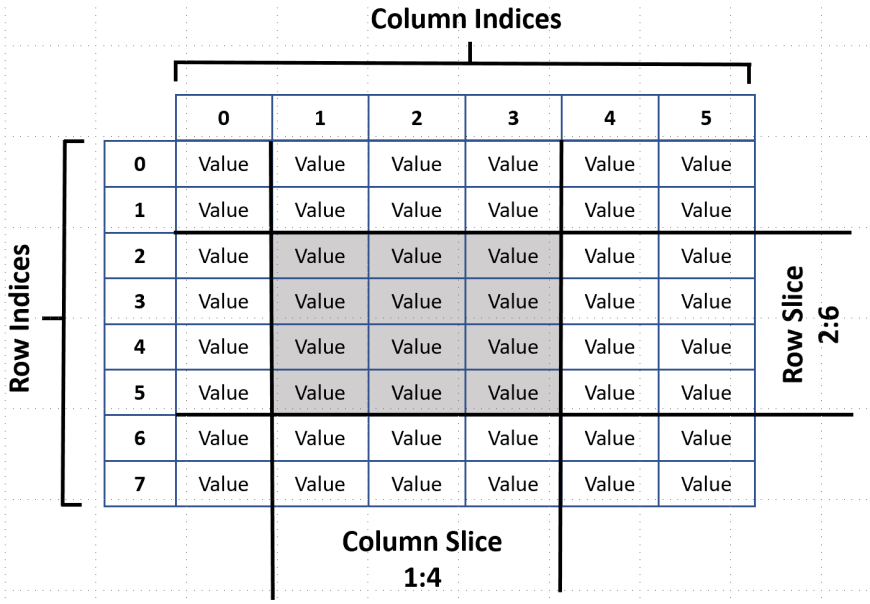

Figure 2: Slicing an 8 x 6 NumPy array

```
##        [ 8,  9],
##        [11, 12]])
```

The code below creates a slice of the first 2 rows of a NumPy matrix, while keeping all columns.

```
B[:2, :]
```

```
## array([[1, 2, 3],
##        [4, 5, 6]])
```

The code below creates a slice with all but the last column.

```
B[:, :-1]
```

```
## array([[ 1,  2],
##        [ 4,  5],
##        [ 7,  8],
##        [10, 11]])
```

And, the code below creates a slice with rows 0 and 2. Don't be confused by the two sets of square brackets, `[]`. The outer square brackets are the slicing operator whereas, the inner square brackets are for the list of row indices.

```
B[[0,2], :]
```

```
## array([[1, 2, 3],
##        [7, 8, 9]])
```

The same result can be achieved with a list of logicals, as shown here.

```
B[[True,False,True,False], :]
```

```
## array([[1, 2, 3],
##        [7, 8, 9]])
```

The results are the same for both of the above cases.

> **Exercise 2-4:** In the foregoing discussion of the matrix inverse, we did not address an important and common problem. Simply put, matricies with colinear or parallel columns are said to be **singular**. The inverse of a singular matrix does not exist. For now, you will determine if the columns of a matrix are nearly colinear. In compuational linear algebra, nearly colinear columns will cause matrix inverse algorithms to fail. Dealing with this problem is a focus of Part 4 of this book. You will now complete the code below to do the following: a. First you will construct a nearly sigular $4x4$ matrix with sequential values 1 to 16 running row-wise. b. Next, using the indices createded by the Itertools `combinations` function you will slice the array column-wise, for every possible pair of columns. c. Compute the cosign of the angle between each of the column-wise slices. d. Print the indices and the cosign of the angle between them. What do these values of the cosign tell you about the columns? Are the columns nearly colinear?

Copies of NumPy arrays

NumPy operations follow the same assignment rules as other Python operators. However, these assignment rules create pitfalls for newcomers. A simple `=` only assigns a new name to the same array. To ensure the ssignment creates a copy, one must use the `copy` function. This idea is illustrated in the code below.

```
C = np.copy(B)
B = B + 1.0
print(B)
```

```
## [[ 2.  3.  4.]
##  [ 5.  6.  7.]
##  [ 8.  9. 10.]
##  [11. 12. 13.]]
```

```
print(C)
```

```
## [[ 1  2  3]
##  [ 4  5  6]
##  [ 7  8  9]
##  [10 11 12]]
```

These results are what we intended. Failure to use the `copy` function can result in unexpected and confusing errors.

Pandas, a Data Scientist's Friend

Now we will turn our attention to the Pandas package (McKinney 2010). Pandas has become a standard tool in data scientists' tool kit for data management and manipulation.

Pandas is an extension of NumPy's arrays. Pandas provides significant capabilities to mange and manipulate tabular data. The focus of this book is not on the data manipulation capabilities of Pandas, so only an overview is provided here. The tutorials on the Pandas.org website provide greater depth.

What is a data frame?

The basic Pandas data structure is the data frame. A data frame is a tabular data structure with a number of special attributes. The most important of these attributes are the row indices, column indices and data types of the columns. A schematic view of a Pandas data frame with these important attributes is illustrated in the figure below.

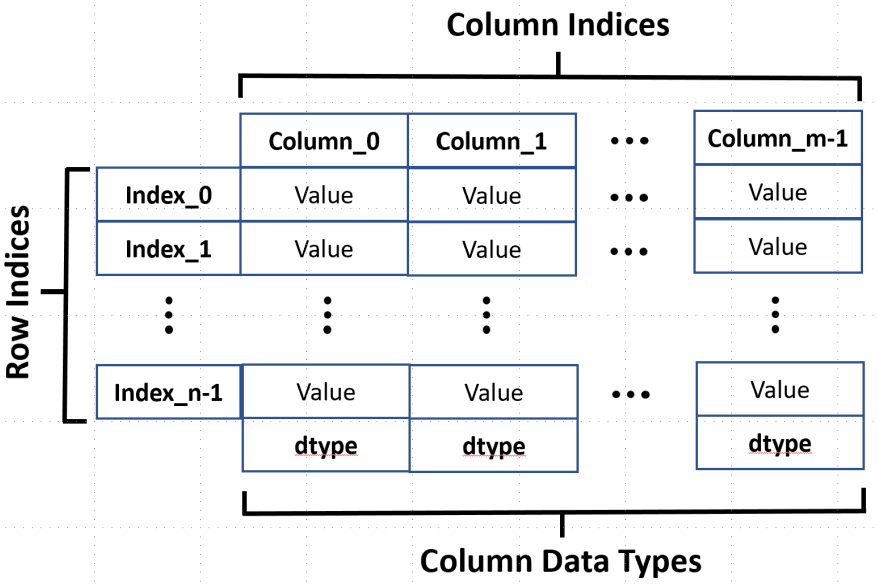

Figure 3: Sematic view of a Pandas data frame

As you can see from the above figure, the basic Pandas data frame is an nxm table. The data are arranged in m columns. Each column has an index, which can be a column name as a string, and a data type. Each column can have a different data type, but each column can only have one type. There is also a set of row n indices. Row indices are typically numeric, in the range $\{0, n-1\}$, but can be names in the form of strings.

An example will help illustrate these points. There are a number of ways to construct a Pandas data frame. Perhaps, the simplest is creating a data frame from an array, using the Pandas DataFrame method, as is done here.

```
a_dataframe = pd.DataFrame([[1, 3, 5],
                            [2, 4, 8],
                            [10, 20, 30],
                            [15, 25, 35]])
a_dataframe
```

```
##     0   1   2
## 0   1   3   5
## 1   2   4   8
## 2  10  20  30
## 3  15  25  35
```

You can see that the data frame has integer column and row indices at this point. We can assign new values to these attributes, as shown in the code below. The `index` attribute changes the row indices and the `columns` attribute changes the column names.

```
a_dataframe.index = ['first_row','second_row','third_row','forth_row']
a_dataframe.columns = ['first_col','second_col','third_col']
a_dataframe
```

```
##             first_col  second_col  third_col
## first_row           1           3          5
## second_row          2           4          8
## third_row          10          20         30
## forth_row          15          25         35
```

Let's add another column to this data frame. This can be done using the `loc` method. The `loc` method allows you to specify the rows as the first argument and the columns as the second argument. A colon, `:` indicates all rows or all columns. The code below adds a new column named 'animal.'

```
a_dataframe.loc[:,'animal'] = ['chicken','chicken','duck','duck']
a_dataframe
```

```
##             first_col  second_col  third_col   animal
## first_row           1           3          5  chicken
## second_row          2           4          8  chicken
## third_row          10          20         30     duck
## forth_row          15          25         35     duck
```

Let's have a look at the type attributes of the columns, the `dtypes`.

```
a_dataframe.dtypes
```

```
## first_col      int64
## second_col     int64
## third_col      int64
## animal        object
## dtype: object
```

The first three columns are integer type. With Pandas, the type of a string column is shown as `object`.

It is possible to change the data types of columns. In the code below we will change the type of the third column to floating point, using the `flaot64` type from NumPy. The string column is coerced to categorical.

```
a_dataframe.loc[:,'animal'] = a_dataframe.loc[:,'animal'].astype('category')
a_dataframe.loc[:,'third_col'] = a_dataframe.loc[:,'third_col'].astype(np.float64)
a_dataframe.dtypes
```

```
## first_col        int64
## second_col       int64
## third_col      float64
## animal        category
## dtype: object
```

Notice the new dtypes of the last two columns.

Slicing Pandas Data Frames

Just as we did for NumPy arrays, Pandas data frame can be sliced. There are a number of ways to define slices for Pandas data frame. Here we will focus on the computationally efficient `loc` and `iloc` methods. You have already seen an example of using `loc` above.

The `iloc` method slices using row and column indices, hence the 'i.' The rules and syntax are essentially identical to those used for slicing NumPy arrays, which we have already discussed.

The `loc` method performs slicing operations on Pandas data frames using column or row names, as well as logical selection. The syntax and rules are similar to NumPy array slicing. For example, the code below creates a slice with the columns up to and including `third_col`.

```
a_dataframe.loc[:,:'third_col']
```

```
##             first_col  second_col  third_col
## first_row           1           3        5.0
## second_row          2           4        8.0
## third_row          10          20       30.0
## forth_row          15          25       35.0
```

As another example, the code below creates a slice starting with `second_col` and ending with, and including, `third_col`.

```
a_dataframe.loc[:,'second_col':'third_col']
```

```
##             second_col  third_col
## first_row            3        5.0
## second_row           4        8.0
## third_row           20       30.0
## forth_row           25       35.0
```

The `loc` method can be used to find a slice of a data frame including the columns in a list, as shown here.

```
a_dataframe.loc[:,['second_col','animal']]
```

```
##             second_col   animal
## first_row            3  chicken
## second_row           4  chicken
## third_row           20     duck
## forth_row           25     duck
```

Finally, a logical operation can be used to for slicing, as shown below. In this case, the `loc` method is used twice. Once for the slicing the data frame and once for the logical operator.

```
a_dataframe.loc[a_dataframe.loc[:,'animal']=='duck',:'third_col']
```

```
##            first_col  second_col  third_col
## third_row         10          20       30.0
## forth_row         15          25       35.0
```

Copies of Pandas data frames

Just like NumPy operations, Pandas operations follow the same assignment rules as other Python operators. An `=` operator only assigns a new name to the same array. Further, the `loc` and `iloc` methods only create references, they do not return copies. This behavior optimizes memory use for scalability. If you want to ensure you have actually made a copy use the `copy` method.

This idea is illustrated in the code below which performs the following steps: 1. A slice is assigned to a new name. 2. A slice is created and copied and assigned to another name. 3. A A constant is added to the first slice using the Pandas `add` method. 4. The values for each name are printed.

> **Programming Note:** Notice there are two methods in the second line of code, This is an example of **chaining operators**. Chaining operators is a powerful approach for computational statistics. We will chain operators many times in the remainder of this book.

```
another_dataframe = a_dataframe.loc[a_dataframe.loc[:,'animal']=='duck',:'third_col']
copy_dataframe = a_dataframe.loc[a_dataframe.loc[:,'animal']=='duck',:'third_col'].copy()
another_dataframe = another_dataframe.add(1.0)
another_dataframe
```

```
##            first_col  second_col  third_col
## third_row       11.0        21.0       31.0
## forth_row       16.0        26.0       36.0
```

```
copy_dataframe
```

```
##            first_col  second_col  third_col
## third_row         10          20       30.0
## forth_row         15          25       35.0
```

Notice the difference in these results. A copy is not the same as a slice of the original data frame.

The Pandas series

We have been working with the 2-dimensional Pandas data frames. There is a special case for univariate data, the Pandas series. This distinction is important. Panads series have different attributes from data frames. This means different methods are applicable. Failure to abide by this distinction will often produce unexpected results or raise exceptions.

The code below creates a single column slice from a data frame. The result is Pandas series.

```
a_series = a_dataframe.loc[:,'second_col'].copy()
a_series
```

```
## first_row      3
## second_row     4
## third_row     20
## forth_row     25
## Name: second_col, dtype: int64
```

A noticeable difference for this series is that there is no column name attribute. Since a Pandas series has only one column, there is no need for this attribute.

Exercise 2-5: A limited number of mathematical operatons can be performed directly on numeric columns in a Pandas data frame. To use the richer capabilities of NumPy the columns of a of a Pandas data frame can be coerced to a Numpy array with the Pandas `as_matrix` method. To exercise this capability, do the following: a. Coerce the columns named `first_col` and `second_col` into a NumPy array. b. Take the element-wise square toot of the resulting array using the Numpy `sqrt` function. c. Assign the resulting Numpy array values back to the columns in the Data Frame. d. Print the data frame and verify that the result is correct.

Bibliography

Abadi, Martn, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, et al. 2015. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems." https://www.tensorflow.org/.

Golub, Gene H., and Charles F. Van Loan. 2012. *Matrix Computations.* Fourth Edition. Johns Hopkins University Press.

Lay, David C., Steven R. Lay, and Judy J. McDonald. 2015. *Linear Algebra: And Its Applications.* Fifth Edition. Pearson.

McKinney, Wes. 2010. "Data Structures for Statistical Computing in Python." http://conference.scipy.org/proceedings/scipy2010/mckinney.html.

Oliphant, Travis E. 1970. *A Guide to NumPy.* USA: Trelgol Publishing.

Paszke, Adam, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. "Automatic Differentiation in PyTorch." In *NIPS-w.*

Strang, Gilbert. 2016. *Introduction to Linear Algebra.* Fifth Edition. Wellesley-Cambridge Press.

Wes McKinney. 2017. *Python for Data Analysis: Data Wrangling with Pandas, NumPy and IPython.* Second Edition. O'Reilly Media.

Zaharia, Matei, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. "Spark: Cluster Computing with Working Sets." http://people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf.