Download this page as a jupyter notebook at Lab 11

Laboratory 11: Databases

Medrano, Giovanni

R11521018

ENGR 1330 Laboratory 11 - In Lab

In [1]:
```
# Preamble script block to identify host, user, and kernel
import sys
! hostname
! whoami
print(sys.executable)
print(sys.version)
print(sys.version_info)
```

```
DESKTOP-6HAS1BN
desktop-6has1bn\medra
C:\Users\medra\anaconda3\python.exe
3.8.5 (default, Sep  3 2020, 21:29:08) [MSC v.1916 64 bit (AMD64)]
sys.version_info(major=3, minor=8, micro=5, releaselevel='final', serial=0)
```

Pandas Cheat Sheet(s)

The Pandas library is a preferred tool for data scientists to perform data manipulation and analysis, next to matplotlib for data visualization and NumPy for scientific computing in Python.

The fast, flexible, and expressive Pandas data structures are designed to make real-world data analysis significantly easier, but this might not be immediately the case for those who are just getting started with it. Exactly because there is so much functionality built into this package that the options are overwhelming.

Hence summary sheets will be useful

- A summary sheet: https://pandas.pydata.org/Pandas_Cheat_Sheet.pdf
- A different one: http://datacamp-community-prod.s3.amazonaws.com/f04456d7-8e61-482f-9cc9-da6f7f25fc9b

Pandas

A data table is called a `DataFrame` in pandas (and other programming environments too).

The figure below from https://pandas.pydata.org/docs/getting_started/index.html illustrates a dataframe model:

Each column and each row in a dataframe is called a series, the header row, and index column are special.

To use pandas, we need to import the module, often pandas has numpy as a dependency so it also must be imported

In [2]:
```
import numpy
import pandas
```

Dataframe-structure using primative python

First lets construct a dataframe like object using python primatives. We will construct 3 lists, one for row names, one for column names, and one for the content.

In [3]:
```
mytabular = numpy.random.randint(1,100,(5,4))
myrowname = ['A','B','C','D','E']
mycolname = ['W','X','Y','Z']
mytable = [['' for jcol in range(len(mycolname)+1)] for irow in range(len(myrowname)+1)
print(mytable)
```

```
[['', '', '', '', ''], ['', '', '', '', ''], ['', '', '', '', ''], ['', '', '', '', ''],
['', '', '', '', ''], ['', '', '', '', '']]
```

The above builds a placeholder named `mytable` for the psuedo-dataframe. Next we populate the table, using a for loop to write the column names in the first row, row names in the first column, and the table fill for the rest of the table.

In [4]:
```
for irow in range(1,len(myrowname)+1): # write the row names
    mytable[irow][0]=myrowname[irow-1]
for jcol in range(1,len(mycolname)+1): # write the column names
    mytable[0][jcol]=mycolname[jcol-1]
for irow in range(1,len(myrowname)+1): # fill the table (note the nested loop)
    for jcol in range(1,len(mycolname)+1):
        mytable[irow][jcol]=mytabular[irow-1][jcol-1]
```

Now lets print the table out by row and we see we have a very dataframe-like structure

In [5]:
```
for irow in range(0,len(myrowname)+1):
    print(mytable[irow][0:len(mycolname)+1])
```

```
['', 'W', 'X', 'Y', 'Z']
['A', 52, 44, 94, 58]
['B', 90, 17, 19, 42]
['C', 31, 61, 53, 92]
['D', 60, 41, 21, 81]
['E', 63, 25, 44, 44]
```

We can also query by row

In [6]:
```
print(mytable[3][0:len(mycolname)+1])
```

```
['C', 31, 61, 53, 92]
```

Or by column

In [7]:
```
for irow in range(0,len(myrowname)+1):  #cannot use implied loop in a column slice
    print(mytable[irow][2])
```

```
X
44
17
61
41
25
```

Or by row+column index; sort of looks like a spreadsheet syntax.

In [8]:
```
print(' ',mytable[0][3])
print(mytable[3][0],mytable[3][3])
```

```
  Y
C 53
```

Create a proper dataframe

We will now do the same using pandas

In [9]:
```
df = pandas.DataFrame(numpy.random.randint(1,100,(5,4)), ['A','B','C','D','E'], ['W','X
df
```

Out[9]:

	W	X	Y	Z
A	14	18	58	98
B	86	41	39	12
C	56	49	92	10
D	81	73	15	97
E	97	32	45	71

We can also turn our table into a dataframe, notice how the constructor adds header row and index column

In [10]:
```
df1 = pandas.DataFrame(mytable)
df1
```

Out[10]:

	0	1	2	3	4
0		W	X	Y	Z
1	A	52	44	94	58
2	B	90	17	19	42
3	C	31	61	53	92
4	D	60	41	21	81
5	E	63	25	44	44

To get proper behavior, we can just reuse our original objects

In [11]:
```
df2 = pandas.DataFrame(mytabular,myrowname,mycolname)
df2
```

Out[11]:

	W	X	Y	Z
A	52	44	94	58
B	90	17	19	42
C	31	61	53	92
D	60	41	21	81
E	63	25	44	44

Getting the shape of dataframes

The shape method will return the row and column rank (count) of a dataframe.

In [12]:
```
df.shape
```

Out[12]: (5, 4)

In [13]:
```
df1.shape
```

Out[13]: (6, 5)

In [14]:
```
df2.shape
```

Out[14]: (5, 4)

Appending new columns

To append a column simply assign a value to a new column name to the dataframe

In [15]:
```
df['new']= 'NA'
df
```

Out[15]:

	W	X	Y	Z	new
A	14	18	58	98	NA
B	86	41	39	12	NA
C	56	49	92	10	NA
D	81	73	15	97	NA
E	97	32	45	71	NA

Appending new rows

A bit trickier but we can create a copy of a row and concatenate it back into the dataframe.

In [16]:
```
newrow = df.loc[['E']].rename(index={"E": "X"}) # create a single row, rename the index
newtable = pandas.concat([df,newrow]) # concatenate the row to bottom of df - note the
```

In [17]:
```
newtable
```

Out[17]:

	W	X	Y	Z	new
A	14	18	58	98	NA
B	86	41	39	12	NA
C	56	49	92	10	NA
D	81	73	15	97	NA
E	97	32	45	71	NA
X	97	32	45	71	NA

Removing Rows and Columns

To remove a column is straightforward, we use the drop method

In [18]:
```
newtable.drop('new', axis=1, inplace = True)
newtable
```

Out[18]:

	W	X	Y	Z
A	14	18	58	98
B	86	41	39	12
C	56	49	92	10
D	81	73	15	97
E	97	32	45	71
X	97	32	45	71

To remove a row, you really got to want to, easiest is probablty to create a new dataframe with the row removed

In [19]:
```
newtable = newtable.loc[['A','B','D','E','X']] # select all rows except C
newtable
```

Out[19]:

	W	X	Y	Z
A	14	18	58	98
B	86	41	39	12
D	81	73	15	97
E	97	32	45	71
X	97	32	45	71

Indexing

We have already been indexing, but a few examples follow:

```
newtable['X'] #Selecing a single column
```

In [20]:

Out[20]:
```
A    18
B    41
D    73
E    32
X    32
Name: X, dtype: int32
```

In [21]:
```
newtable[['X','W']] #Selecing a multiple columns
```

Out[21]:

	X	W
A	18	14
B	41	86
D	73	81
E	32	97
X	32	97

In [22]:
```
newtable.loc['E'] #Selecing rows based on label via loc[ ] indexer
```

Out[22]:
```
W    97
X    32
Y    45
Z    71
Name: E, dtype: int32
```

In [23]:
```
newtable.loc[['E','X','B']] #Selecing multiple rows based on label via loc[ ] indexer
```

Out[23]:

	W	X	Y	Z
E	97	32	45	71
X	97	32	45	71
B	86	41	39	12

In [24]:
```
newtable.loc[['B','E','D'],['X','Y']] #Selecting elemens via both rows and columns via
```

Out[24]:

	X	Y
B	41	39
E	32	45
D	73	15

Conditional Selection

In [25]:
```
df = pandas.DataFrame({'col1':[1,2,3,4,5,6,7,8],
                   'col2':[444,555,666,444,666,111,222,222],
                   'col3':['orange','apple','grape','mango','jackfruit','watermelon','b
df
```

Out[25]:

	col1	col2	col3
0	1	444	orange
1	2	555	apple
2	3	666	grape
3	4	444	mango
4	5	666	jackfruit
5	6	111	watermelon
6	7	222	banana
7	8	222	peach

In [26]:

```
#What fruit corresponds to the number 555 in ‘col2’?

df[df['col2']==555]['col3']
```

Out[26]:

```
1    apple
Name: col3, dtype: object
```

In [27]:

```
#What fruit corresponds to the minimum number in ‘col2’?

df[df['col2']==df['col2'].min()]['col3']
```

Out[27]:

```
5    watermelon
Name: col3, dtype: object
```

Descriptor Functions

In [28]:

```
#Creating a dataframe from a dictionary

df = pandas.DataFrame({'col1':[1,2,3,4,5,6,7,8],
                       'col2':[444,555,666,444,666,111,222,222],
                       'col3':['orange','apple','grape','mango','jackfruit','watermelon','b
df
```

Out[28]:

	col1	col2	col3
0	1	444	orange
1	2	555	apple
2	3	666	grape
3	4	444	mango
4	5	666	jackfruit
5	6	111	watermelon
6	7	222	banana
7	8	222	peach

`head` method

Returns the first few rows, useful to infer structure

In [29]:
```
#Returns only the first five rows

df.head()
```

Out[29]:

	col1	col2	col3
0	1	444	orange
1	2	555	apple
2	3	666	grape
3	4	444	mango
4	5	666	jackfruit

`info` method

Returns the data model (data column count, names, data types)

In [30]:
```
#Info about the dataframe

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8 entries, 0 to 7
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   col1    8 non-null      int64
 1   col2    8 non-null      int64
 2   col3    8 non-null      object
dtypes: int64(2), object(1)
memory usage: 320.0+ bytes
```

`describe` method

Returns summary statistics of each numeric column.

Also returns the minimum and maximum value in each column, and the IQR (Interquartile Range).

Again useful to understand structure of the columns.

In [31]:
```
#Statistics of the dataframe

df.describe()
```

Out[31]:

	col1	col2
count	8.00000	8.0000
mean	4.50000	416.2500
std	2.44949	211.8576
min	1.00000	111.0000
25%	2.75000	222.0000

	col1	col2
50%	4.50000	444.0000
75%	6.25000	582.7500
max	8.00000	666.0000

Counting and Sum methods

There are also methods for counts and sums by specific columns

In [32]:
```
df['col2'].product() #product of a specified column
```
Out[32]:
```
7228334039530122496
```

In [33]:
```
df['col2'].sum() #Sum of a specified column
```
Out[33]:
```
3330
```

The `unique` method returns a list of unique values (filters out duplicates in the list, underlying dataframe is preserved)

In [34]:
```
df['col2'].unique() #Returns the list of unique values along the indexed column
```
Out[34]:
```
array([444, 555, 666, 111, 222], dtype=int64)
```

The `nunique` method returns a count of unique values

In [35]:
```
df['col2'].nunique() #Returns the total number of unique values along the indexed colum
```
Out[35]:
```
5
```

The `value_counts()` method returns the count of each unique value (kind of like a histogram, but each value is the bin)

In [36]:
```
df['col2'].value_counts()  #Returns the number of occurences of each unique value
```
Out[36]:
```
222    2
444    2
666    2
111    1
555    1
Name: col2, dtype: int64
```

Using functions in dataframes - symbolic apply

The power of pandas is an ability to apply a function to each element of a dataframe series (or a whole frame) by a technique called symbolic (or synthetic programming) application of the function.

Its pretty complicated but quite handy, best shown by an example

In [37]:
```
def times2(x):  # A prototype function to scalar multiply an object x by 2
    return(x//2)
```

```
print(df)
print('Apply the times2 function to col2')
df['col2'].apply(times2) #Symbolic apply the function to each element of column col2, r
```

```
   col1  col2       col3
0     1   444     orange
1     2   555      apple
2     3   666      grape
3     4   444      mango
4     5   666  jackfruit
5     6   111 watermelon
6     7   222     banana
7     8   222      peach
Apply the times2 function to col2
```

Out[37]:
```
0    222
1    277
2    333
3    222
4    333
5     55
6    111
7    111
Name: col2, dtype: int64
```

Sorts

In [38]:
```
df.sort_values('col2', ascending = True) #Sorting based on columns
```

Out[38]:

	col1	col2	col3
5	6	111	watermelon
6	7	222	banana
7	8	222	peach
0	1	444	orange
3	4	444	mango
1	2	555	apple
2	3	666	grape
4	5	666	jackfruit

Exercise 1

Create a prototype function to compute the cube root of a numeric object (literally two lines to define the function), recall exponentation is available in primative python.

Apply your function to column **'X'** of dataframe `newtable` created above

In [41]:
```
# Define your function here:

def upperCase(z):
    val = z[0].upper() + z[1:]
    return val
df["col3"].apply(upperCase)
```

Out[41]:
```
0       Orange
1        Apple
2        Grape
3        Mango
4    Jackfruit
5   Watermelon
6       Banana
7        Peach
Name: col3, dtype: object
```

Aggregating (Grouping Values) dataframe contents

In [42]:
```
#Creating a dataframe from a dictionary

data = {
    'key' : ['A', 'B', 'C', 'A', 'B', 'C'],
    'data1' : [1, 2, 3, 4, 5, 6],
    'data2' : [10, 11, 12, 13, 14, 15],
    'data3' : [20, 21, 22, 13, 24, 25]
}

df1 = pandas.DataFrame(data)
df1
```

Out[42]:

	key	data1	data2	data3
0	A	1	10	20
1	B	2	11	21
2	C	3	12	22
3	A	4	13	13
4	B	5	14	24
5	C	6	15	25

In [43]:
```
# Grouping and summing values in all the columns based on the column 'key'

df1.groupby('key').sum()
```

Out[43]:

	data1	data2	data3
key			
A	5	23	33
B	7	25	45
C	9	27	47

In [44]:
```
# Grouping and summing values in the selected columns based on the column 'key'

df1.groupby('key')[['data1', 'data2']].sum()
```

Out[44]:

	data1	data2

	key	data1	data2
key			
A		5	23
B		7	25
C		9	27

Filtering out missing values

In [45]:
```
#Creating a dataframe from a dictionary

df = pandas.DataFrame({'col1':[1,2,3,4,None,6,7,None],
                       'col2':[444,555,None,444,666,111,None,222],
                       'col3':['orange','apple','grape','mango','jackfruit','watermelon','b
df
```

Out[45]:

	col1	col2	col3
0	1.0	444.0	orange
1	2.0	555.0	apple
2	3.0	NaN	grape
3	4.0	444.0	mango
4	NaN	666.0	jackfruit
5	6.0	111.0	watermelon
6	7.0	NaN	banana
7	NaN	222.0	peach

Below we drop any row that contains a `NaN` code.

In [46]:
```
df_dropped = df.dropna()
df_dropped
```

Out[46]:

	col1	col2	col3
0	1.0	444.0	orange
1	2.0	555.0	apple
3	4.0	444.0	mango
5	6.0	111.0	watermelon

Below we replace `NaN` codes with some value, in this case 0

In [47]:
```
df_filled1 = df.fillna(0)
df_filled1
```

Out[47]:

	col1	col2	col3

	col1	col2	col3
0	1.0	444.0	orange
1	2.0	555.0	apple
2	3.0	0.0	grape
3	4.0	444.0	mango
4	0.0	666.0	jackfruit
5	6.0	111.0	watermelon
6	7.0	0.0	banana
7	0.0	222.0	peach

Below we replace `NaN` codes with some value, in this case the mean value of of the column in which the missing value code resides.

In [48]:
```
df_filled2 = df.fillna(df.mean())
df_filled2
```

Out[48]:

	col1	col2	col3
0	1.000000	444.0	orange
1	2.000000	555.0	apple
2	3.000000	407.0	grape
3	4.000000	444.0	mango
4	3.833333	666.0	jackfruit
5	6.000000	111.0	watermelon
6	7.000000	407.0	banana
7	3.833333	222.0	peach

Exercise 2

Replace the **'NaN'** codes with the string 'missing' in dataframe **'df'**

In [49]:
```
# Replace the NaN with the string 'missing' here:

df_filled2 = df.fillna("missing")
df_filled2
```

Out[49]:

	col1	col2	col3
0	1	444	orange
1	2	555	apple
2	3	missing	grape
3	4	444	mango

	col1	col2	col3
4	missing	666	jackfruit
5	6	111	watermelon
6	7	missing	banana
7	missing	222	peach

```
In [ ]:
```