chapter-1-2

February 2, 2022

Run in Google Colab

1 Python Library for Data Science: A Quick Glance

1.1 Introduction

In this lesson we will present the main scientific computation libraries in python used in data analysis. These are the most important libraries of a general nature for analyzing data in Python. In particular we will focus on:

- **Numpy** Numpy is a Python library with math functionalities. It allows us to work with multi-dimensional arrays, matrices, generate random numbers, linear algebra routines, and more.
- **Matplotlib/Seaborn** Matplotlib is a library that allows us to make basic plots, while Seaborn specializes in statistics visualization. The main difference is in the lines of code you need to write to create a plot. Seaborn is easier to learn, has default themes, and makes better-looking plots than Matplotlib by default.
- **Pandas** Pandas is a powerful tool that offers a variety of ways to manipulate and clean data. Pandas work with dataframes that structures data in a table similar to an Excel spreadsheet, but faster and with all the power of Python.

We will reserve a specific study in the course of the other lessons on two extremely important libraries for machine learning applications: scikit-learn and keras.

1.2 Basic NumPy

Numpy (it stands for Numerical Python) is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays. If you are already familiar with MATLAB, you might find this tutorial useful to get started with Numpy. NumPy helps to create arrays (multidimensional arrays), with the help of bindings of C++. Therefore, it is quite fast. There are in-built functions of NumPy as well. It is the fundamental package for scientific computing with Python.

Numpy arrays are collections of things, all of which must be the same type, that work similarly to lists (as we've described them so far). The most important are:

1. You can easily perform elementwise operations (and matrix algebra) on arrays
2. Arrays can be n-dimensional
3. There is no equivalent to append, although arrays can be concatenated

As we shall see, arrays can be created from existing collections such as lists, or instantiated "from scratch" in a few useful ways.

[5]:
```
# We need to import the numpy library to have access to it
# We can also create an alias for a library, this is something you will commonly␣
↪see with numpy
import numpy as np
```

1.2.1 Why do we need NumPy?

Does a question arise that why do we need a NumPy array when we have python lists? The answer is we can perform operations on all the elements of a NumPy array at once, which are not possible with python lists. For example, we can't multiply two lists directly we will have to do it element-wise. This is where the role of NumPy comes into play.

[6]:
```
list1 = [2, 4, 6, 7, 8]
list2 = [3, 4, 6, 1, 5]

print(list1*list2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-6-0193bd45e9db> in <module>
      2 list2 = [3, 4, 6, 1, 5]
      3
----> 4 print(list1*list2)

TypeError: can't multiply sequence by non-int of type 'list'
```

[7]:
```
import numpy as np

list1 = [2, 4, 6, 7, 8]
list2 = [3, 4, 6, 1, 5]

arr1 = np.array(list1)
arr2 = np.array(list2)

print(arr1*arr2)
```

```
[ 6 16 36  7 40]
```

1.2.2 Arrays

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. ***The number of dimensions is the rank of the array***; the ***shape*** of an array is a tuple of integers giving ***the size of the array along each dimension***.

We can initialize numpy arrays from nested Python lists, and access elements using square brackets:

[8]:
```python
a = np.array([1, 2, 3])  # Create a rank 1 array
print(type(a), a.shape, a[0], a[1], a[2])
a[0] = 5                 # Change an element of the array
print(a)
```

```
<class 'numpy.ndarray'> (3,) 1 2 3
[5 2 3]
```

[9]:
```python
b = np.array([[1,2,3],[4,5,6]])    # Create a rank 2 array
print(b)
print('The dimension of b is : ' + str(b.ndim))
```

```
[[1 2 3]
 [4 5 6]]
The dimension of b is : 2
```

[10]:
```python
print(b.shape)
print(b[0, 0], b[0, 1], b[1, 0])
```

```
(2, 3)
1 2 4
```

Numpy also provides many functions to create arrays:

[11]:
```python
a = np.zeros((2,2))   # Create an array of all zeros
print(a)

b = np.ones((1,2))    # Create an array of all ones
print(b)

c = np.full((2,2), 7) # Create a constant array
print(c)

d = np.eye(2)         # Create a 2x2 identity matrix
print(d)

e = np.random.random((2,2)) # Create an array filled with random values
print(e)
```

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

```
[[1. 1.]]
[[7 7]
 [7 7]]
[[1. 0.]
 [0. 1.]]
[[0.81289678 0.98905384]
 [0.02005508 0.52106272]]
```

[12]:
```
# Make an array from a list
alist = [2, 3, 4]
blist = [5, 6, 7]
a = np.array(alist)
b = np.array(blist)
print(a, type(a))
print(b, type(b))
```

```
[2 3 4] <class 'numpy.ndarray'>
[5 6 7] <class 'numpy.ndarray'>
```

1.2.3 Array Indexing

Numpy offers several ways to index into arrays.

Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array:

[13]:
```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(a)
```

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

[14]:
```
print(a.shape)
```

```
(3, 4)
```

[15]:
```
# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]
```

```
print(b)
```

```
[[2 3]
 [6 7]]
```

IMPORTANT : ***A slice of an array is a view into the same data, so modifying it will modify the original array***.

[16]:
```
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
b = a[:2, 1:3]
#
print("\n'a' matrix before slicing\n")
print(a)
#
# BEWARE: b[0, 0] is the same piece of data as a[0, 1] !!!
#
b[0, 0] = 77
#
print('\n'+ 100*'-' + "\n\n'a' matrix after slicing\n")
print(a)
```

```

'a' matrix before slicing

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

--------------------------------------------------------------------------------
--------------------

'a' matrix after slicing

[[ 1 77  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

Integer Indexing Vs Slicing

[17]:
```
# Create the following rank 2 array with shape (3, 4)
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(a)
```

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

Two ways of accessing the data in the middle row of the array. Using integer indexing yields an array of lower rank, while using slicing yields an array of the same rank as the original array:

[18]:
```
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)
print(row_r2, row_r2.shape)
```

```
[5 6 7 8] (4,)
[[5 6 7 8]] (1, 4)
```

[19]:
```
# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)
print()
print(col_r2, col_r2.shape)
```

```
[ 2  6 10] (3,)

[[ 2]
 [ 6]
 [10]] (3, 1)
```

When you index into numpy arrays using slicing, the resulting array view will always be a subarray of the original array. In contrast, integer array indexing allows you to construct arbitrary arrays using the data from another array. Here is an example:

[20]:
```
print(a)
```

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

[21]:
```
# An example of integer array indexing.
# The returned array will have shape (3,)
c = a[[0, 1, 2], [0, 2, 3]]
print(c)
print(c.shape)
```

```
[ 1  7 12]
(3,)
```

[22]:
```
# for example you can get immediately all the diagonal elements of a matrix
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12], [13,14,15,16]])
c= a[np.arange(a.shape[0]), np.arange(a.shape[1])]
print(a)
print('\n' + 100*'-' + '\n')
print(c)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
```

```
 [ 9 10 11 12]
 [13 14 15 16]]

--------------------------------------------------------------------------------
--------------------

[ 1  6 11 16]
```

IMPORTANT : In case of slice, a view of the array is returned but **index array a copy of the original array is returned.**

[23]:
```
c[:] = 42
print(c)
print('\n' + 100*'-' + '\n')
print(a)
```

```
[42 42 42 42]

--------------------------------------------------------------------------------
--------------------

[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
```

[24]:
```
# When using integer array indexing, you can reuse the same
# element from the source array:
print(a[[0, 0], [1, 1]])
```

```
[2 2]
```

One useful trick with integer array indexing is selecting or mutating one element from each row of a matrix:

[25]:
```
print(a)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
```

[26]:
```
# Create an array of indices
b = np.array([0, 2, 0, 1])

# Select one element from each row of a using the indices in b
print(a[np.arange(4), b])  # Prints "[ 1  7  9 14]"
```

```
[ 1  7  9 14]
```

[27]:
```python
# Mutate one element from each row of a using the indices in b
a[np.arange(4), b] =42
print(a)
```

```
[[42  2  3  4]
 [ 5  6 42  8]
 [42 10 11 12]
 [13 42 15 16]]
```

Slicing and indexing in a multidimensional array can be a little bit tricky compared to slicing and indexing in a one-dimensional array.

[28]:
```python
array = np.array([
    [2, 4, 5, 6],
    [3, 1, 6, 9],
    [4, 5, 1, 9],
    [2, 9, 1, 7]
])
print(array)

# Slicing and indexing in 4x4 array
# Print first two rows and first two columns
print("\nPrint first two rows and first two columns :\n\n", array[0:2, 0:2])

# Print all rows and last two columns
print("\nPrint all rows and last two columns        :\n\n", array[:, 2:4])

# Print all column but middle two rows
print("\nPrint all column but middle two rows       :\n\n", array[1:3, :])
```

```
[[2 4 5 6]
 [3 1 6 9]
 [4 5 1 9]
 [2 9 1 7]]

Print first two rows and first two columns :

 [[2 4]
 [3 1]]

Print all rows and last two columns        :

 [[5 6]
 [6 9]
 [1 9]
 [1 7]]

Print all column but middle two rows       :
```

```
[[3 1 6 9]
 [4 5 1 9]]
```

Boolean Array Indexing Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition. Here is an example:

[29]:
```python
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)  # Find the elements of a that are bigger than 2;
                    # this returns a numpy array of Booleans of the same
                    # shape as a, where each slot of bool_idx tells
                    # whether that element of a is > 2.

print(bool_idx)
```

```
[[False False]
 [ True  True]
 [ True  True]]
```

[30]:
```python
# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])

# We can do all of the above in a single concise statement:
print(a[a > 2])
```

```
[3 4 5 6]
[3 4 5 6]
```

For brevity we have left out a lot of details about numpy array indexing; if you want to know more you should read the documentation.

1.2.4 Datatypes

Every numpy array is a grid of elements of the same type. Numpy provides a large set of numeric datatypes that you can use to construct arrays. Numpy tries to guess a datatype when you create an array, but functions that construct arrays usually also include an optional argument to explicitly specify the datatype. Here is an example:

[31]:
```python
x = np.array([1, 2])  # Let numpy choose the datatype
y = np.array([1.0, 2.0])  # Let numpy choose the datatype
z = np.array([1, 2], dtype=np.int64)  # Force a particular datatype
```

```
print(x.dtype, y.dtype, z.dtype)
```

```
int32 float64 int64
```

You can read all about numpy datatypes in the documentation.

1.2.5 Array Math

Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions in the numpy module:

[32]:
```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
print(x + y)
print(np.add(x, y))
```

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

[33]:
```
# Elementwise difference; both produce the array
print(x - y)
print(np.subtract(x, y))
```

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

[34]:
```
# Elementwise product; both produce the array
print(x * y)
print(np.multiply(x, y))
```

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

[35]:
```
# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))
```

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

[36]:

```
# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

Note that unlike MATLAB, `*` is elementwise multiplication, not matrix multiplication. We instead use the dot function to compute inner products of vectors, to multiply a vector by a matrix, and to multiply matrices. dot is available both as a function in the numpy module and as an instance method of array objects:

[37]:

```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))
```

```
219
219
```

You can also use the `@` operator which is equivalent to numpy's `dot` operator.

[38]:

```
print(v @ w)
```

```
219
```

[39]:

```
# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))
print(x @ v)
```

```
[29 67]
[29 67]
[29 67]
```

[40]:

```
# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
```

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

```
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
```

Numpy provides many useful functions for performing computations on arrays; one of the most useful is `sum`:

[41]:
```
x = np.array([[1,2],[3,4]])

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

```
10
[4 6]
[3 7]
```

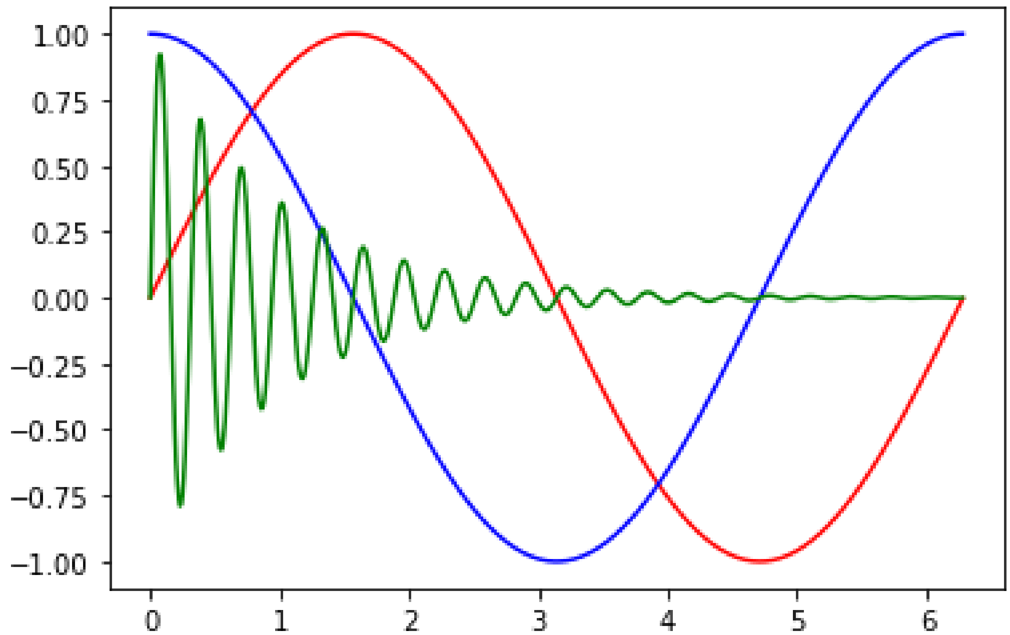
[42]:
```
import math

x = np.arange(0, 2*math.pi, 0.01)
y = np.sin(x)
z = np.cos(x)
w = np.sin(20*x)*np.exp(-x)
```

[43]:
```
import matplotlib.pyplot as plt

plt.plot(x, y, 'r')
plt.plot(x, z, 'b')
plt.plot(x, w, 'g')
plt.show
```

[43]:
```
<function matplotlib.pyplot.show(*args, **kw)>
```



You can find the full list of mathematical functions provided by numpy in the documentation.

Apart from computing mathematical functions using arrays, we frequently need to reshape or otherwise manipulate data in arrays. The simplest example of this type of operation is transposing a matrix; to transpose a matrix, simply use the T attribute of an array object:

[44]:
```
x = np.array([[1,2],[3,4]])
print(x)
print("transpose\n", x.T)
```

```
[[1 2]
 [3 4]]
transpose
 [[1 3]
 [2 4]]
```

[45]:
```
v = np.array([[1,2,3]])
print(v )
print("transpose\n", v.T)
```

```
[[1 2 3]]
transpose
 [[1]
 [2]
 [3]]
```

1.2.6 Broadcasting

Suppose we want to add a constant vector to each row of a matrix. We could do it like this:

[46]:
```
# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([42, 42, 42])
y = np.empty_like(x)   # Create an empty matrix with the same shape as x

# Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

print(y)
```

```
[[43 44 45]
 [46 47 48]
 [49 50 51]
 [52 53 54]]
```

This works; however when the matrix x is very large, computing an explicit loop in Python could be slow. Note that adding the vector v to each row of the matrix x is equivalent to forming a matrix vv by stacking multiple copies of v vertically, then performing elementwise summation of x and vv. We could implement this approach like this:

[47]:
```
vv = np.tile(v, (4, 1))  # Stack 4 copies of v on top of each other
print(vv)                # Prints "[[42 42 42]
                         #          [42 42 42]
                         #          [42 42 42]
                         #          [42 42 42]]"
```

```
[[42 42 42]
 [42 42 42]
 [42 42 42]
 [42 42 42]]
```

[48]:
```
y = x + vv  # Add x and vv elementwise
print(y)
```

```
[[43 44 45]
 [46 47 48]
 [49 50 51]
 [52 53 54]]
```

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array. For example, Numpy broadcasting allows us to perform this computation without actually creating multiple copies of v. Consider this version, using broadcasting:

[49]:
```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)
```

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

The line `y = x + v` works even though `x` has shape `(4, 3)` and `v` has shape `(3,)` due to broadcasting; this line works as if v actually had shape `(4, 3)`, where each row was a copy of `v`, and the sum was performed elementwise.

Broadcasting two arrays together follows these rules:

1. If the arrays do not have the same rank, prepend the shape of the lower rank array with 1s until both shapes have the same length.
2. The two arrays are said to be compatible in a dimension if they have the same size in the dimension, or if one of the arrays has size 1 in that dimension.
3. The arrays can be broadcast together if they are compatible in all dimensions.
4. After broadcasting, each array behaves as if it had shape equal to the elementwise maximum of shapes of the two input arrays.
5. In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension

If this explanation does not make sense, try reading the explanation from the documentation or this explanation.

Functions that support broadcasting are known as universal functions. You can find the list of all universal functions in the documentation.

Here are some applications of broadcasting:

[50]:
```
# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
# To compute an outer product, we first reshape v to be a column
# vector of shape (3, 1); we can then broadcast it against w to yield
# an output of shape (3, 2), which is the outer product of v and w:

print(np.reshape(v, (3, 1)) * w)
```

```
[[ 4  5]
```

```
 [ 8 10]
 [12 15]]
```

[51]:

```python
# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:
x = np.array([[1,2,3], [4,5,6]])
print('-----> w array:\n')
print(w)
print('\n-----> x array:\n')
print(x)
print('\n-----> x transpose:\n')
print(x.T)
print('\n-----> x transpose plus w:\n')
print(x.T + w)
print('\n-----> final result:\n')
print((x.T + w).T)
```

```
-----> w array:

[4 5]

-----> x array:

[[1 2 3]
 [4 5 6]]

-----> x transpose:

[[1 4]
 [2 5]
 [3 6]]

-----> x transpose plus w:

[[ 5  9]
 [ 6 10]
 [ 7 11]]

-----> final result:

[[ 5  6  7]
 [ 9 10 11]]
```

```
[52]: # Another solution is to reshape w to be a row vector of shape (2, 1);
# we can then broadcast it directly against x to produce the same
# output.
print(x + np.reshape(w, (2, 1)))
```

```
[[ 5  6  7]
 [ 9 10 11]]
```

```
[53]: # Multiply a matrix by a constant:
# x has shape (2, 3). Numpy treats scalars as arrays of shape ();
# these can be broadcast together to shape (2, 3), producing the
# following array:
print(x * 2)
```

```
[[ 2  4  6]
 [ 8 10 12]]
```

This brief overview has touched on many of the important things that you need to know about numpy, but is far from complete. Check out the numpy reference to find out much more about numpy.

1.2.7 NumPy in Data Science & Machine Learning

NumPy is a very popular Python library for large multi-dimensional array and matrix processing. With the help of a large collection of high-level mathematical functions it is very useful for fundamental scientific computations in Machine Learning.

It is particularly useful for,

- Linear Algebra
- Fourier Transform
- Random Number Generations

High-end libraries like TensorFlow uses NumPy internally for manipulation of Tensors.

Lots of ML concepts are tied up with linear algebra. It helps in

- To understand PCA(Principal Component Analysis),
- To build better ML algorithms from scratch,
- For processing Graphics in ML,
- It helps to understand Matrix factorization.

In fact, it could be said that ML completely uses matrix operations. The Linear Algebra module of NumPy offers various methods to apply linear algebra on any NumPy array. One can find:

- Rank, determinant, transpose, trace, inverse, etc. of an array.
- Eigenvalues and eigenvectors of the given matrices
- The dot product of two scalar values, as well as vector values.
- Solve a linear matrix equation and much more!

Example Calculating the inverse of a matrix

[54]:
```
array = np.array([
    [6, 1, 1],
    [4, -2, 5],
    [2, 8, 7]
])

inverse = np.linalg.inv(array)
print(inverse)
```

```
[[ 0.17647059 -0.00326797 -0.02287582]
 [ 0.05882353 -0.13071895  0.08496732]
 [-0.11764706  0.1503268   0.05228758]]
```

[55]:
```
print(np.round(array.dot(inverse), 8))
```

```
[[ 1.  0.  0.]
 [-0.  1.  0.]
 [-0.  0.  1.]]
```

Example Find eigenvalues and eigenvectors

[56]:
```
eigenVal, eigenVec = np.linalg.eig(array)
print(eigenVal)
print(eigenVec)
```

```
[11.24862343  5.09285054 -5.34147398]
[[ 0.24511338  0.75669314  0.02645665]
 [ 0.40622202 -0.03352363 -0.84078293]
 [ 0.88028581 -0.65291014  0.54072554]]
```

Example Solve a linear matrix equation

[57]:
```
A = np.array([
    [1, 3],
    [2, 4]
])

b = np.array([
    [7],
    [10]
])

x = np.linalg.solve(A, b)
print(x)
```

```
[[1.]
 [2.]]
```

1.3 Intro to Matplotlib

Matplotlib is a plotting library. In this section give a brief introduction to the `matplotlib.pyplot` module, which provides a plotting system similar to that of MATLAB.

[58]:
```
import matplotlib.pyplot as plt
```

By running this special iPython command, we will be displaying plots inline:

[59]:
```
%matplotlib inline
```

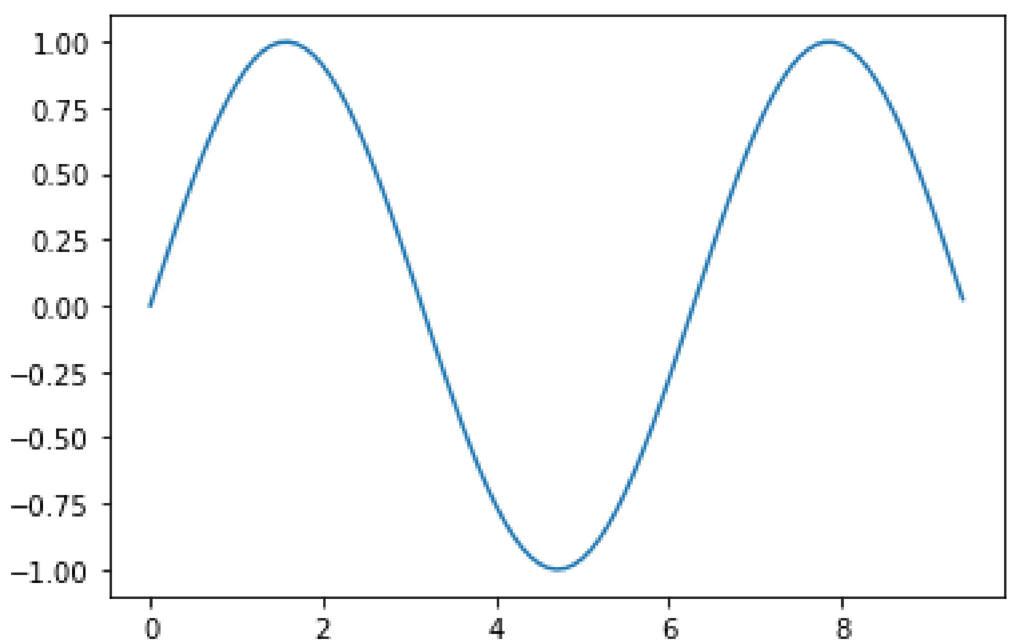
1.3.1 Plotting

The most important function in `matplotlib` is plot, which allows you to plot 2D data. Here is a simple example:

[60]:
```
# Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# Plot the points using matplotlib
plt.plot(x, y)
```

[60]:
```
[<matplotlib.lines.Line2D at 0x1cb4b16f208>]
```

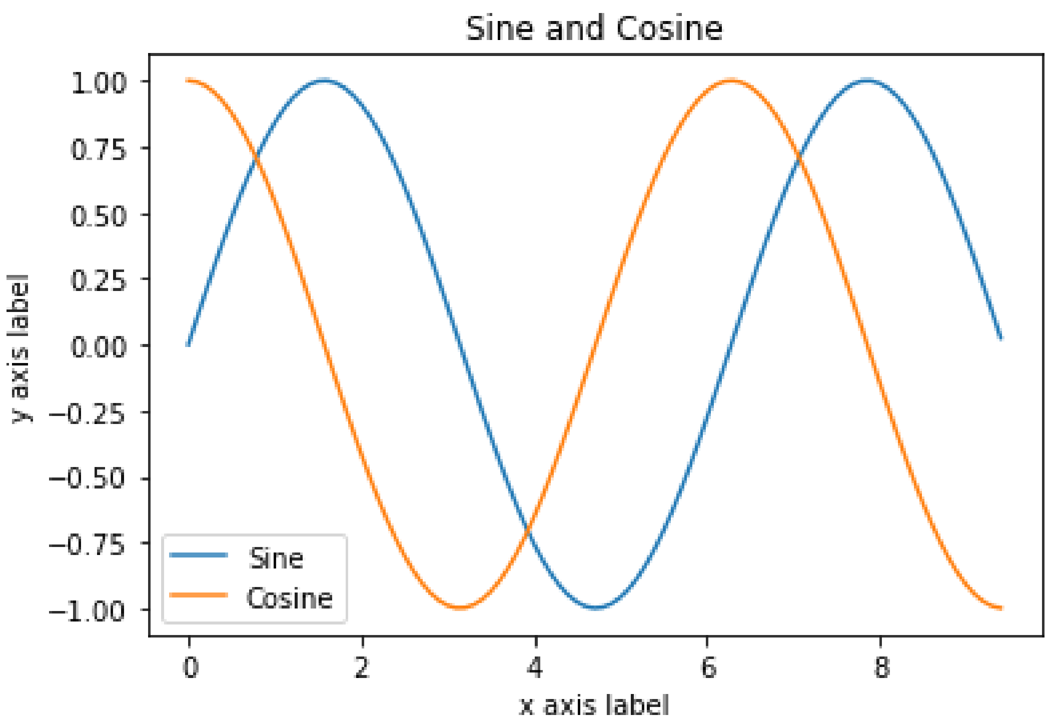


With just a little bit of extra work we can easily plot multiple lines at once, and add a title, legend, and axis labels:

[61]:
```
y_sin = np.sin(x)
y_cos = np.cos(x)

# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
```

[61]: <matplotlib.legend.Legend at 0x1cb4b1cbac8>



1.3.2 Subplots

You can plot different things in the same figure using the subplot function. Here is an example:

[62]:
```
# Compute the x and y coordinates for points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
```

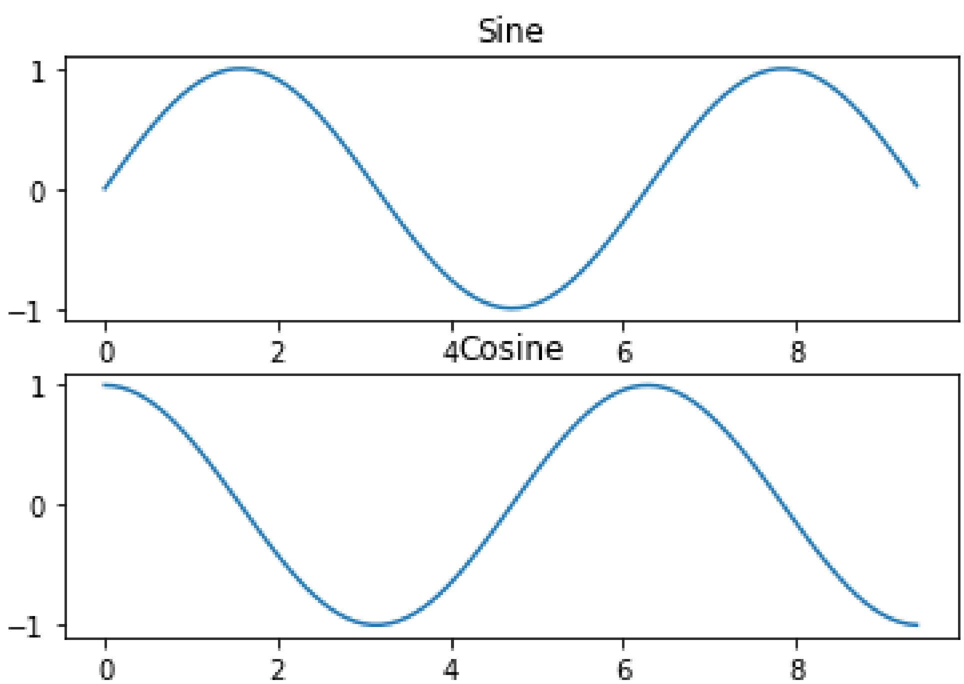
```
y_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,
# and set the first such subplot as active.
plt.subplot(2, 1, 1)

# Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

# Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# Show the figure.
plt.show()
```



You can read much more about the `subplot` function in the documentation.

1.4 Basic Pandas

The pandas package is probably the most important tool at the disposal of Data Scientists and Analysts working in Python today. The powerful machine learning and glamorous visualization tools may get all the attention, but pandas is the backbone of most data projects. To import pandas we usually import it with a shorter name since it's used so much:

[63]:
```
import pandas as pd
```

Now to the basic components of pandas.

1.4.1 Core components of pandas: Series and DataFrames

The primary two components of pandas are the `Series` and `DataFrame`.

A `Series` is essentially a column, and a `DataFrame` is a multi-dimensional table made up of a collection of Series.

DataFrames and Series are quite similar in that many operations that you can do with one you can do with the other, such as filling in null values and calculating the mean.

You'll see how these components work when we start working with data below.

1.4.2 Creating DataFrames from scratch

Creating DataFrames right in Python is good to know and quite useful when testing new methods and functions you find in the pandas docs.

There are *many* ways to create a DataFrame from scratch, a first option is to just use a simple `dict`.

[64]:
```
data = {
    'Open'  :[1.20575,1.20566,1.20582,1.20574,1.20596,1.20590],
    'High'  :[1.20576,1.20586,1.20592,1.20601,1.20615,1.20593],
    'Low'   :[1.20560,1.20565,1.20571,1.20569,1.20582,1.20580],
    'Close' :[1.20566,1.20582,1.20572,1.20597,1.20592,1.20588],
    'Volume':[212,88,83,184,246,131]
}
```

And then pass it to the pandas DataFrame constructor:

[65]:
```
fx_eur_usd = pd.DataFrame(data)
fx_eur_usd
```

[65]:

	Open	High	Low	Close	Volume
0	1.20575	1.20576	1.20560	1.20566	212
1	1.20566	1.20586	1.20565	1.20582	88
2	1.20582	1.20592	1.20571	1.20572	83
3	1.20574	1.20601	1.20569	1.20597	184
4	1.20596	1.20615	1.20582	1.20592	246
5	1.20590	1.20593	1.20580	1.20588	131

How did that work?

Each *(key, value)* item in `data` corresponds to a *column* in the resulting DataFrame.

The **Index** of this DataFrame was given to us on creation as the numbers 0-3, but we could also create our own when we initialize the DataFrame.

Let's have time as our index:

[66]:
```
fx_eur_usd=pd.DataFrame(data, index=['08/02/2021 15:25', \
                                     '08/02/2021 15:26', \
                                     '08/02/2021 15:27', \
                                     '08/02/2021 15:28', \
                                     '08/02/2021 15:29', \
                                     '08/02/2021 15:30'])
fx_eur_usd
```

[66]:

	Open	High	Low	Close	Volume
08/02/2021 15:25	1.20575	1.20576	1.20560	1.20566	212
08/02/2021 15:26	1.20566	1.20586	1.20565	1.20582	88
08/02/2021 15:27	1.20582	1.20592	1.20571	1.20572	83
08/02/2021 15:28	1.20574	1.20601	1.20569	1.20597	184
08/02/2021 15:29	1.20596	1.20615	1.20582	1.20592	246
08/02/2021 15:30	1.20590	1.20593	1.20580	1.20588	131

So now we could **loc**ate a price by using their time:

[67]:
```
fx_eur_usd.loc['08/02/2021 15:27']
```

[67]:
```
Open        1.20582
High        1.20592
Low         1.20571
Close       1.20572
Volume     83.00000
Name: 08/02/2021 15:27, dtype: float64
```

There's more on locating and extracting data from the DataFrame later, but now you should be able to create a DataFrame with any random data to learn on.

Let's move on to some quick methods for creating DataFrames from various other sources.

1.4.3 How to read in data

It's quite simple to load data from various file formats into a DataFrame. In the following examples we'll keep using our eur/usd forex data, but this time it's coming from various files.

Reading data from CSVs With CSV files all you need is a single line to load in the data:

[68]:
```
path = './data/csv/'
```

```
df = pd.read_csv(path + 'EURUSD_M1.csv')
df
```

[68]:
```
                       Time\tOpen\tHigh\tLow\tClose\tVolume
0      2021-02-08 15:25:00\t1.20575\t1.20576\t1.2056\...
1      2021-02-08 15:26:00\t1.20566\t1.20586\t1.20565...
2      2021-02-08 15:27:00\t1.20582\t1.20592\t1.20571...
3      2021-02-08 15:28:00\t1.20574\t1.20601\t1.20569...
4      2021-02-08 15:29:00\t1.20596\t1.20615\t1.20582...
...                                                  ...
49995  2021-03-29 12:55:00\t1.17814\t1.17827\t1.17812...
49996  2021-03-29 12:56:00\t1.17824\t1.17825\t1.17819...
49997  2021-03-29 12:57:00\t1.17825\t1.17847\t1.1781\...
49998  2021-03-29 12:58:00\t1.17846\t1.17855\t1.17839...
49999  2021-03-29 12:59:00\t1.17858\t1.17859\t1.17844...

[50000 rows x 1 columns]
```

[69]:
```
df = pd.read_csv(path + 'EURUSD_M1.csv', sep = '\t')
df
```

[69]:
```
                      Time     Open     High      Low    Close  Volume
0      2021-02-08 15:25:00  1.20575  1.20576  1.20560  1.20566     212
1      2021-02-08 15:26:00  1.20566  1.20586  1.20565  1.20582      88
2      2021-02-08 15:27:00  1.20582  1.20592  1.20571  1.20572      83
3      2021-02-08 15:28:00  1.20574  1.20601  1.20569  1.20597     184
4      2021-02-08 15:29:00  1.20596  1.20615  1.20582  1.20592     246
...                    ...      ...      ...      ...      ...     ...
49995  2021-03-29 12:55:00  1.17814  1.17827  1.17812  1.17823      88
49996  2021-03-29 12:56:00  1.17824  1.17825  1.17819  1.17825      58
49997  2021-03-29 12:57:00  1.17825  1.17847  1.17810  1.17845     109
49998  2021-03-29 12:58:00  1.17846  1.17855  1.17839  1.17855      92
49999  2021-03-29 12:59:00  1.17858  1.17859  1.17844  1.17855     154

[50000 rows x 6 columns]
```

CSVs don't have indexes like our DataFrames, so all we need to do is just designate the `index_col` when reading:

[70]:
```
df = pd.read_csv(path + 'EURUSD_M1.csv', sep = '\t', index_col=0)
df
```

[70]:
```
                        Open     High      Low    Close  Volume
Time
2021-02-08 15:25:00  1.20575  1.20576  1.20560  1.20566     212
2021-02-08 15:26:00  1.20566  1.20586  1.20565  1.20582      88
2021-02-08 15:27:00  1.20582  1.20592  1.20571  1.20572      83
```

```
2021-02-08 15:28:00  1.20574  1.20601  1.20569  1.20597     184
2021-02-08 15:29:00  1.20596  1.20615  1.20582  1.20592     246
...                      ...      ...      ...      ...     ...
2021-03-29 12:55:00  1.17814  1.17827  1.17812  1.17823      88
2021-03-29 12:56:00  1.17824  1.17825  1.17819  1.17825      58
2021-03-29 12:57:00  1.17825  1.17847  1.17810  1.17845     109
2021-03-29 12:58:00  1.17846  1.17855  1.17839  1.17855      92
2021-03-29 12:59:00  1.17858  1.17859  1.17844  1.17855     154

[50000 rows x 5 columns]
```

1.4.4 Most important DataFrame operations

DataFrames possess hundreds of methods and other operations that are crucial to any analysis. As a beginner, you should know the operations that perform simple transformations of your data and those that provide fundamental statistical analysis.

Let's load in the IMDB movies dataset to begin:

[71]:
```
movies_df = pd.read_csv(path + "IMDB-Movie-Data.csv", index_col="Title")
```

We're loading this dataset from a CSV and designating the movie titles to be our index.

Viewing your data The first thing to do when opening a new dataset is print out a few rows to keep as a visual reference. We accomplish this with `.head()`:

[1]:
```
movies_df.head()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-b687c1d924a0> in <module>
----> 1 movies_df.head()

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

`.head()` outputs the **first** five rows of your DataFrame by default, but we could also pass a number as well: `movies_df.head(10)` would output the top ten rows, for example.

To see the **last** five rows use `.tail()`. `tail()` also accepts a number, and in this case we printing the bottom two rows.:

[2]:
```
movies_df.tail(2)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-46b7a669ef61> in <module>
```

```
----> 1 movies_df.tail(2)

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

Typically when we load in a dataset, we like to view the first five or so rows to see what's under the hood. Here we can see the names of each column, the index, and examples of values in each row.

You'll notice that the index in our DataFrame is the *Title* column, which you can tell by how the word *Title* is slightly lower than the rest of the columns.

Getting info about your data `.info()` should be one of the very first commands you run after loading your data:

[3]:
```
movies_df.info()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-0dc782dfd5a1> in <module>
----> 1 movies_df.info()

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

`.info()` provides the essential details about your dataset, such as the number of rows and columns, the number of non-null values, what type of data is in each column, and how much memory your DataFrame is using.

Notice in our movies dataset we have some obvious missing values in the `Revenue` and `Metascore` columns. We'll look at how to handle those in a bit.

Seeing the datatype quickly is actually quite useful. Imagine you just imported some JSON and the integers were recorded as strings. You go to do some arithmetic and find an "unsupported operand" Exception because you can't do math with strings. Calling `.info()` will quickly point out that your column you thought was all integers are actually string objects.

Another fast and useful attribute is `.shape`, which outputs just a tuple of (rows, columns):

[4]:
```
movies_df.shape
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-971005451c9b> in <module>
----> 1 movies_df.shape

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

Note that `.shape` has no parentheses and is a simple tuple of format (rows, columns). So we have **1000 rows** and **11 columns** in our movies DataFrame.

You'll be going to `.shape` a lot when cleaning and transforming data. For example, you might filter some rows based on some criteria and then want to know quickly how many rows were removed.

Handling duplicates This dataset does not have duplicate rows, but it is always important to verify you aren't aggregating duplicate rows.

To demonstrate, let's simply just double up our movies DataFrame by appending it to itself:

[5]:
```
temp_df = movies_df.append(movies_df)

temp_df.shape
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-5-682e9f259182> in <module>
----> 1 temp_df = movies_df.append(movies_df)
      2
      3 temp_df.shape

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

Using `append()` will return a copy without affecting the original DataFrame. We are capturing this copy in `temp` so we aren't working with the real data.

Notice call `.shape` quickly proves our DataFrame rows have doubled.

Now we can try dropping duplicates:

[6]:
```
temp_df = temp_df.drop_duplicates()

temp_df.shape
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-2e135e66f439> in <module>
----> 1 temp_df = temp_df.drop_duplicates()
      2
      3 temp_df.shape

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

Just like `append()`, the `drop_duplicates()` method will also return a copy of your DataFrame, but this time with duplicates removed. Calling `.shape` confirms we're back to the 1000 rows of our original dataset.

It's a little verbose to keep assigning DataFrames to the same variable like in this example. For this reason, pandas has the `inplace` keyword argument on many of its methods. Using `inplace=True` will modify the DataFrame object in place:

[7]:
```
temp_df.drop_duplicates(inplace=True)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-d185a6b6c56f> in <module>
----> 1 temp_df.drop_duplicates(inplace=True)

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

Now our `temp_df` *will* have the transformed data automatically.

Another important argument for `drop_duplicates()` is `keep`, which has three possible options:

- `first`: (default) Drop duplicates except for the first occurrence.
- `last`: Drop duplicates except for the last occurrence.
- `False`: Drop all duplicates.

Since we didn't define the `keep` arugment in the previous example it was defaulted to `first`. This means that if two rows are the same pandas will drop the second row and keep the first row. Using `last` has the opposite effect: the first row is dropped.

`keep`, on the other hand, will drop all duplicates. If two rows are the same then both will be dropped. Watch what happens to `temp_df`:

[8]:
```
temp_df = movies_df.append(movies_df)  # make a new copy

temp_df.drop_duplicates(inplace=True, keep=False)

temp_df.shape
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-8-b0eef36cd1a0> in <module>
----> 1 temp_df = movies_df.append(movies_df)  # make a new copy
      2 
      3 temp_df.drop_duplicates(inplace=True, keep=False)
      4 
      5 temp_df.shape

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

Since all rows were duplicates, `keep=False` dropped them all resulting in zero rows being left over. If you're wondering why you would want to do this, one reason is that it allows you to locate all duplicates in your dataset. When conditional selections are shown below you'll see how to do that.

Column cleanup Many times datasets will have verbose column names with symbols, upper and lowercase words, spaces, and typos. To make selecting data by column name easier we can spend a little time cleaning up their names.

Here's how to print the column names of our dataset:

[20]:
```
movies_df.columns
```

[20]:
```
Index(['Rank', 'Genre', 'Description', 'Director', 'Actors', 'Year',
       'Runtime (Minutes)', 'Rating', 'Votes', 'Revenue (Millions)',
       'Metascore'],
      dtype='object')
```

Not only does `.columns` come in handy if you want to rename columns by allowing for simple copy and paste, it's also useful if you need to understand why you are receiving a `Key Error` when selecting data by column.

We can use the `.rename()` method to rename certain or all columns via a `dict`. We don't want parentheses, so let's rename those:

[21]:
```
movies_df.rename(columns={
        'Runtime (Minutes)': 'Runtime',
        'Revenue (Millions)': 'Revenue_millions'
    }, inplace=True)


movies_df.columns
```

[21]:
```
Index(['Rank', 'Genre', 'Description', 'Director', 'Actors', 'Year', 'Runtime',
       'Rating', 'Votes', 'Revenue_millions', 'Metascore'],
      dtype='object')
```

But what if we want to lowercase all names? Instead of using `.rename()` we could also set a list of names to the columns like so:

[22]:
```
movies_df.columns = ['rank', 'genre', 'description', 'director', 'actors',␣
 ↪'year', 'runtime',
                     'rating', 'votes', 'revenue_millions', 'metascore']


movies_df.columns
```

[22]:
```
Index(['rank', 'genre', 'description', 'director', 'actors', 'year', 'runtime',
       'rating', 'votes', 'revenue_millions', 'metascore'],
      dtype='object')
```

But that's too much work. Instead of just renaming each column manually we can do a list comprehension:

[23]:
```
movies_df.columns = [col.lower() for col in movies_df]

movies_df.columns
```

[23]:
```
Index(['rank', 'genre', 'description', 'director', 'actors', 'year', 'runtime',
        'rating', 'votes', 'revenue_millions', 'metascore'],
       dtype='object')
```

`list` (and `dict`) comprehensions come in handy a lot when working with pandas and data in general.

It's a good idea to lowercase, remove special characters, and replace spaces with underscores if you'll be working with a dataset for some time.

1.4.5 How to work with missing values

When exploring data, you'll most likely encounter missing or null values, which are essentially placeholders for non-existent values. Most commonly you'll see Python's `None` or NumPy's `np.nan`, each of which are handled differently in some situations.

There are two options in dealing with nulls:

1. Get rid of rows or columns with nulls
2. Replace nulls with non-null values, a technique known as **imputation**

Let's calculate to total number of nulls in each column of our dataset. The first step is to check which cells in our DataFrame are null:

[24]:
```
movies_df.isnull()
```

[24]:
```
                          rank  genre  description  director  actors   year  \
Title
Guardians of the Galaxy  False  False        False     False   False  False
Prometheus               False  False        False     False   False  False
Split                    False  False        False     False   False  False
Sing                     False  False        False     False   False  False
Suicide Squad            False  False        False     False   False  False
...                        ...    ...          ...       ...     ...    ...
Secret in Their Eyes     False  False        False     False   False  False
Hostel: Part II          False  False        False     False   False  False
Step Up 2: The Streets   False  False        False     False   False  False
Search Party             False  False        False     False   False  False
Nine Lives               False  False        False     False   False  False

                         runtime  rating  votes  revenue_millions  metascore
Title
```

```
Guardians of the Galaxy    False   False  False             False      False
Prometheus                 False   False  False             False      False
Split                      False   False  False             False      False
Sing                       False   False  False             False      False
Suicide Squad              False   False  False             False      False
...                          ...     ...    ...               ...        ...
Secret in Their Eyes       False   False  False              True      False
Hostel: Part II            False   False  False             False      False
Step Up 2: The Streets     False   False  False             False      False
Search Party               False   False  False              True      False
Nine Lives                 False   False  False             False      False

[1000 rows x 11 columns]
```

Notice `isnull()` returns a DataFrame where each cell is either True or False depending on that cell's null status.

To count the number of nulls in each column we use an aggregate function for summing:

[25]:
```
movies_df.isnull().sum()
```

[25]:
```
rank                 0
genre                0
description          0
director             0
actors               0
year                 0
runtime              0
rating               0
votes                0
revenue_millions   128
metascore           64
dtype: int64
```

`.isnull()` just by iteself isn't very useful, and is usually used in conjunction with other methods, like `sum()`.

We can see now that our data has **128** missing values for `revenue_millions` and **64** missing values for `metascore`.

Removing null values Data Scientists and Analysts regularly face the dilemma of dropping or imputing null values, and is a decision that requires intimate knowledge of your data and its context. Overall, removing null data is only suggested if you have a small amount of missing data.

Remove nulls is pretty simple:

[26]:
```
movies_df.dropna()
```

[26]:
```
                              rank                     genre  \
Title
Guardians of the Galaxy          1   Action,Adventure,Sci-Fi
Prometheus                       2  Adventure,Mystery,Sci-Fi
Split                            3           Horror,Thriller
Sing                             4   Animation,Comedy,Family
Suicide Squad                    5  Action,Adventure,Fantasy
...                            ...                       ...
Resident Evil: Afterlife       994   Action,Adventure,Horror
Project X                      995                    Comedy
Hostel: Part II                997                    Horror
Step Up 2: The Streets         998       Drama,Music,Romance
Nine Lives                    1000     Comedy,Family,Fantasy

                                                                 description  \
Title
Guardians of the Galaxy   A group of intergalactic criminals are forced ...
Prometheus                Following clues to the origin of mankind, a te...
Split                     Three girls are kidnapped by a man with a diag...
Sing                      In a city of humanoid animals, a hustling thea...
Suicide Squad             A secret government agency recruits some of th...
...                                                                     ...
Resident Evil: Afterlife  While still out to destroy the evil Umbrella C...
Project X                 3 high school seniors throw a birthday party t...
Hostel: Part II           Three American college students studying abroa...
Step Up 2: The Streets    Romantic sparks occur between two dance studen...
Nine Lives                A stuffy businessman finds himself trapped ins...

                                      director  \
Title
Guardians of the Galaxy             James Gunn
Prometheus                        Ridley Scott
Split                       M. Night Shyamalan
Sing                      Christophe Lourdelet
Suicide Squad                       David Ayer
...                                        ...
Resident Evil: Afterlife    Paul W.S. Anderson
Project X                      Nima Nourizadeh
Hostel: Part II                       Eli Roth
Step Up 2: The Streets              Jon M. Chu
Nine Lives                    Barry Sonnenfeld

                                                                      actors  \
Title
Guardians of the Galaxy   Chris Pratt, Vin Diesel, Bradley Cooper, Zoe S...
Prometheus                Noomi Rapace, Logan Marshall-Green, Michael Fa...
Split                     James McAvoy, Anya Taylor-Joy, Haley Lu Richar...
```

```
Sing                        Matthew McConaughey,Reese Witherspoon, Seth Ma...
Suicide Squad               Will Smith, Jared Leto, Margot Robbie, Viola D...
...                                                                      ...
Resident Evil: Afterlife    Milla Jovovich, Ali Larter, Wentworth Miller,K...
Project X                   Thomas Mann, Oliver Cooper, Jonathan Daniel Br...
Hostel: Part II             Lauren German, Heather Matarazzo, Bijou Philli...
Step Up 2: The Streets      Robert Hoffman, Briana Evigan, Cassie Ventura,...
Nine Lives                  Kevin Spacey, Jennifer Garner, Robbie Amell,Ch...

                            year  runtime  rating   votes  revenue_millions  \
Title
Guardians of the Galaxy     2014      121     8.1  757074            333.13
Prometheus                  2012      124     7.0  485820            126.46
Split                       2016      117     7.3  157606            138.12
Sing                        2016      108     7.2   60545            270.32
Suicide Squad               2016      123     6.2  393727            325.02
...                          ...      ...     ...     ...               ...
Resident Evil: Afterlife    2010       97     5.9  140900             60.13
Project X                   2012       88     6.7  164088             54.72
Hostel: Part II             2007       94     5.5   73152             17.54
Step Up 2: The Streets      2008       98     6.2   70699             58.01
Nine Lives                  2016       87     5.3   12435             19.64

                            metascore
Title
Guardians of the Galaxy          76.0
Prometheus                       65.0
Split                            62.0
Sing                             59.0
Suicide Squad                    40.0
...                               ...
Resident Evil: Afterlife         37.0
Project X                        48.0
Hostel: Part II                  46.0
Step Up 2: The Streets           50.0
Nine Lives                       11.0

[838 rows x 11 columns]
```

This operation will delete any **row** with at least a single null value, but it will return a new DataFrame without altering the original one. You could specify `inplace=True` in this method as well.

So in the case of our dataset, this operation would remove 128 rows where `revenue_millions` is null and 64 rows where `metascore` is null. This obviously seems like a waste since there's perfectly good data in the other columns of those dropped rows. That's why we'll look at imputation next.

Other than just dropping rows, you can also drop columns with null values by setting `axis=1`:

[27]:
```
movies_df.dropna(axis=1)
```

[27]:
```
                         rank                     genre  \
Title
Guardians of the Galaxy     1   Action,Adventure,Sci-Fi
Prometheus                  2  Adventure,Mystery,Sci-Fi
Split                       3           Horror,Thriller
Sing                        4   Animation,Comedy,Family
Suicide Squad               5  Action,Adventure,Fantasy
...                       ...                       ...
Secret in Their Eyes      996       Crime,Drama,Mystery
Hostel: Part II           997                    Horror
Step Up 2: The Streets    998       Drama,Music,Romance
Search Party              999          Adventure,Comedy
Nine Lives               1000     Comedy,Family,Fantasy

                                                               description  \
Title
Guardians of the Galaxy  A group of intergalactic criminals are forced ...
Prometheus               Following clues to the origin of mankind, a te...
Split                    Three girls are kidnapped by a man with a diag...
Sing                     In a city of humanoid animals, a hustling thea...
Suicide Squad            A secret government agency recruits some of th...
...                                                                    ...
Secret in Their Eyes     A tight-knit team of rising investigators, alo...
Hostel: Part II          Three American college students studying abroa...
Step Up 2: The Streets   Romantic sparks occur between two dance studen...
Search Party             A pair of friends embark on a mission to reuni...
Nine Lives               A stuffy businessman finds himself trapped ins...

                                     director  \
Title
Guardians of the Galaxy            James Gunn
Prometheus                       Ridley Scott
Split                      M. Night Shyamalan
Sing                     Christophe Lourdelet
Suicide Squad                      David Ayer
...                                       ...
Secret in Their Eyes                Billy Ray
Hostel: Part II                      Eli Roth
Step Up 2: The Streets            Jon M. Chu
Search Party                   Scot Armstrong
Nine Lives                   Barry Sonnenfeld

                                                                    actors  \
Title
Guardians of the Galaxy  Chris Pratt, Vin Diesel, Bradley Cooper, Zoe S...
```

```
Prometheus                Noomi Rapace, Logan Marshall-Green, Michael Fa...
Split                     James McAvoy, Anya Taylor-Joy, Haley Lu Richar...
Sing                      Matthew McConaughey,Reese Witherspoon, Seth Ma...
Suicide Squad             Will Smith, Jared Leto, Margot Robbie, Viola D...
...                                                                      ...
Secret in Their Eyes      Chiwetel Ejiofor, Nicole Kidman, Julia Roberts...
Hostel: Part II           Lauren German, Heather Matarazzo, Bijou Philli...
Step Up 2: The Streets    Robert Hoffman, Briana Evigan, Cassie Ventura,...
Search Party              Adam Pally, T.J. Miller, Thomas Middleditch,Sh...
Nine Lives                Kevin Spacey, Jennifer Garner, Robbie Amell,Ch...

                          year  runtime  rating   votes
Title
Guardians of the Galaxy   2014      121     8.1  757074
Prometheus                2012      124     7.0  485820
Split                     2016      117     7.3  157606
Sing                      2016      108     7.2   60545
Suicide Squad             2016      123     6.2  393727
...                        ...      ...     ...     ...
Secret in Their Eyes      2015      111     6.2   27585
Hostel: Part II           2007       94     5.5   73152
Step Up 2: The Streets    2008       98     6.2   70699
Search Party              2014       93     5.6    4881
Nine Lives                2016       87     5.3   12435

[1000 rows x 9 columns]
```

In our dataset, this operation would drop the `revenue_millions` and `metascore` columns.

Intuition side note: What's with this `axis=1` parameter?

It's not immediately obvious where `axis` comes from and why you need it to be 1 for it to affect columns. To see why, just look at the `.shape` output:

[28]:
```
movies_df.shape
```

[28]:
```
(1000, 11)
```

As we learned above, this is a tuple that represents the shape of the DataFrame, i.e. 1000 rows and 11 columns. Note that the *rows* are at index zero of this tuple and *columns* are at **index one** of this tuple. This is why `axis=1` affects columns. This comes from NumPy, and is a great example of why learning NumPy is worth your time.

1.4.6 Imputation

Imputation is a conventional feature engineering technique used to keep valuable data that have null values.

There may be instances where dropping every row with a null value removes too big a chunk from your dataset, so instead we can impute that null with another value, usually the **mean** or the **median** of that column.

Let's look at imputing the missing values in the `revenue_millions` column. First we'll extract that column into its own variable:

[29]:
```
revenue = movies_df['revenue_millions']
```

Using square brackets is the general way we select columns in a DataFrame.

If you remember back to when we created DataFrames from scratch, the keys of the `dict` ended up as column names. Now when we select columns of a DataFrame, we use brackets just like if we were accessing a Python dictionary.

`revenue` now contains a Series:

[30]:
```
revenue.head()
```

[30]:
```
Title
Guardians of the Galaxy    333.13
Prometheus                 126.46
Split                      138.12
Sing                       270.32
Suicide Squad              325.02
Name: revenue_millions, dtype: float64
```

Slightly different formatting than a DataFrame, but we still have our `Title` index.

We'll impute the missing values of revenue using the mean. Here's the mean value:

[31]:
```
revenue_mean = revenue.mean()

revenue_mean
```

[31]:
```
82.95637614678897
```

With the mean, let's fill the nulls using `fillna()`:

[32]:
```
revenue.fillna(revenue_mean, inplace=True)
```

We have now replaced all nulls in `revenue` with the mean of the column. Notice that by using `inplace=True` we have actually affected the original `movies_df`:

[33]:
```
movies_df.isnull().sum()
```

[33]:
```
rank                  0
genre                 0
description           0
director              0
actors                0
```

```
year                0
runtime             0
rating              0
votes               0
revenue_millions    0
metascore          64
dtype: int64
```

Imputing an entire column with the same value like this is a basic example. It would be a better idea to try a more granular imputation by Genre or Director.

For example, you would find the mean of the revenue generated in each genre individually and impute the nulls in each genre with that genre's mean.

Let's now look at more ways to examine and understand the dataset.

1.4.7 Understanding your variables

Using `describe()` on an entire DataFrame we can get a summary of the distribution of continuous variables:

[34]:
```
movies_df.describe()
```

[34]:
```
              rank         year      runtime       rating         votes  \
count  1000.000000  1000.000000  1000.000000  1000.000000  1.000000e+03
mean    500.500000  2012.783000   113.172000     6.723200  1.698083e+05
std     288.819436     3.205962    18.810908     0.945429  1.887626e+05
min       1.000000  2006.000000    66.000000     1.900000  6.100000e+01
25%     250.750000  2010.000000   100.000000     6.200000  3.630900e+04
50%     500.500000  2014.000000   111.000000     6.800000  1.107990e+05
75%     750.250000  2016.000000   123.000000     7.400000  2.399098e+05
max    1000.000000  2016.000000   191.000000     9.000000  1.791916e+06

       revenue_millions   metascore
count       1000.000000  936.000000
mean          82.956376   58.985043
std           96.412043   17.194757
min            0.000000   11.000000
25%           17.442500   47.000000
50%           60.375000   59.500000
75%           99.177500   72.000000
max          936.630000  100.000000
```

Understanding which numbers are continuous also comes in handy when thinking about the type of plot to use to represent your data visually.

`.describe()` can also be used on a categorical variable to get the count of rows, unique count of categories, top category, and freq of top category:

[35]:
```
movies_df['genre'].describe()
```

[35]:
```
count                        1000
unique                        207
top        Action,Adventure,Sci-Fi
freq                           50
Name: genre, dtype: object
```

This tells us that the genre column has 207 unique values, the top value is Action/Adventure/Sci-Fi, which shows up 50 times (freq).

`.value_counts()` can tell us the frequency of all values in a column:

[36]:
```
movies_df['genre'].value_counts().head(10)
```

[36]:
```
Action,Adventure,Sci-Fi       50
Drama                         48
Comedy,Drama,Romance          35
Comedy                        32
Drama,Romance                 31
Animation,Adventure,Comedy    27
Action,Adventure,Fantasy      27
Comedy,Drama                  27
Comedy,Romance                26
Crime,Drama,Thriller          24
Name: genre, dtype: int64
```

Relationships between continuous variables By using the correlation method `.corr()` we can generate the relationship between each continuous variable:

[37]:
```
movies_df.corr()
```

[37]:
```
                      rank      year   runtime    rating     votes  \
rank              1.000000 -0.261605 -0.221739 -0.219555 -0.283876
year             -0.261605  1.000000 -0.164900 -0.211219 -0.411904
runtime          -0.221739 -0.164900  1.000000  0.392214  0.407062
rating           -0.219555 -0.211219  0.392214  1.000000  0.511537
votes            -0.283876 -0.411904  0.407062  0.511537  1.000000
revenue_millions -0.252996 -0.117562  0.247834  0.189527  0.607941
metascore        -0.191869 -0.079305  0.211978  0.631897  0.325684

                  revenue_millions  metascore
rank                     -0.252996  -0.191869
year                     -0.117562  -0.079305
runtime                   0.247834   0.211978
rating                    0.189527   0.631897
votes                     0.607941   0.325684
```

```
revenue_millions          1.000000   0.133328
metascore                 0.133328   1.000000
```

Correlation tables are a numerical representation of the bivariate relationships in the dataset.

Positive numbers indicate a positive correlation — one goes up the other goes up — and negative numbers represent an inverse correlation — one goes up the other goes down. 1.0 indicates a perfect correlation.

So looking in the first row, first column we see `rank` has a perfect correlation with itself, which is obvious. On the other hand, the correlation between `votes` and `revenue_millions` is 0.6. A little more interesting.

Examining bivariate relationships comes in handy when you have an outcome or dependent variable in mind and would like to see the features most correlated to the increase or decrease of the outcome. You can visually represent bivariate relationships with scatterplots (seen below in the plotting section).

For a deeper look into data summarizations check out Essential Statistics for Data Science.

Let's now look more at manipulating DataFrames.

1.4.8 DataFrame slicing, selecting, extracting

Up until now we've focused on some basic summaries of our data. We've learned about simple column extraction using single brackets, and we imputed null values in a column using `fillna()`. Below are the other methods of slicing, selecting, and extracting you'll need to use constantly.

It's important to note that, although many methods are the same, DataFrames and Series have different attributes, so you'll need be sure to know which type you are working with or else you will receive attribute errors.

Let's look at working with columns first.

By column You already saw how to extract a column using square brackets like this:

[38]:
```python
genre_col = movies_df['genre']

type(genre_col)
```

[38]:
```
pandas.core.series.Series
```

This will return a *Series*. To extract a column as a *DataFrame*, you need to pass a list of column names. In our case that's just a single column:

[39]:
```python
genre_col = movies_df[['genre']]

type(genre_col)
```

[39]:
```
pandas.core.frame.DataFrame
```

Since it's just a list, adding another column name is easy:

[40]:
```
subset = movies_df[['genre', 'rating']]

subset.head()
```

[40]:
```
                                           genre  rating
Title
Guardians of the Galaxy   Action,Adventure,Sci-Fi     8.1
Prometheus               Adventure,Mystery,Sci-Fi     7.0
Split                             Horror,Thriller     7.3
Sing                      Animation,Comedy,Family     7.2
Suicide Squad            Action,Adventure,Fantasy     6.2
```

Now we'll look at getting data by rows.

By rows For rows, we have two options:

- `.loc` - **loc**ates by name
- `.iloc`- **loc**ates by numerical **i**ndex

Remember that we are still indexed by movie Title, so to use `.loc` we give it the Title of a movie:

[41]:
```
prom = movies_df.loc["Prometheus"]

prom
```

[41]:
```
rank                                                               2
genre                                       Adventure,Mystery,Sci-Fi
description        Following clues to the origin of mankind, a te...
director                                                Ridley Scott
actors             Noomi Rapace, Logan Marshall-Green, Michael Fa...
year                                                            2012
runtime                                                          124
rating                                                             7
votes                                                         485820
revenue_millions                                              126.46
metascore                                                         65
Name: Prometheus, dtype: object
```

On the other hand, with `iloc` we give it the numerical index of Prometheus:

[42]:
```
prom = movies_df.iloc[1]
```

`loc` and `iloc` can be thought of as similar to Python `list` slicing. To show this even further, let's select multiple rows.

How would you do it with a list? In Python, just slice with brackets like `example_list[1:4]`. It's works the same way in pandas:

[43]:
```
movie_subset = movies_df.loc['Prometheus':'Sing']

movie_subset = movies_df.iloc[1:4]

movie_subset
```

[43]:
```
            rank                     genre  \
Title
Prometheus     2  Adventure,Mystery,Sci-Fi
Split          3           Horror,Thriller
Sing           4   Animation,Comedy,Family

                                                  description  \
Title
Prometheus  Following clues to the origin of mankind, a te...
Split       Three girls are kidnapped by a man with a diag...
Sing        In a city of humanoid animals, a hustling thea...

                        director  \
Title
Prometheus          Ridley Scott
Split         M. Night Shyamalan
Sing        Christophe Lourdelet

                                                       actors  year  runtime  \
Title
Prometheus  Noomi Rapace, Logan Marshall-Green, Michael Fa...  2012      124
Split       James McAvoy, Anya Taylor-Joy, Haley Lu Richar...  2016      117
Sing        Matthew McConaughey,Reese Witherspoon, Seth Ma...  2016      108

            rating   votes  revenue_millions  metascore
Title
Prometheus     7.0  485820            126.46       65.0
Split          7.3  157606            138.12       62.0
Sing           7.2   60545            270.32       59.0
```

One important distinction between using `.loc` and `.iloc` to select multiple rows is that `.loc` includes the movie *Sing* in the result, but when using `.iloc` we're getting rows 1:4 but the movie at index 4 (*Suicide Squad*) is not included.

Slicing with `.iloc` follows the same rules as slicing with lists, the object at the index at the end is not included.

Conditional selections We've gone over how to select columns and rows, but what if we want to make a conditional selection?

For example, what if we want to filter our movies DataFrame to show only films directed by Ridley Scott or films with a rating greater than or equal to 8.0?

To do that, we take a column from the DataFrame and apply a Boolean condition to it. Here's an example of a Boolean condition:

[44]:
```
condition = (movies_df['director'] == "Ridley Scott")

condition.head()
```

[44]:
```
Title
Guardians of the Galaxy    False
Prometheus                  True
Split                      False
Sing                       False
Suicide Squad              False
Name: director, dtype: bool
```

Similar to `isnull()`, this returns a Series of True and False values: True for films directed by Ridley Scott and False for ones not directed by him.

We want to filter out all movies not directed by Ridley Scott, in other words, we don't want the False films. To return the rows where that condition is True we have to pass this operation into the DataFrame:

[45]:
```
movies_df[movies_df['director'] == "Ridley Scott"].head()
```

[45]:
```
                        rank                     genre  \
Title
Prometheus                 2  Adventure,Mystery,Sci-Fi
The Martian              103    Adventure,Drama,Sci-Fi
Robin Hood               388    Action,Adventure,Drama
American Gangster        471     Biography,Crime,Drama
Exodus: Gods and Kings   517    Action,Adventure,Drama


                                                              description  \
Title
Prometheus              Following clues to the origin of mankind, a te...
The Martian             An astronaut becomes stranded on Mars after hi...
Robin Hood              In 12th century England, Robin and his band of...
American Gangster       In 1970s America, a detective works to bring d...
Exodus: Gods and Kings  The defiant leader Moses rises up against the ...


                            director  \
Title
Prometheus              Ridley Scott
The Martian             Ridley Scott
Robin Hood              Ridley Scott
American Gangster       Ridley Scott
Exodus: Gods and Kings  Ridley Scott
```

```
                                                                actors  \
Title
Prometheus                Noomi Rapace, Logan Marshall-Green, Michael Fa...
The Martian               Matt Damon, Jessica Chastain, Kristen Wiig, Ka...
Robin Hood                Russell Crowe, Cate Blanchett, Matthew Macfady...
American Gangster         Denzel Washington, Russell Crowe, Chiwetel Eji...
Exodus: Gods and Kings    Christian Bale, Joel Edgerton, Ben Kingsley, S...

                          year  runtime  rating   votes  revenue_millions  \
Title
Prometheus                2012      124     7.0  485820            126.46
The Martian               2015      144     8.0  556097            228.43
Robin Hood                2010      140     6.7  221117            105.22
American Gangster         2007      157     7.8  337835            130.13
Exodus: Gods and Kings    2014      150     6.0  137299             65.01

                          metascore
Title
Prometheus                     65.0
The Martian                    80.0
Robin Hood                     53.0
American Gangster              76.0
Exodus: Gods and Kings         52.0
```

You can get used to looking at these conditionals by reading it like:

> Select movies_df where movies_df director equals Ridley Scott

Let's look at conditional selections using numerical values by filtering the DataFrame by ratings:

[46]:
```
movies_df[movies_df['rating'] >= 8.6].head(3)
```

[46]:
```
                 rank                     genre  \
Title
Interstellar       37    Adventure,Drama,Sci-Fi
The Dark Knight    55        Action,Crime,Drama
Inception          81   Action,Adventure,Sci-Fi

                                                       description  \
Title
Interstellar     A team of explorers travel through a wormhole ...
The Dark Knight  When the menace known as the Joker wreaks havo...
Inception        A thief, who steals corporate secrets through ...

                           director  \
Title
Interstellar     Christopher Nolan
The Dark Knight  Christopher Nolan
```

```
Inception        Christopher Nolan

                                                             actors  year  \
Title
Interstellar     Matthew McConaughey, Anne Hathaway, Jessica Ch...   2014
The Dark Knight  Christian Bale, Heath Ledger, Aaron Eckhart,Mi...   2008
Inception        Leonardo DiCaprio, Joseph Gordon-Levitt, Ellen...   2010

                 runtime  rating    votes  revenue_millions  metascore
Title
Interstellar         169     8.6  1047747            187.99       74.0
The Dark Knight      152     9.0  1791916            533.32       82.0
Inception            148     8.8  1583625            292.57       74.0
```

We can make some richer conditionals by using logical operators | for "or" and & for "and".

Let's filter the the DataFrame to show only movies by Christopher Nolan OR Ridley Scott:

[47]:
```
movies_df[(movies_df['director'] == 'Christopher Nolan') |␣
 ↪(movies_df['director'] == 'Ridley Scott')].head()
```

[47]:
```
                 rank                     genre  \
Title
Prometheus          2  Adventure,Mystery,Sci-Fi
Interstellar       37    Adventure,Drama,Sci-Fi
The Dark Knight    55        Action,Crime,Drama
The Prestige       65      Drama,Mystery,Sci-Fi
Inception          81   Action,Adventure,Sci-Fi

                                                       description  \
Title
Prometheus       Following clues to the origin of mankind, a te...
Interstellar     A team of explorers travel through a wormhole ...
The Dark Knight  When the menace known as the Joker wreaks havo...
The Prestige     Two stage magicians engage in competitive one-...
Inception        A thief, who steals corporate secrets through ...

                          director  \
Title
Prometheus            Ridley Scott
Interstellar     Christopher Nolan
The Dark Knight  Christopher Nolan
The Prestige     Christopher Nolan
Inception        Christopher Nolan

                                                             actors  year  \
Title
Prometheus       Noomi Rapace, Logan Marshall-Green, Michael Fa...   2012
```

```
Interstellar     Matthew McConaughey, Anne Hathaway, Jessica Ch...   2014
The Dark Knight  Christian Bale, Heath Ledger, Aaron Eckhart,Mi...   2008
The Prestige     Christian Bale, Hugh Jackman, Scarlett Johanss...   2006
Inception        Leonardo DiCaprio, Joseph Gordon-Levitt, Ellen...   2010

                 runtime  rating    votes  revenue_millions  metascore
Title
Prometheus           124     7.0   485820            126.46       65.0
Interstellar         169     8.6  1047747            187.99       74.0
The Dark Knight      152     9.0  1791916            533.32       82.0
The Prestige         130     8.5   913152             53.08       66.0
Inception            148     8.8  1583625            292.57       74.0
```

We need to make sure to group evaluations with parentheses so Python knows how to evaluate the conditional.

Using the isin() method we could make this more concise though:

[48]:
```
movies_df[movies_df['director'].isin(['Christopher Nolan', 'Ridley Scott'])].
↪head()
```

[48]:
```
                 rank                     genre  \
Title
Prometheus          2  Adventure,Mystery,Sci-Fi
Interstellar       37    Adventure,Drama,Sci-Fi
The Dark Knight    55        Action,Crime,Drama
The Prestige       65      Drama,Mystery,Sci-Fi
Inception          81   Action,Adventure,Sci-Fi

                                                       description  \
Title
Prometheus       Following clues to the origin of mankind, a te...
Interstellar     A team of explorers travel through a wormhole ...
The Dark Knight  When the menace known as the Joker wreaks havo...
The Prestige     Two stage magicians engage in competitive one-...
Inception        A thief, who steals corporate secrets through ...

                          director  \
Title
Prometheus            Ridley Scott
Interstellar     Christopher Nolan
The Dark Knight  Christopher Nolan
The Prestige     Christopher Nolan
Inception        Christopher Nolan

                                                            actors  year  \
Title
Prometheus       Noomi Rapace, Logan Marshall-Green, Michael Fa...   2012
```

```
Interstellar     Matthew McConaughey, Anne Hathaway, Jessica Ch...   2014
The Dark Knight  Christian Bale, Heath Ledger, Aaron Eckhart,Mi...   2008
The Prestige     Christian Bale, Hugh Jackman, Scarlett Johanss...   2006
Inception        Leonardo DiCaprio, Joseph Gordon-Levitt, Ellen...   2010

                 runtime  rating    votes  revenue_millions  metascore
Title
Prometheus           124     7.0   485820            126.46       65.0
Interstellar         169     8.6  1047747            187.99       74.0
The Dark Knight      152     9.0  1791916            533.32       82.0
The Prestige         130     8.5   913152             53.08       66.0
Inception            148     8.8  1583625            292.57       74.0
```

Let's say we want all movies that were released between 2005 and 2010, have a rating above 8.0, but made below the 25th percentile in revenue.

Here's how we could do all of that:

[49]:
```
movies_df[
    ((movies_df['year'] >= 2005) & (movies_df['year'] <= 2010))
    & (movies_df['rating'] > 8.0)
    & (movies_df['revenue_millions'] < movies_df['revenue_millions'].quantile(0.
↪25))
]
```

[49]:
```
                     rank                genre  \
Title
3 Idiots              431         Comedy,Drama
The Lives of Others   477       Drama,Thriller
Incendies             714    Drama,Mystery,War
Taare Zameen Par      992  Drama,Family,Music

                                                          description  \
Title
3 Idiots             Two friends are searching for their long lost ...
The Lives of Others  In 1984 East Berlin, an agent of the secret po...
Incendies            Twins journey to the Middle East to discover t...
Taare Zameen Par     An eight-year-old boy is thought to be a lazy ...

                                             director  \
Title
3 Idiots                              Rajkumar Hirani
The Lives of Others  Florian Henckel von Donnersmarck
Incendies                            Denis Villeneuve
Taare Zameen Par                           Aamir Khan

                                                          actors  year  \
Title
```

```
3 Idiots                Aamir Khan, Madhavan, Mona Singh, Sharman Joshi  2009
The Lives of Others  Ulrich MÃijhe, Martina Gedeck,Sebastian Koch, Ul...  2006
Incendies            Lubna Azabal, MÃľlissa DÃľsormeaux-Poulin, Maxim...  2010
Taare Zameen Par     Darsheel Safary, Aamir Khan, Tanay Chheda, Sac...  2007

                     runtime  rating   votes  revenue_millions  metascore
Title
3 Idiots                 170     8.4  238789              6.52       67.0
The Lives of Others      137     8.5  278103             11.28       89.0
Incendies                131     8.2   92863              6.86       80.0
Taare Zameen Par         165     8.5  102697              1.20       42.0
```

If you recall up when we used `.describe()` the 25th percentile for revenue was about 17.4, and we can access this value directly by using the `quantile()` method with a float of 0.25.

So here we have only four movies that match that criteria.

1.4.9 Applying functions

It is possible to iterate over a DataFrame or Series as you would with a list, but doing so — especially on large datasets — is very slow.

An efficient alternative is to `apply()` a function to the dataset. For example, we could use a function to convert movies with an 8.0 or greater to a string value of "good" and the rest to "bad" and use this transformed values to create a new column.

First we would create a function that, when given a rating, determines if it's good or bad:

[9]:
```
def rating_function(x):
    if x >= 8.0:
        return "good"
    else:
        return "bad"
```

Now we want to send the entire rating column through this function, which is what `apply()` does:

[10]:
```
movies_df["rating_category"] = movies_df["rating"].apply(rating_function)

movies_df.head(2)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-10-14645f4710d4> in <module>
----> 1 movies_df["rating_category"] = movies_df["rating"].apply(rating_function)
      2
      3 movies_df.head(2)
```

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

The `.apply()` method passes every value in the `rating` column through the `rating_function` and then returns a new Series. This Series is then assigned to a new column called `rating_category`.

You can also use anonymous functions as well. This lambda function achieves the same result as `rating_function`:

[11]:
```
movies_df["rating_category"] = movies_df["rating"].apply(lambda x: 'good' if x␣
↪>= 8.0 else 'bad')

movies_df.head(2)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-11-2e503968b610> in <module>
----> 1 movies_df["rating_category"] = movies_df["rating"].apply(lambda x: 'good'␣
↪if x >= 8.0 else 'bad')
      2
      3 movies_df.head(2)

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

Overall, using `apply()` will be much faster than iterating manually over rows because pandas is utilizing vectorization.

> Vectorization: a style of computer programming where operations are applied to whole arrays instead of individual elements —Wikipedia

A good example of high usage of `apply()` is during natural language processing (NLP) work. You'll need to apply all sorts of text cleaning functions to strings to prepare for machine learning.

1.4.10 Brief Plotting

Another great thing about pandas is that it integrates with Matplotlib, so you get the ability to plot directly off DataFrames and Series. To get started we need to import Matplotlib (`pip install matplotlib`):

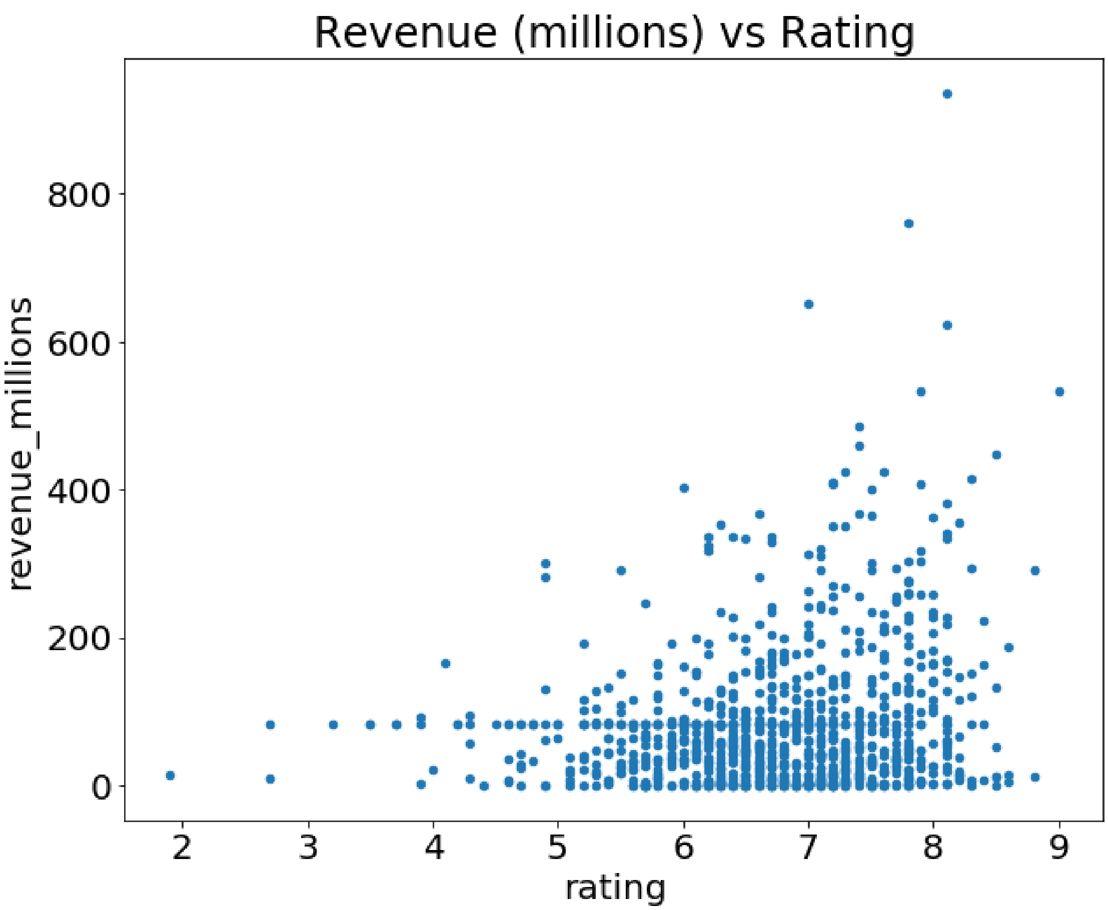
[53]:
```
import matplotlib.pyplot as plt
plt.rcParams.update({'font.size': 20, 'figure.figsize': (10, 8)}) # set font and␣
↪plot size to be larger
```

Now we can begin. There won't be a lot of coverage on plotting, but it should be enough to explore you're data easily.

Side note: For categorical variables utilize Bar Charts* and Boxplots. For continuous variables utilize Histograms, Scatterplots, Line graphs, and Boxplots.

Let's plot the relationship between ratings and revenue. All we need to do is call `.plot()` on `movies_df` with some info about how to construct the plot:

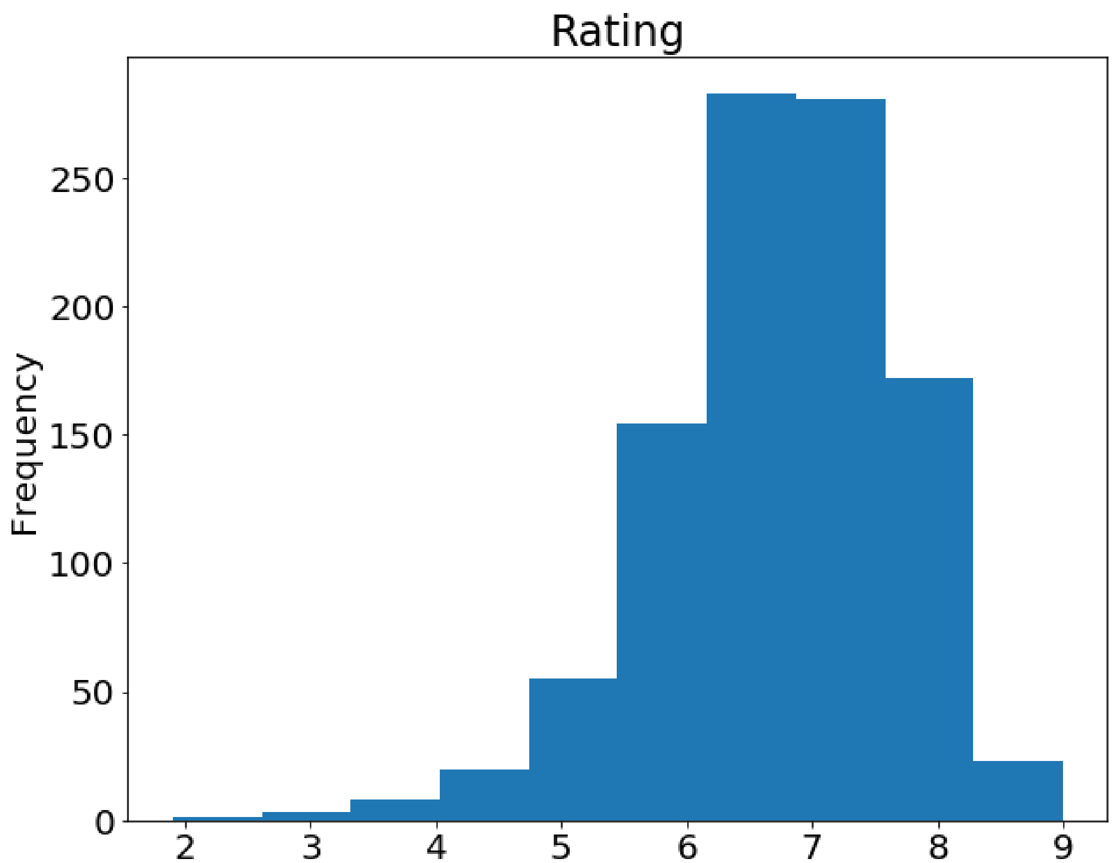
[54]:
```
movies_df.plot(kind='scatter', x='rating', y='revenue_millions', title='Revenue␣
↪(millions) vs Rating');
```



What's with the semicolon? It's not a syntax error, just a way to hide the `<matplotlib.axes._subplots.AxesSubplot at 0x26613b5cc18>` output when plotting in Jupyter notebooks.

If we want to plot a simple Histogram based on a single column, we can call plot on a column:

[55]:
```
movies_df['rating'].plot(kind='hist', title='Rating');
```



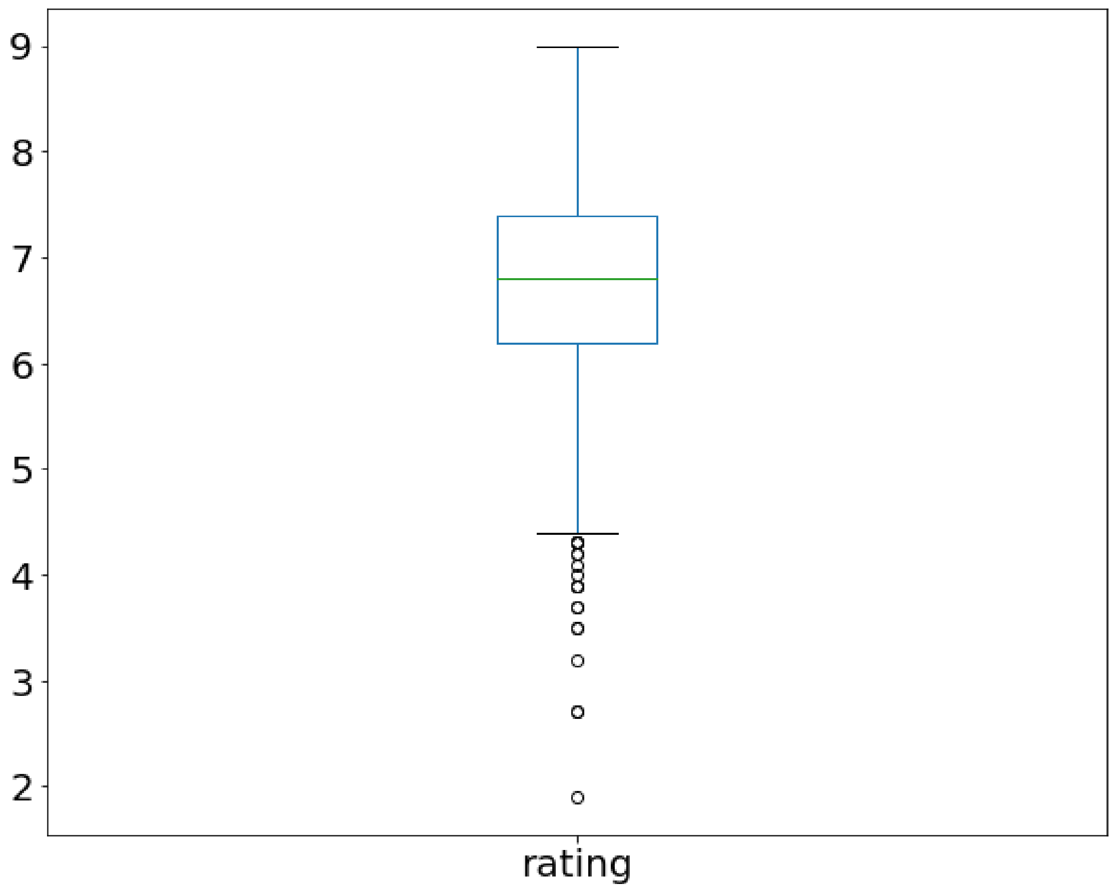
Do you remember the `.describe()` example at the beginning of this tutorial? Well, there's a graphical representation of the interquartile range, called the Boxplot. Let's recall what `describe()` gives us on the ratings column:

[56]:
```
movies_df['rating'].describe()
```

[56]:
```
count    1000.000000
mean        6.723200
std         0.945429
min         1.900000
25%         6.200000
50%         6.800000
75%         7.400000
max         9.000000
Name: rating, dtype: float64
```

Using a Boxplot we can visualize this data:

[57]:
```
movies_df['rating'].plot(kind="box");
```

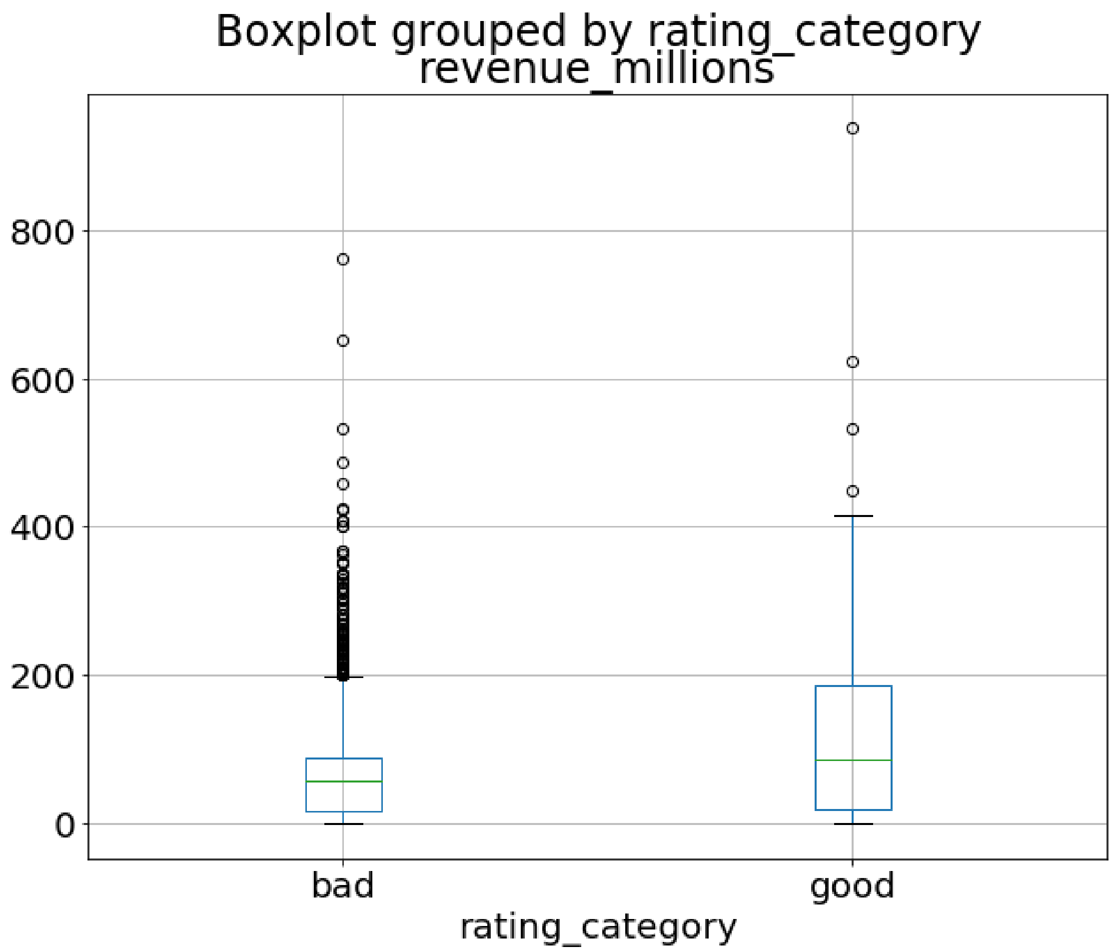


Source: *Flowing Data*

By combining categorical and continuous data, we can create a Boxplot of revenue that is grouped by the Rating Category we created above:

[58]:
```
movies_df.boxplot(column='revenue_millions', by='rating_category');
```

```
C:\Users\User\Anaconda3\lib\site-packages\numpy\core\_asarray.py:83:
VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths
or shapes) is deprecated. If you meant to do this, you must specify
'dtype=object' when creating the ndarray
  return array(a, dtype, copy=False, order=order)
```



That's the general idea of plotting with pandas. There's too many plots to mention, so definitely take a look at the `plot()` docs here for more information on what it can do.

2 References & Credits

To keep improving, view the extensive tutorials offered by the official pandas docs, follow along with a few Kaggle kernels, and keep working on your own projects!