Practical Tutorial on Data Manipulation with Numpy and Pandas in Python

Introduction

The pandas library has emerged into a power house of data manipulation tasks in python since it was developed in 2008. With its intuitive syntax and flexible data structure, it's easy to learn and enables faster data computation. The development of numpy and pandas libraries has extended python's multi-purpose nature to solve machine learning problems as well. The acceptance of python language in machine learning has been phenomenal since then.

This is just one more reason underlining the need for you to learn these libraries now. Published in early 2017, this blog claimed that python jobs outnumbered R jobs.

In this tutorial, we'll learn about using numpy and pandas libraries for data manipulation from scratch. Instead of going into theory, we'll take a practical approach.

First, we'll understand the syntax and commonly used functions of the respective libraries. Later, we'll work on a real-life data set.

Note: This tutorial is best suited for people who know the basics of python. No further knowledge is expected. Make sure you have python installed on your laptop.

Table of Contents

6 Important things you should know about Numpy and Pandas

1. The data manipulation capabilities of pandas are built on top of the numpy library. In a way, numpy is a dependency of the pandas library.
2. Pandas is best at handling tabular data sets comprising different variable types (integer, float, double, etc.). In addition, the pandas library can also be used to perform even the most naive of tasks such as loading data or doing feature engineering on time series data.
3. Numpy is most suitable for performing basic numerical computations such as mean, median, range, etc. Alongside, it also supports the creation of multi-dimensional arrays.
4. Numpy library can also be used to integrate C/C++ and Fortran code.
5. Remember, python is a zero indexing language unlike R where indexing starts at one.
6. The best part of learning pandas and numpy is the strong active community support you'll get from around the world.

Just to give you a flavor of the numpy library, we'll quickly go through its syntax structures and some important commands such as slicing, indexing, concatenation, etc. All these commands will come in handy when using pandas as well. Let's get started!

Starting with Numpy

In [1]:

```
#load the library and check its version, just to make sure we aren't using an ol
der version
import numpy as np
np.__version__
```

Out[1]:

```
'1.16.4'
```

In [2]:

```
#create a list comprising numbers from 0 to 9
L = list(range(10))
```

In [8]:

```
#converting integers to string - this style of handling lists is known as list c
omprehension.
#List comprehension offers a versatile way to handle list manipulations tasks ea
sily. We'll learn about them in future tutorials. Here's an example.

#Generally,
[type(item) for item in L]
[int, int, int, int, int, int, int, int, int, int]
```

Out[8]:

```
[int, int, int, int, int, int, int, int, int, int]
```

In [9]:

```
#Example
[str(c) for c in L]
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

Out[9]:

```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

Creating Arrays

Numpy arrays are homogeneous in nature, i.e., they comprise one data type (integer, float, double, etc.) unlike lists.

In [10]:

```
#creating arrays
np.zeros(10, dtype='int')
```

Out[10]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

In [11]:

```
#creating a 3 row x 5 column matrix
np.ones((3,5), dtype=float)
```

Out[11]:

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

In [12]:

```
#creating a matrix with a predefined value
np.full((3,5),1.23)
```

Out[12]:

```
array([[1.23, 1.23, 1.23, 1.23, 1.23],
       [1.23, 1.23, 1.23, 1.23, 1.23],
       [1.23, 1.23, 1.23, 1.23, 1.23]])
```

In [13]:

```
#create an array with a set sequence
np.arange(0, 20, 2)
```

Out[13]:

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

In [14]:

```
#create an array of even space between the given range of values
np.linspace(0, 1, 5)
```

Out[14]:

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

In [15]:

```
#create a 3x3 array with mean 0 and standard deviation 1 in a given dimension
np.random.normal(0, 1, (3,3))
```

Out[15]:

```
array([[ 0.02707637, -1.10789253,  0.6057782 ],
       [-0.38102892,  0.97694408, -0.10576023],
       [ 0.10142768, -0.17266347, -0.33814613]])
```

In [16]:

```
#create an identity matrix
np.eye(3)
```

Out[16]:

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

In [18]:

```
#set a random seed
np.random.seed(0)



x1 = np.random.randint(10, size=6) #one dimension
x2 = np.random.randint(10, size=(3,4)) #two dimension
x3 = np.random.randint(10, size=(3,4,5)) #three dimension
```

In [19]:

```
print("x3 ndim:", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
```

```
x3 ndim: 3
x3 shape: (3, 4, 5)
x3 size:  60
```

Array Indexing

The important thing to remember is that indexing in python starts at zero.

In [21]:

```
x1 = np.array([4, 3, 4, 4, 8, 4])
```

In [22]:

```
x1
```

Out[22]:

```
array([4, 3, 4, 4, 8, 4])
```

In [23]:

```
#assess value to index zero
x1[0]
```

Out[23]:

```
4
```

In [24]:

```
#assess fifth value
x1[4]
```

Out[24]:

```
8
```

In [25]:

```
#get the last value
x1[-1]
```

Out[25]:

```
4
```

In [26]:

```
#get the second last value
x1[-2]
```

Out[26]:

```
8
```

In [27]:

```
#in a multidimensional array, we need to specify row and column index
x2
```

Out[27]:

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

In [28]:

```
#1st row and 2nd column value
x2[2,3]
```

Out[28]:

```
7
```

In [29]:

```
#3rd row and last value from the 3rd column
x2[2,-1]
```

Out[29]:

```
7
```

In [30]:

```
#replace value at 0,0 index
x2[0,0] = 12
```

In [31]:

```
x2
```

Out[31]:

```
array([[12,  5,  2,  4],
       [ 7,  6,  8,  8],
       [ 1,  6,  7,  7]])
```

Array Slicing

Now, we'll learn to access multiple or a range of elements from an array.

In [32]:

```
x = np.arange(10)
```

In [33]:

```
x
```

Out[33]:

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

In [34]:

```
#from start to 4th position
x[:5]
```

Out[34]:

```
array([0, 1, 2, 3, 4])
```

In [35]:

```
#from 4th position to end
x[4:]
```

Out[35]:

```
array([4, 5, 6, 7, 8, 9])
```

In [36]:

```
#from 4th to 6th position
x[4:7]
```

Out[36]:

```
array([4, 5, 6])
```

In [37]:

```
#return elements at even place
x[ : : 2]
```

Out[37]:

```
array([0, 2, 4, 6, 8])
```

In [38]:

```
#return elements from first position step by two
x[1::2]
```

Out[38]:

```
array([1, 3, 5, 7, 9])
```

In [39]:

```
#reverse the array
x[::-1]
```

Out[39]:

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

Array Concatenation

Many a time, we are required to combine different arrays. So, instead of typing each of their elements manually, you can use array concatenation to handle such tasks easily.

In [40]:

```
#You can concatenate two or more arrays at once.
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
z = [21,21,21]
```

In [41]:

```
np.concatenate([x, y,z])
```

Out[41]:

```
array([ 1,  2,  3,  3,  2,  1, 21, 21, 21])
```

In [42]:

```
#You can also use this function to create 2-dimensional arrays.
grid = np.array([[1,2,3],[4,5,6]])
```

In [43]:

```
np.concatenate([grid,grid])
```

Out[43]:

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

In [44]:

```
#Using its axis parameter, you can define row-wise or column-wise matrix
np.concatenate([grid,grid],axis=1)
```

Out[44]:

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

Until now, we used the concatenation function of arrays of equal dimension. But, what if you are required to combine a 2D array with 1D array? In such situations, np.concatenate might not be the best option to use. Instead, you can use np.vstack or np.hstack to do the task. Let's see how!

In [45]:

```
x = np.array([3,4,5])
grid = np.array([[1,2,3],[17,18,19]])
```

In [46]:

```
np.vstack([x,grid])
```

Out[46]:

```
array([[ 3,  4,  5],
       [ 1,  2,  3],
       [17, 18, 19]])
```

In [48]:

```
#Similarly, you can add an array using np.hstack
z = np.array([[9],[9]])
np.hstack([grid,z])
```

Out[48]:

```
array([[ 1,  2,  3,  9],
       [17, 18, 19,  9]])
```

Also, we can split the arrays based on pre-defined positions. Let's see how!

In [49]:

```
x = np.arange(10)
```

In [50]:

```
x
```

Out[50]:

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

In [51]:

```
x1,x2,x3 = np.split(x,[3,6])
```

In [55]:

```
print (x1,x2,x3)
```

```
[0 1 2] [3 4 5] [6 7 8 9]
```

In [56]:

```
grid = np.arange(16).reshape((4,4))
```

In [57]:

```
grid
```

Out[57]:

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

In [58]:

```
upper,lower = np.vsplit(grid,[2])
```

In [59]:

```
print (upper, lower)
```

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

In addition to the functions we learned above, there are several other mathematical functions available in the numpy library such as sum, divide, multiple, abs, power, mod, sin, cos, tan, log, var, min, mean, max, etc. which you can be used to perform basic arithmetic calculations. Feel free to refer to numpy documentation for more information on such functions.

Let's move on to pandas now. Make sure you following each line below because it'll help you in doing data manipulation using pandas.

Let's start with Pandas

In [60]:

```
#load library - pd is just an alias. I used pd because it's short and literally
 abbreviates pandas.
#You can use any name as an alias.
import pandas as pd
```

In [61]:

```
#create a data frame - dictionary is used here where keys get converted to colum
n names and values to row values.
data = pd.DataFrame({'Country': ['Russia','Colombia','Chile','Equador','Nigeria'
],
                     'Rank':[121,40,100,130,11]})
data
```

Out[61]:

	Country	Rank
0	Russia	121
1	Colombia	40
2	Chile	100
3	Equador	130
4	Nigeria	11

In [62]:

```
#We can do a quick analysis of any data set using:
data.describe()
```

Out[62]:

	Rank
count	5.000000
mean	80.400000
std	52.300096
min	11.000000
25%	40.000000
50%	100.000000
75%	121.000000
max	130.000000

Remember, describe() method computes summary statistics of integer / double variables. To get the complete information about the data set, we can use info() function.

In [63]:

```
#Among other things, it shows the data set has 5 rows and 2 columns with their r
espective names.
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 2 columns):
Country    5 non-null object
Rank       5 non-null int64
dtypes: int64(1), object(1)
memory usage: 160.0+ bytes
```

In [64]:

```
#Let's create another data frame.
data = pd.DataFrame({'group':['a', 'a', 'a', 'b','b', 'b', 'c', 'c','c'],'ounce
s':[4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
```

In [66]:

```
data
```

Out[66]:

	group	ounces
0	a	4.0
1	a	3.0
2	a	12.0
3	b	6.0
4	b	7.5
5	b	8.0
6	c	3.0
7	c	5.0
8	c	6.0

In [65]:

```
#Let's sort the data frame by ounces - inplace = True will make changes to the data
data.sort_values(by=['ounces'],ascending=True,inplace=False)
```

Out[65]:

	group	ounces
1	a	3.0
6	c	3.0
0	a	4.0
7	c	5.0
3	b	6.0
8	c	6.0
4	b	7.5
5	b	8.0
2	a	12.0

We can sort the data by not just one column but multiple columns as well.

In [69]:

```
data.sort_values(by=['group','ounces'],ascending=[True,False],inplace=False)
```

Out[69]:

	group	ounces
2	a	12.0
0	a	4.0
1	a	3.0
5	b	8.0
4	b	7.5
3	b	6.0
8	c	6.0
7	c	5.0
6	c	3.0

Often, we get data sets with duplicate rows, which is nothing but noise. Therefore, before training the model, we need to make sure we get rid of such inconsistencies in the data set. Let's see how we can remove duplicate rows.

In [71]:

```
#create another data with duplicated rows
data = pd.DataFrame({'k1':['one']*3 + ['two']*4, 'k2':[3,2,1,3,3,4,4]})
```

In [72]:

```
data
```

Out[72]:

	k1	k2
0	one	3
1	one	2
2	one	1
3	two	3
4	two	3
5	two	4
6	two	4

In [73]:

```
#sort values
data.sort_values(by='k2')
```

Out[73]:

	k1	k2
2	one	1
1	one	2
0	one	3
3	two	3
4	two	3
5	two	4
6	two	4

In [74]:

```
#remove duplicates - ta da!
data.drop_duplicates()
```

Out[74]:

	k1	k2
0	one	3
1	one	2
2	one	1
3	two	3
5	two	4

Here, we removed duplicates based on matching row values across all columns. Alternatively, we can also remove duplicates based on a particular column. Let's remove duplicate values from the k1 column.

In [75]:

```
data.drop_duplicates(subset='k1')
```

Out[75]:

	k1	k2
0	one	3
3	two	3

Now, we will learn to categorize rows based on a predefined criteria. It happens a lot while data processing where you need to categorize a variable. For example, say we have got a column with country names and we want to create a new variable 'continent' based on these country names. In such situations, we will require the steps below:

In [76]:

```
data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon', 'Pastrami','corne
d beef', 'Bacon', 'pastrami', 'honey ham','nova lox'],
                     'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
```

In [77]:

```
data
```

Out[77]:

	food	ounces
0	bacon	4.0
1	pulled pork	3.0
2	bacon	12.0
3	Pastrami	6.0
4	corned beef	7.5
5	Bacon	8.0
6	pastrami	3.0
7	honey ham	5.0
8	nova lox	6.0

Now, we want to create a new variable which indicates the type of animal which acts as the source of the food. To do that, first we'll create a dictionary to map the food to the animals. Then, we'll use map function to map the dictionary's values to the keys. Let's see how is it done.

In [78]:

```
meat_to_animal = {
'bacon': 'pig',
'pulled pork': 'pig',
'pastrami': 'cow',
'corned beef': 'cow',
'honey ham': 'pig',
'nova lox': 'salmon'
}

def meat_2_animal(series):
    if series['food'] == 'bacon':
        return 'pig'
    elif series['food'] == 'pulled pork':
        return 'pig'
    elif series['food'] == 'pastrami':
        return 'cow'
    elif series['food'] == 'corned beef':
        return 'cow'
    elif series['food'] == 'honey ham':
        return 'pig'
    else:
        return 'salmon'
```

In [79]:

```
#create a new variable
data['animal'] = data['food'].map(str.lower).map(meat_to_animal)
```

In [80]:

```
data
```

Out[80]:

	food	ounces	animal
0	bacon	4.0	pig
1	pulled pork	3.0	pig
2	bacon	12.0	pig
3	Pastrami	6.0	cow
4	corned beef	7.5	cow
5	Bacon	8.0	pig
6	pastrami	3.0	cow
7	honey ham	5.0	pig
8	nova lox	6.0	salmon

In [81]:

```
#another way of doing it is: convert the food values to the lower case and apply the function
lower = lambda x: x.lower()
data['food'] = data['food'].apply(lower)
data['animal2'] = data.apply(meat_2_animal, axis='columns')
```

In [82]:

```
data
```

Out[82]:

	food	ounces	animal	animal2
0	bacon	4.0	pig	pig
1	pulled pork	3.0	pig	pig
2	bacon	12.0	pig	pig
3	pastrami	6.0	cow	cow
4	corned beef	7.5	cow	cow
5	bacon	8.0	pig	pig
6	pastrami	3.0	cow	cow
7	honey ham	5.0	pig	pig
8	nova lox	6.0	salmon	salmon

Another way to create a new variable is by using the assign function. With this tutorial, as you keep discovering the new functions, you'll realize how powerful pandas is.

In [84]:

```
data.assign(new_variable = data['ounces']*10)
```

Out[84]:

	food	ounces	animal	animal2	new_variable
0	bacon	4.0	pig	pig	40.0
1	pulled pork	3.0	pig	pig	30.0
2	bacon	12.0	pig	pig	120.0
3	pastrami	6.0	cow	cow	60.0
4	corned beef	7.5	cow	cow	75.0
5	bacon	8.0	pig	pig	80.0
6	pastrami	3.0	cow	cow	30.0
7	honey ham	5.0	pig	pig	50.0
8	nova lox	6.0	salmon	salmon	60.0

Let's remove the column animal2 from our data frame.

In [85]:

```
data.drop('animal2',axis='columns',inplace=True)
```

In [86]:

```
data
```

Out[86]:

	food	ounces	animal
0	bacon	4.0	pig
1	pulled pork	3.0	pig
2	bacon	12.0	pig
3	pastrami	6.0	cow
4	corned beef	7.5	cow
5	bacon	8.0	pig
6	pastrami	3.0	cow
7	honey ham	5.0	pig
8	nova lox	6.0	salmon

We frequently find missing values in our data set. A quick method for imputing missing values is by filling the missing value with any random number. Not just missing values, you may find lots of outliers in your data set, which might require replacing. Let's see how can we replace values.

In [87]:

```
#Series function from pandas are used to create arrays
data = pd.Series([1., -999., 2., -999., -1000., 3.])
```

In [88]:

```
data
```

Out[88]:

```
0       1.0
1    -999.0
2       2.0
3    -999.0
4   -1000.0
5       3.0
dtype: float64
```

In [89]:

```
#replace -999 with NaN values
data.replace(-999, np.nan,inplace=True)
```

In [90]:

```
data
```

Out[90]:

```
0       1.0
1       NaN
2       2.0
3       NaN
4   -1000.0
5       3.0
dtype: float64
```

In [91]:

```
#We can also replace multiple values at once.
data = pd.Series([1., -999., 2., -999., -1000., 3.])
data.replace([-999,-1000],np.nan,inplace=True)
```

In [92]:

```
data
```

Out[92]:

```
0    1.0
1    NaN
2    2.0
3    NaN
4    NaN
5    3.0
dtype: float64
```

Now, let's learn how to rename column names and axis (row names).

In [95]:

```
data = pd.DataFrame(np.arange(12).reshape((3, 4)),
index=['Ohio', 'Colorado', 'New York'],columns=['one', 'two', 'three', 'four'])
```

In [96]:

```
data
```

Out[96]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
New York	8	9	10	11

In [97]:

```
#Using rename function
data.rename(index = {'Ohio':'SanF'}, columns={'one':'one_p','two':'two_p'},inpla
ce=True)
```

In [98]:

```
data
```

Out[98]:

	one_p	two_p	three	four
SanF	0	1	2	3
Colorado	4	5	6	7
New York	8	9	10	11

In [99]:

```
#You can also use string functions
data.rename(index = str.upper, columns=str.title,inplace=True)
```

In [100]:

```
data
```

Out[100]:

	One_P	Two_P	Three	Four
SANF	0	1	2	3
COLORADO	4	5	6	7
NEW YORK	8	9	10	11

Next, we'll learn to categorize (bin) continuous variables.

In [101]:

```
ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

We'll divide the ages into bins such as 18-25, 26-35,36-60 and 60 and above.

In [102]:

```
#Understand the output - '(' means the value is included in the bin, '[' means t
he value is excluded
bins = [18, 25, 35, 60, 100]
cats = pd.cut(ages, bins)
```

In [103]:

```
cats
```

Out[103]:

```
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (6
0, 100], (35, 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] <
(60, 100]]
```

In [104]:

```
#To include the right bin value, we can do:
pd.cut(ages,bins,right=False)
```

Out[104]:

```
[[18, 25), [18, 25), [25, 35), [25, 35), [18, 25), ..., [25, 35), [6
0, 100), [35, 60), [35, 60), [25, 35)]
Length: 12
Categories (4, interval[int64]): [[18, 25) < [25, 35) < [35, 60) <
[60, 100)]
```

In [113]:

```
#Let's check how many observations fall under each bin
pd.value_counts(cats)
```

Out[113]:

```
(18, 25]     5
(35, 60]     3
(25, 35]     3
(60, 100]    1
dtype: int64
```

Also, we can pass a unique name to each label.

In [114]:

```
bin_names = ['Youth', 'YoungAdult', 'MiddleAge', 'Senior']
new_cats = pd.cut(ages, bins,labels=bin_names)

pd.value_counts(new_cats)
```

Out[114]:

```
Youth         5
MiddleAge     3
YoungAdult    3
Senior        1
dtype: int64
```

In [115]:

```
#we can also calculate their cumulative sum
pd.value_counts(new_cats).cumsum()
```

Out[115]:

```
Youth          5
MiddleAge      8
YoungAdult    11
Senior        12
dtype: int64
```

In [116]:

```
#we can also calculate their cumulative sum
pd.value_counts(new_cats).cumsum()
```

Out[116]:

```
Youth          5
MiddleAge      8
YoungAdult    11
Senior        12
dtype: int64
```

Let's proceed and learn about grouping data and creating pivots in pandas. It's an immensely important data analysis method which you'd probably have to use on every data set you work with.

In [117]:

```
df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
                   'key2' : ['one', 'two', 'one', 'two', 'one'],
                   'data1' : np.random.randn(5),
                   'data2' : np.random.randn(5)})
```

In [118]:

```
df
```

Out[118]:

	key1	key2	data1	data2
0	a	one	1.254414	1.149076
1	a	two	1.419102	-1.193578
2	b	one	-0.743856	1.141042
3	b	two	-2.517437	1.509445
4	a	one	-1.507096	1.067775

In [119]:

```
#calculate the mean of data1 column by key1
grouped = df['data1'].groupby(df['key1'])
```

In [121]:

```
grouped.mean()
```

Out[121]:

```
key1
a    0.388807
b   -1.630647
Name: data1, dtype: float64
```

Now, let's see how to slice the data frame.

In [122]:

```
dates = pd.date_range('20130101',periods=6)
df = pd.DataFrame(np.random.randn(6,4),index=dates,columns=list('ABCD'))
```

In [123]:

```
df
```

Out[123]:

	A	B	C	D
2013-01-01	-0.686589	0.014873	-0.375666	-0.038224
2013-01-02	0.367974	-0.044724	-0.302375	-2.224404
2013-01-03	0.724006	0.359003	1.076121	0.192141
2013-01-04	0.852926	0.018357	0.428304	0.996278
2013-01-05	-0.491150	0.712678	1.113340	-2.153675
2013-01-06	-0.416111	-1.070897	0.221139	-1.123057

In [124]:

```
#get first n rows from the data frame
df[:3]
```

Out[124]:

	A	B	C	D
2013-01-01	-0.686589	0.014873	-0.375666	-0.038224
2013-01-02	0.367974	-0.044724	-0.302375	-2.224404
2013-01-03	0.724006	0.359003	1.076121	0.192141

In [125]:

```
#slice based on date range
df['20130101':'20130104']
```

Out[125]:

	A	B	C	D
2013-01-01	-0.686589	0.014873	-0.375666	-0.038224
2013-01-02	0.367974	-0.044724	-0.302375	-2.224404
2013-01-03	0.724006	0.359003	1.076121	0.192141
2013-01-04	0.852926	0.018357	0.428304	0.996278

In [126]:

```
#slicing based on column names
df.loc[:,['A','B']]
```

Out[126]:

	A	B
2013-01-01	-0.686589	0.014873
2013-01-02	0.367974	-0.044724
2013-01-03	0.724006	0.359003
2013-01-04	0.852926	0.018357
2013-01-05	-0.491150	0.712678
2013-01-06	-0.416111	-1.070897

In [127]:

```
#slicing based on both row index labels and column names
df.loc['20130102':'20130103',['A','B']]
```

Out[127]:

	A	B
2013-01-02	0.367974	-0.044724
2013-01-03	0.724006	0.359003

In [128]:

```
#slicing based on index of columns
df.iloc[3] #returns 4th row (index is 3rd)
```

Out[128]:

```
A    0.852926
B    0.018357
C    0.428304
D    0.996278
Name: 2013-01-04 00:00:00, dtype: float64
```

In [129]:

```
#returns a specific range of rows
df.iloc[2:4, 0:2]
```

Out[129]:

	A	B
2013-01-03	0.724006	0.359003
2013-01-04	0.852926	0.018357

In [130]:

```
#returns specific rows and columns using lists containing columns or row indexes
df.iloc[[1,5],[0,2]]
```

Out[130]:

	A	C
2013-01-02	0.367974	-0.302375
2013-01-06	-0.416111	0.221139

Similarly, we can do Boolean indexing based on column values as well. This helps in filtering a data set based on a pre-defined condition.

In [132]:

```
#we can copy the data set
df2 = df.copy()
df2['E']=['one', 'one','two','three','four','three']
df2
```

Out[132]:

	A	B	C	D	E
2013-01-01	-0.686589	0.014873	-0.375666	-0.038224	one
2013-01-02	0.367974	-0.044724	-0.302375	-2.224404	one
2013-01-03	0.724006	0.359003	1.076121	0.192141	two
2013-01-04	0.852926	0.018357	0.428304	0.996278	three
2013-01-05	-0.491150	0.712678	1.113340	-2.153675	four
2013-01-06	-0.416111	-1.070897	0.221139	-1.123057	three

In [133]:

```
#select rows based on column values
df2[df2['E'].isin(['two','four'])]
```

Out[133]:

	A	B	C	D	E
2013-01-03	0.724006	0.359003	1.076121	0.192141	two
2013-01-05	-0.491150	0.712678	1.113340	-2.153675	four

In [134]:

```
#select all rows except those with two and four
df2[~df2['E'].isin(['two','four'])]
```

Out[134]:

	A	B	C	D	E
2013-01-01	-0.686589	0.014873	-0.375666	-0.038224	one
2013-01-02	0.367974	-0.044724	-0.302375	-2.224404	one
2013-01-04	0.852926	0.018357	0.428304	0.996278	three
2013-01-06	-0.416111	-1.070897	0.221139	-1.123057	three

We can also use a query method to select columns based on a criterion. Let's see how!

In [138]:

```
#list all columns where A is greater than C
df.query('A > C')
```

Out[138]:

	A	B	C	D
2013-01-02	0.367974	-0.044724	-0.302375	-2.224404
2013-01-04	0.852926	0.018357	0.428304	0.996278

In [139]:

```
#using OR condition
df.query('A < B | C > A')
```

Out[139]:

	A	B	C	D
2013-01-01	-0.686589	0.014873	-0.375666	-0.038224
2013-01-03	0.724006	0.359003	1.076121	0.192141
2013-01-05	-0.491150	0.712678	1.113340	-2.153675
2013-01-06	-0.416111	-1.070897	0.221139	-1.123057

In [140]:

```
#create a data frame
data = pd.DataFrame({'group': ['a', 'a', 'a', 'b','b', 'b', 'c', 'c','c'],
                     'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
data
```

Out[140]:

	group	ounces
0	a	4.0
1	a	3.0
2	a	12.0
3	b	6.0
4	b	7.5
5	b	8.0
6	c	3.0
7	c	5.0
8	c	6.0

In [148]:

```
#calculate means of each group
data.pivot_table(values='ounces',index='group',aggfunc=np.mean)
data.group
```

Out[148]:

```
0    a
1    a
2    a
3    b
4    b
5    b
6    c
7    c
8    c
Name: group, dtype: object
```

In [ ]:

```
#calculate count by each group
data.pivot_table(values='ounces',index='group',aggfunc='count')
```

In [146]:

```
data
```

Out[146]:

	group	ounces
0	a	4.0
1	a	3.0
2	a	12.0
3	b	6.0
4	b	7.5
5	b	8.0
6	c	3.0
7	c	5.0
8	c	6.0

Up till now, we've become familiar with the basics of pandas library using toy examples. Now, we'll take up a real-life data set and use our newly gained knowledge to explore it.

Exploring ML Data Set

We'll work with the popular adult data set.The data set has been taken from UCI Machine Learning Repository. You can download the data from here. In this data set, the dependent variable is "target." It is a binary classification problem. We need to predict if the salary of a given person is less than or more than 50K.

In [149]:

```
#load the data
train  = pd.read_csv("/root/Downloads/datafiles19cdaf8/train.csv")
test = pd.read_csv("/root/Downloads/datafiles19cdaf8/test.csv")
```

In [150]:

```
#check data set
train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32561 entries, 0 to 32560
Data columns (total 15 columns):
age               32561 non-null int64
workclass         30725 non-null object
fnlwgt            32561 non-null int64
education         32561 non-null object
education.num     32561 non-null int64
marital.status    32561 non-null object
occupation        30718 non-null object
relationship      32561 non-null object
race              32561 non-null object
sex               32561 non-null object
capital.gain      32561 non-null int64
capital.loss      32561 non-null int64
hours.per.week    32561 non-null int64
native.country    31978 non-null object
target            32561 non-null object
dtypes: int64(6), object(9)
memory usage: 3.7+ MB
```

We see that, the train data has 32561 rows and 15 columns. Out of these 15 columns, 6 have integers classes and the rest have object (or character) classes. Similarly, we can check for test data. An alternative way of quickly checking rows and columns is

In [152]:

```
print ("The train data has",train.shape)
print ("The test data has",test.shape)
```

```
The train data has (32561, 15)
The test data has (16281, 15)
```

In [153]:

```
#Let have a glimpse of the data set
train.head()
```

Out[153]:

	age	workclass	fnlwgt	education	education.num	marital.status	occupation	relationshi
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-famil
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husban
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-famil
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husban
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wif

Now, let's check the missing values (if present) in this data.

In [154]:

```
nans = train.shape[0] - train.dropna().shape[0]
print ("%d rows have missing values in the train data" %nans)
```

```
2399 rows have missing values in the train data
```

In [155]:

```
nand = test.shape[0] - test.dropna().shape[0]
print ("%d rows have missing values in the test data" %nand)
```

```
1221 rows have missing values in the test data
```

We should be more curious to know which columns have missing values.

In [156]:

```
#only 3 columns have missing values
train.isnull().sum()
```

Out[156]:

```
age                  0
workclass         1836
fnlwgt               0
education            0
education.num        0
marital.status       0
occupation        1843
relationship         0
race                 0
sex                  0
capital.gain         0
capital.loss         0
hours.per.week       0
native.country     583
target               0
dtype: int64
```

Let's count the number of unique values from character variables.

In [157]:

```
cat = train.select_dtypes(include=['O'])
cat.apply(pd.Series.nunique)
```

Out[157]:

```
workclass          8
education         16
marital.status     7
occupation        14
relationship       6
race               5
sex                2
native.country    41
target             2
dtype: int64
```

Since missing values are found in all 3 character variables, let's impute these missing values with their respective modes.

In [158]:

```
#Education
train.workclass.value_counts(sort=True)
train.workclass.fillna('Private',inplace=True)


#Occupation
train.occupation.value_counts(sort=True)
train.occupation.fillna('Prof-specialty',inplace=True)


#Native Country
train['native.country'].value_counts(sort=True)
train['native.country'].fillna('United-States',inplace=True)
```

Let's check again if there are any missing values left.

In [159]:

```
train.isnull().sum()
```

Out[159]:

```
age               0
workclass         0
fnlwgt            0
education         0
education.num     0
marital.status    0
occupation        0
relationship      0
race              0
sex               0
capital.gain      0
capital.loss      0
hours.per.week    0
native.country    0
target            0
dtype: int64
```

Now, we'll check the target variable to investigate if this data is imbalanced or not.

In [160]:

```
#check proportion of target variable
train.target.value_counts()/train.shape[0]
```

Out[160]:

```
 <=50K    0.75919
 >50K     0.24081
Name: target, dtype: float64
```

We see that 75% of the data set belongs to <=50K class. This means that even if we take a rough guess of target prediction as <=50K, we'll get 75% accuracy. Isn't that amazing? Let's create a cross tab of the target variable with education. With this, we'll try to understand the influence of education on the target variable.

In [161]:

```
pd.crosstab(train.education, train.target,margins=True)/train.shape[0]
```

Out[161]:

target education	<=50K	>50K	All
10th	0.026750	0.001904	0.028654
11th	0.034243	0.001843	0.036086
12th	0.012285	0.001013	0.013298
1st-4th	0.004975	0.000184	0.005160
5th-6th	0.009736	0.000491	0.010227
7th-8th	0.018611	0.001228	0.019840
9th	0.014957	0.000829	0.015786
Assoc-acdm	0.024631	0.008139	0.032769
Assoc-voc	0.031357	0.011087	0.042443
Bachelors	0.096250	0.068210	0.164461
Doctorate	0.003286	0.009398	0.012684
HS-grad	0.271060	0.051442	0.322502
Masters	0.023464	0.029452	0.052916
Preschool	0.001566	0.000000	0.001566
Prof-school	0.004699	0.012991	0.017690
Some-college	0.181321	0.042597	0.223918
All	0.759190	0.240810	1.000000

We see that out of 75% people with <=50K salary, 27% people are high school graduates, which is correct as people with lower levels of education are expected to earn less. On the other hand, out of 25% people with >=50K salary, 6% are bachelors and 5% are high-school grads. Now, this pattern seems to be a matter of concern. That's why we'll have to consider more variables before coming to a conclusion.

If you've come this far, you might be curious to get a taste of building your first machine learning model. In the coming week we'll share an exclusive tutorial on machine learning in python. However, let's get a taste of it here.

We'll use the famous and formidable scikit learn library. Scikit learn accepts data in numeric format. Now, we'll have to convert the character variable into numeric. We'll use the labelencoder function.

In label encoding, each unique value of a variable gets assigned a number, i.e., let's say a variable color has four values ['red','green','blue','pink'].

Label encoding this variable will return output as: red = 2 green = 0 blue = 1 pink = 3

In [162]:

```
#load sklearn and encode all object type variables
from sklearn import preprocessing

for x in train.columns:
    if train[x].dtype == 'object':
        lbl = preprocessing.LabelEncoder()
        lbl.fit(list(train[x].values))
        train[x] = lbl.transform(list(train[x].values))
```

Let's check the changes applied to the data set.

In [164]:

```
train.head()
```

Out[164]:

	age	workclass	fnlwgt	education	education.num	marital.status	occupation	relationshi
0	39	6	77516	9	13	4	0	
1	50	5	83311	9	13	2	3	
2	38	3	215646	11	9	0	5	
3	53	3	234721	1	7	2	5	
4	28	3	338409	9	13	2	9	

As we can see, all the variables have been converted to numeric, including the target variable.

In [165]:

```
#<50K = 0 and >50K = 1
train.target.value_counts()
```

Out[165]:

```
0    24720
1     7841
Name: target, dtype: int64
```

Building a Random Forest Model

Let's create a random forest model and check the model's accuracy.

In [189]:

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score


y = train['target']
del train['target']

X = train
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.3,random_state=
1,stratify=y)

#train the RF classifier
clf = RandomForestClassifier(n_estimators = 500, max_depth = 6)
clf.fit(X_train,y_train)


RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                max_depth=6, max_features='auto', max_leaf_nodes=None,
                min_impurity_split=1e-07, min_samples_leaf=1,
                min_samples_split=2, min_weight_fraction_leaf=0.0,
                n_estimators=500, n_jobs=1, oob_score=False, random_state=None,
                verbose=0, warm_start=False)


clf.predict(X_test)
```

Out[189]:

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

Now, let's make prediction on the test set and check the model's accuracy.

In [190]:

```
#make prediction and check model's accuracy
prediction = clf.predict(X_test)
acc =  accuracy_score(np.array(y_test),prediction)
print ('The accuracy of Random Forest is {}'.format(acc))
```

```
The accuracy of Random Forest is 0.852492578564848
```

Hurrah! Our learning algorithm gave 85% accuracy. Well, we can do tons of things on this data and improve the accuracy. We'll learn about it in future articles. What's next?

In this tutorial, we divided the train data into two halves and made prediction on the test data. As your exercise, you should use this model and make prediction on the test data we loaded initially. You can perform same set of steps we did on the train data to complete this exercise. In case you face any difficulty, feel free to share it in Comments below.

Summary

This tutorial is meant to help python developers or anyone who's starting with python to get a taste of data manipulation and a little bit of machine learning using python. I'm sure, by now you would be convinced that python is actually very powerful in handling and processing data sets. But, what we learned here is just the tip of the iceberg. Don't get complacent with this knowledge.

To dive deeper in pandas, check its documentation and start exploring. If you get stuck anywhere, you can drop your questions or suggestions in Comments below. Hope you found this tutorial useful.