# PANDAS

The most basic of an Pandas object is the Series, which is akin to a one-dimensional array.

```
In [4]: obj = pd.Series([4, 7, -5, 3])
In [5]: obj
Out[5]:
0 4
1 7
2 -5
3 3
```

A series object has both index and values.

```
In [145]: obj.index
Out[145]: Int64Index([0, 1, 2, 3], dtype='int64')

In [146]: obj.values
Out[146]: array([ 4,  7, -5,  3])
```

Moving to the DataFrame object.

You could think of it as 2-d array. But while arrays hold data of one type only, the dataframe can hold data of multiple types.

For small dataframes, we could a dictionary data structure to construct a dataframe.

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
 'year': [2000, 2001, 2002, 2001, 2002],
 'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
frame = pd.DataFrame(data)

In [38]: frame
Out[38]:
 pop state year
0 1.5 Ohio 2000
1 1.7 Ohio 2001
2 3.6 Ohio 2002
3 2.4 Nevada 2001
4 2.9 Nevada 2002
```

```
In [40]: frame2 = pgDataFrame(data, columns=['year', 'state', 'pop', 'debt'],
 ....: index=['one', 'two', 'three', 'four', 'five'])
In [41]: frame2
Out[41]:
 year state pop debt
one 2000 Ohio 1.5 NaN
two 2001 Ohio 1.7 NaN
three 2002 Ohio 3.6 NaN
four 2001 Nevada 2.4 NaN
five 2002 Nevada 2.9 NaN
```

If we don’t specify what goes into a column, pandas automatically fills the column with Nan.

We can access the columns.

```
In [43]: frame2['state']

In [44]: frame2.year
```

And we can access rows.

```
In [45]: frame2.ix['three']
Out[45]:
year 2002
state Ohio
pop 3.6
debt NaN
Name: three

Also you use matlab style indexing to access rows and columns

frame2[1:], frame[1:3], ...
```

# Data comes in variety of formats

csv, tsv, excel, json, xml ...

A non-trivial task for the intrepid data scientist is to format raw data into a form that can be nicely accessed by standard readers. Consider a snippet of the access log file:

```
cran.stat.ucla.edu 164.67.41.11 - - [20/Aug/2009:00:00:00 -0700] "GET
/bin/macosx/universal/contrib/r-release/rpart_3.1-45.tgz HTTP/1.0" 304 - "-" "UCLAseek"
cran.stat.ucla.edu 164.67.41.11 - - [20/Aug/2009:00:00:00 -0700] "GET
/bin/windows/contrib/r-release/rpart_3.1-45.zip HTTP/1.0" 304 - "-" "UCLAseek"
```

You can’t easily cut on spaces,even though it’s easy to see by the eye that the field are delineated by spaces.  We’ll spend hours trying to massage that file into something that can’t be easily read.

```
In [35]: import pandas as pd

In [36]: usda = pd.read_csv("USDA.csv")
```

The read_csv method assumes by default that the first line is the header. There are options to change that default.

```
In [6]: usda.head()
```

Out[6]:

| | ID | Description | Calories | Protein | TotalFat | Carbohydrate | Sodium | SaturatedFat | Cholesterol | Sugar | Calcium | Iron | Potassium | VitaminC | VitaminE | VitaminD |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1001 | BUTTER,WITH SALT | 717 | 0.85 | 81.11 | 0.06 | 714 | 51.368 | 215 | 0.06 | 24 | 0.02 | 24 | 0 | 2.32 | 1.5 |
| 1 | 1002 | BUTTER,WHIPPED,WITH SALT | 717 | 0.85 | 81.11 | 0.06 | 827 | 50.489 | 219 | 0.06 | 24 | 0.16 | 26 | 0 | 2.32 | 1.5 |
| 2 | 1003 | BUTTER OIL,ANHYDROUS | 876 | 0.28 | 99.48 | 0.00 | 2 | 61.924 | 256 | 0.00 | 4 | 0.00 | 5 | 0 | 2.80 | 1.8 |
| 3 | 1004 | CHEESE,BLUE | 353 | 21.40 | 28.74 | 2.34 | 1395 | 18.669 | 75 | 0.50 | 528 | 0.31 | 256 | 0 | 0.25 | 0.5 |
| 4 | 1005 | CHEESE,BRICK | 371 | 23.24 | 29.68 | 2.79 | 560 | 18.764 | 94 | 0.51 | 674 | 0.43 | 136 | 0 | 0.26 | 0.5 |

```
In [8]: usda.describe()
Out[8]:
                ID     Calories      Protein     TotalFat  Carbohydrate  \
count   7058.000000  7057.000000  7057.000000  7057.000000   7057.000000
mean   14259.821196   219.695338    11.710368    10.320614     20.697860
std     8577.179705   172.198755    10.919356    16.814191     27.630443
min     1001.000000     0.000000     0.000000     0.000000      0.000000
25%     8387.250000    85.000000     2.290000     0.720000      0.000000
50%    13293.500000   181.000000     8.200000     4.370000      7.130000
75%    18336.750000   331.000000    20.430000    12.700000     28.170000
max    93600.000000   902.000000    88.320000   100.000000    100.000000

             Sodium  SaturatedFat  Cholesterol        Sugar      Calcium  \
count   6974.000000   6757.000000  6770.000000  5148.000000  6922.000000
mean     322.059220      3.452267    41.551994     8.256540    73.530627
std     1045.416931      6.921267   122.963028    15.361509   222.445338
min        0.000000      0.000000     0.000000     0.000000     0.000000
25%       37.000000      0.172000     0.000000     0.000000     9.000000
50%       79.000000      1.256000     3.000000     1.395000    19.000000
75%      386.000000      4.028000    69.000000     7.875000    56.000000
max    38758.000000     95.600000  3100.000000    99.800000  7364.000000

              Iron     Potassium     VitaminC     VitaminE     VitaminD
count  6935.000000   6649.000000  6726.000000  4338.000000  4224.000000
mean      2.828368    301.357949     9.435980     1.487462     0.576918
std       6.019878    415.638949    71.256536     5.386914     4.301147
min       0.000000      0.000000     0.000000     0.000000     0.000000
25%       0.520000    135.000000     0.000000     0.120000     0.000000
50%       1.330000    250.000000     0.000000     0.270000     0.000000
75%       2.620000    348.000000     3.100000     0.710000     0.100000
max     123.600000  16500.000000  2400.000000   149.400000   250.000000
```

```
In [11]: usda.dtypes
Out[11]:
ID                int64
Description      object
Calories        float64
Protein         float64
TotalFat        float64
Carbohydrate    float64
Sodium          float64
SaturatedFat    float64
Cholesterol     float64
Sugar           float64
Calcium         float64
Iron            float64
Potassium       float64
VitaminC        float64
VitaminE        float64
VitaminD        float64
dtype: object
```

We can check the header of the dataframe.

```
In [41]: usda.columns
Out[41]:
Index([ID, Description, Calories, Protein, TotalFat, Carbohydrate, Sodium,
       SaturatedFat, Cholesterol, Sugar, Calcium, Iron, Potassium,
       VitaminC, VitaminE, VitaminD], dtype=object)

Access columns by name

In [13]: usda.Sodium.head()
Out[13]:
0     714
1     827
2       2
3    1395
4     560
Name: Sodium, dtype: float64
```

```
In [14]: usda['Calcium'].tail()
Out[14]:
7053     18
7054     66
7055     10
7056     10
7057    118
Name: Calcium, dtype: float64
```

Columns, e.g. usda[‘Protein’], are like Series Objects. Whatever methods you can perform on a Series, you can pretty much used on a column or on a row for that matter.

Missing data is something you also always be worried about.

```
In [157]: usda['Calcium'].notnull()
Out[157]:
0      True
1      True
2      True
…
```

```
In [156]: usda['Calcium'].isnull()
Out[156]:
0     False
1     False
2     False
...
```

In [158]: usda['Calcium'].isnull().value_counts()
Out[158]:
False  6922
True    136
dtype: int64

Max values across columns, also works with mean, std,counts,var...

```
In [15]: usda.max()
Out[15]:
ID                93600
Description     ZWIEBACK
Calories             902
Protein            88.32
TotalFat             100
Carbohydrate         100
Sodium             38758
SaturatedFat        95.6
Cholesterol         3100
Sugar               99.8
Calcium             7364
Iron               123.6
Potassium          16500
VitaminC            2400
VitaminE           149.4
VitaminD             250
dtype: object
```

Which food has the max salt content?

```
In [20]: usda.Sodium.idxmax()
Out[20]: 264

In [21]: usda['Description'][264]
Out[21]: 'SALT,TABLE'
```

Or Calcium content?

```
In [23]: usda.Description[usda.Calcium.argmax()]
Out[23]: 'LEAVENING AGENTS,BAKING PDR,DOUBLE-ACTING,STRAIGHT PO4'
```

idxmax gives back to row index, while argmax give the actual position index.

We construct a smaller dataframe from the larger dataframe based on certain criteria

```
HighSodium=usda[usda['Sodium']>10000]

In [29]: len(HighSodium)

Out[29]: 10

In [31]: HighSodium.shape

Out[31]: (10, 16)


In [149]: HighSodium.Sodium

Out[149]:

264     38758

921     26000

922     24000

924     23875

925     24000

937     11588

1302    17152

5320    10600

5323    27360

5697    26050

Name: Sodium, dtype: float64
```

```
In [32]: HighSodium.Description
Out[32]:
264                                                 SALT,TABLE
921                        SOUP,BF BROTH OR BOUILLON,PDR,DRY
922                                SOUP,BEEF BROTH,CUBED,DRY
924                         SOUP,CHICK BROTH OR BOUILLON,DRY
925                               SOUP,CHICK BROTH CUBES,DRY
937                                        GRAVY,AU JUS,DRY
1302                                           ADOBO FRESCO
5320    LEAVENING AGENTS,BAKING PDR,DOUBLE-ACTING,NA A...
5323                           LEAVENING AGENTS,BAKING SODA
5697                         DESSERTS,RENNIN,TABLETS,UNSWTND
Name: Description, dtype: object
```

We add new columns to dataframes

```
usda['lohi']= usda['Sodium']>10000

ZeroSugar = usda['Sugar']==0
```

Pandas has a version of the apply function in R

```
In [72]: %paste
def f(x):
    if x ==0: return "no sodium"
    elif x>0 and x<79:
        return "low sodium"
    elif x>79 and x<386:
        return "med sodium"
    else: return "high sodium"
In [73]: usda.Sodium.apply(f)
Out[73]:
0     high sodium
1     high sodium
2      low sodium
3     high sodium
```

**f can be any function that can do just about anything ….**

```
f = lambda x: '%.2f' % x
In [43]: f2 = lambda x: 'Max: %.3f, Min: %.3f' % (x.max(), x.min())

In [44]: f3 = lambda x: (x.max(), x.min())

In [45]: f4 = lambda x: Series([x.max(), x.min()], index=['max', 'min'])

f = lambda x: x.max() - x.min()
```

In this case, I used apply to categorise slices of dataset into no, low, med, high sodium foods

## The Groupby function is awesome.

```
In [74]: usda['cate_sodium']=usda.Sodium.apply(f)
In [77]: usda.cate_sodium.value_counts()
Out[77]:
low sodium     3334
high sodium    1861
med sodium     1715
no sodium       148
dtype: int64
In [75]: woot=usda.groupby('cate_sodium')
```

## Woot by itself doesn't seem to do much …

```
In [79]: woot
Out[79]: <pandas.core.groupby.DataFrameGroupBy object at 0x3f82f50>
```

But ...

```
In [76]: woot.mean()
Out[76]:
                        ID    Calories    Protein   TotalFat  Carbohydrate  \
cate_sodium
high sodium  14250.334766  264.222581  11.248935  11.934167     28.792828
low sodium   14556.291842  190.805039  13.162298   8.413023     16.046515
med sodium   14090.692128  191.790087  10.303924   7.382548     21.928035
no sodium     9660.344595  634.270270   1.099459  67.060338      9.489730
```

| cate_sodium | Sodium | SaturatedFat | Cholesterol | Sugar | Calcium | \ |
|---|---|---|---|---|---|---|
| high sodium | 987.629150 | 3.540177 | 31.111047 | 9.160616 | 120.048104 | |
| low sodium | 36.393221 | 3.181010 | 40.894294 | 6.594942 | 52.941230 | |
| med sodium | 215.562099 | 2.533783 | 54.252962 | 10.632438 | 70.545831 | |
| no sodium | 0.000000 | 19.464950 | 32.912409 | 4.533727 | 13.027397 | |

| cate_sodium | Iron | Potassium | VitaminC | VitaminE | VitaminD | lohi |
|---|---|---|---|---|---|---|
| high sodium | 4.218927 | 272.857992 | 6.504529 | 1.977789 | 0.967218 | 0.005373 |
| low sodium | 2.383719 | 321.191626 | 10.024868 | 0.731172 | 0.287794 | 0.000000 |
| med sodium | 2.428721 | 313.042608 | 10.322524 | 1.486775 | 0.542636 | 0.000000 |
| no sodium | 0.694514 | 59.772414 | 20.881560 | 10.985217 | 3.418072 | 0.000000 |

We have now the mean of each column delineated by the sodium classification.

```
In [78]: woot.count()
Out[78]:
                ID  Description  Calories  Protein  TotalFat  Carbohydrate  \
cate_sodium
high sodium   1861         1861      1860     1860      1860          1860
low sodium    3334         3334      3334     3334      3334          3334
med sodium    1715         1715      1715     1715      1715          1715
no sodium      148          148       148      148       148           148

              Sodium  SaturatedFat  Cholesterol  Sugar  Calcium  Iron  \
cate_sodium
high sodium     1777          1767         1738   1460     1767  1771
low sodium      3334          3179         3207   2331     3318  3315
med sodium      1715          1671         1688   1247     1691  1705
no sodium        148           140          137    110      146   144
```

**Other useful functions for the groupby object:** count, sum, mean, median, std, var, min, max, prod, first, last.

There is also multistep grouping

```
In [125]: usda.groupby(['cate_sodium','ZeroSugar'])
Out[125]: <pandas.core.groupby.DataFrameGroupBy object at 0x5ae2d10>

In [126]: yeah=usda.groupby(['cate_sodium','ZeroSugar'])
In [127]: yeah.max()
```

```
In [127]: yeah.max()
Out[127]:                          Calories  Protein  TotalFat  Carbohydrate  Sodium  \
cate_sodium ZeroSugar
high sodium False           900    87.75    100.00         95.40   26050
            True            889    82.60     98.59         86.85   38758
low sodium  False           878    86.00     99.01        100.00      78
            True            898    88.32    100.00         94.80      78
med sodium  False           716    55.30     76.08         97.57     385
            True            897    85.60     99.50         83.33     385
no sodium   False           902    15.03    100.00         98.60       0
            True            902    20.00    100.00         88.79       0
```

You could also use the pivot table function. Read more to how use it to make nifty classifications.

```
In [130]: usda.pivot_table(['Protein', 'Carbohydrate'], rows=['cate_sodium','ZeroSugar'])
Out[130]:
                           Carbohydrate    Protein
cate_sodium ZeroSugar
high sodium False          32.645858  10.238215
            True            4.206825  17.698294
low sodium  False          22.716289   8.739301
            True            1.126942  23.056068
med sodium  False          26.246927   7.677356
            True            0.967406  23.051297
no sodium   False          18.987727   2.160758
            True            1.845000   0.245244
```

**Let’s take a look at the csun student data.**

Copy the files in your home directory.

studat = pd.read_csv('student_course.dat',delimiter="|")

```
In [5]: studat.head()
Out[5]:
   1_Student_Id  2_Term_Id 3_Campus_Cd  4_College_Cd        5_Department_Cd  \
0     775745845       2003       NORTH            26             Journalism
1     775745845       2003       NORTH            31             Psychology
2     775745845       2003       NORTH            47             Philosophy
3     775745845       2003       NORTH            76  Physics and Astronomy
4     775745845       2003       NORTH            92            Kinesiology

  6_Include_In_Gpa_Ind  7_Course_Credits 8_Course_Cd 9_Course_Ref_No  \
0                    Y                 3     JOUR372          11052
1                    Y                 3      PSY150          95015
2                    N                 4     PHIL230          73085
3                    Y                 3     ASTR152          86013
4                    Y                 1     KIN126A          36076
```

| 10_Registered_Ind | ... | 20_Final_Grade | 21_Final_Grade_Date | \ |
|---|---|---|---|---|
| 0 | Y | ... | B+ | 20000603 |
| 1 | Y | ... | B+ | 20000603 |
| 2 | Y | ... | W | 20000503 |
| 3 | Y | ... | B- | 20000603 |
| 4 | Y | ... | A | 20000530 |

| 22_Final_Grade_Official_Ind | 23_Quality_Points | 24_Null | \ |
|---|---|---|---|
| 0 | Y | 9.9 | NaN |
| 1 | Y | 9.9 | NaN |
| 2 | Y | 0.0 | NaN |
| 3 | Y | 8.1 | NaN |
| 4 | Y | 4.0 | NaN |

| 25_Transfer_Course_Ind | 26_In_Progress_Course_Ind | 27_Course_Type_Cd | \ |
|---|---|---|---|
| 0 | N | N | LEC |
| 1 | N | N | LEC |
| 2 | N | N | LEC |
| 3 | N | N | LEC |
| 4 | N | N | ACT |

| 28_Ext_Course_Provider | 29_Delivery_Method_Cd |
|---|---|
| 0 | NaN | In Person |
| 1 | NaN | In Person |
| 2 | NaN | In Person |
| 3 | NaN | In Person |
| 4 | NaN | In Person |

**Resources for further learning.**

Python for data analysis. That’s a pretty good book for basic data analysis. The book also has a good collection of datasets that are very intriguing.

http://pandas.pydata.org/pandas-docs/stable/10min.html

http://byumcl.bitbucket.org/bootcamp2013/labs/pandas.html#

```
In [6]: studat.columns
Out[6]: Index([u'1_Student_Id', u'2_Term_Id', u'3_Campus_Cd', u'4_College_Cd',
u'5_Department_Cd', u'6_Include_In_Gpa_Ind', u'7_Course_Credits', u'8_Course_Cd',
u'9_Course_Ref_No', u'10_Registered_Ind', u'11_Registration_Status_Cd',
u'12_Registration_Status_Date', u'13_Null', u'14_Registered_Credits', u'15_Earned_Credits',
u'16_Null', u'17_Null', u'18_Gradable_Ind', u'19_Midterm_Grade', u'20_Final_Grade',
u'21_Final_Grade_Date', u'22_Final_Grade_Official_Ind', u'23_Quality_Points', u'24_Null',
u'25_Transfer_Course_Ind', u'26_In_Progress_Course_Ind', u'27_Course_Type_Cd',
u'28_Ext_Course_Provider', u'29_Delivery_Method_Cd'], dtype='object')

In [47]: studat['gpa'] = studat['23_Quality_Points']/studat['7_Course_Credits']

In [49]: studat['gpa'].head()
Out[49]:
0    3.3
1    3.3
2    0.0
3    2.7
4    4.0
Name: gpa, dtype: float64
```

```
In [51]: studat.pivot_table('gpa', rows='2_Term_Id', aggfunc=mean)
Out[51]:
2_Term_Id
597          2.448276
603          2.148148
607          1.500000
613          2.000000
617          3.302326
623          3.000000
627          2.000000
633          2.600000
637          2.842857
643          1.571429
647          3.000000
653          2.000000
657          2.000000
```