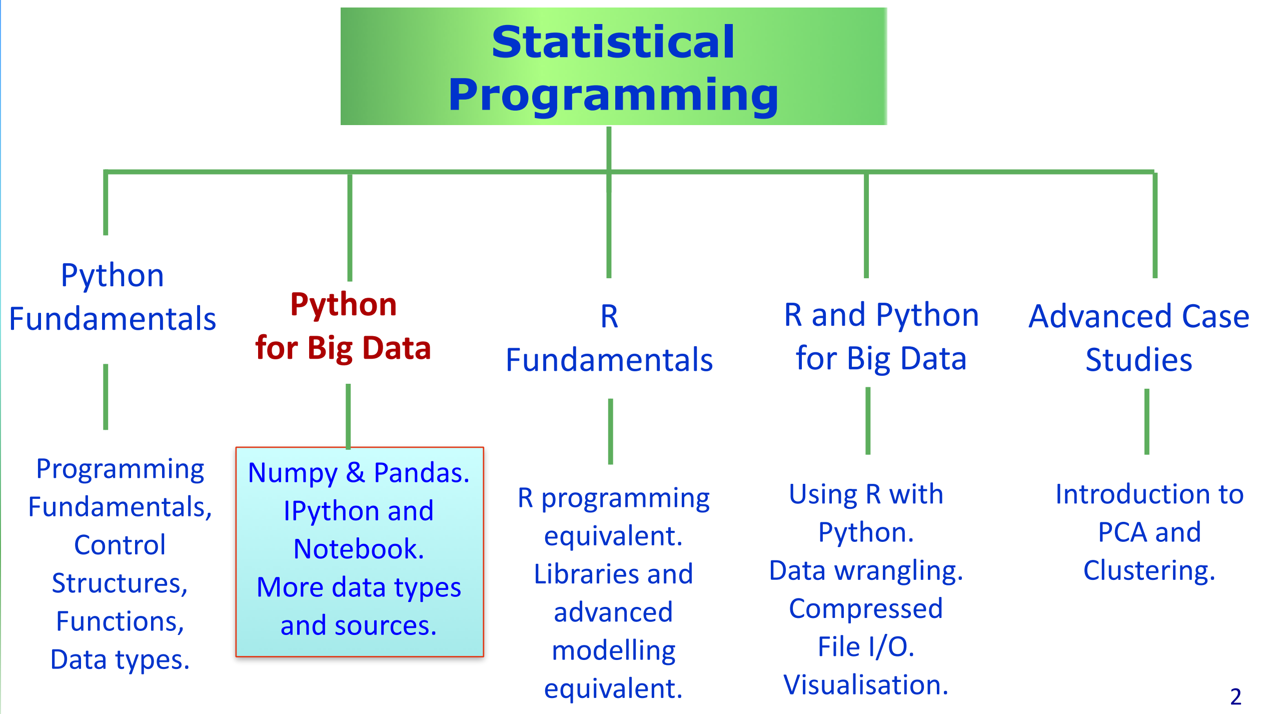
COMP 5070

Statistical Programming
for
Data Science

Python for Big Data: pandas

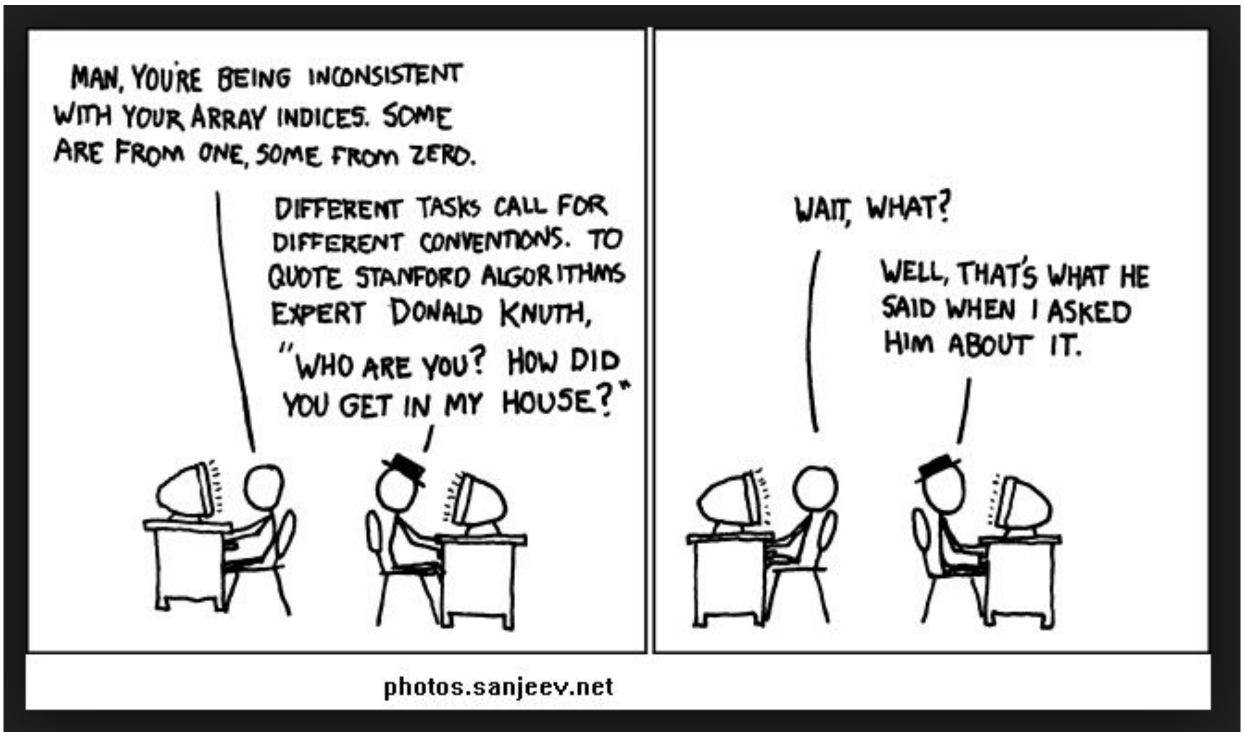
University of South Australia

The course at a glance



Statistical Programming
Python Fundamentals
Programming Fundamentals, Control Structures, Functions, Data types.
Python for Big Data
Numpy & Pandas. IPython and Notebook. More data types and sources.
R Fundamentals
R programming equivalent. Libraries and advanced modelling equivalent.
R and Python for Big Data
Using R with Python. Data wrangling. Compressed File I/O. Visualisation.
Advanced Case Studies
Introduction to PCA and Clustering.

The issue with pandas …


MAN, YOU'RE BEING INCONSISTENT WITH YOUR ARRAY INDICES. SOME ARE FROM ONE, SOME FROM ZERO.
DIFFERENT TASKS CALL FOR DIFFERENT CONVENTIONS. TO QUOTE STANFORD ALGORITHMS EXPERT DONALD KNUTH, "WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"
WAIT, WHAT?
WELL, THAT'S WHAT HE SAID WHEN I ASKED HIM ABOUT IT.
photos.sanjeev.net

First (and only) up … pandas

- The answer to R?
- New (and rich) data structures:
  - Series
  - DataFrame
- Indexing:
  - Regular
  - Hierarchical, Partial
- Immutability and Reindexing
- Function mapping
  - apply, applymap
- Missing data

"I don't know much about cloud computing, but I think it might be responsible for the strange weather we're having."

pandas in a Nutshell

- "R for Python"
- Provides easy to use data structures and many useful helper functions for data cleanup and transformations.
- Fast! (backed by numpy arrays)
- Contains high-level data structures and manipulation tools including structured or tabular data.
- Provides a rich, high-level interface making most common data tasks very concise and simple.
- Provides domain-specific functionality, e.g. time series manipulation and easy handling of missing data (not present in NumPy).

pandas in a Nutshell

- A clean axis indexing design to support fast data alignment, lookups, hierarchical indexing, and more high-performance data structures.
- Functionality includes:
  - Series/TimeSeries: 1D labelled vector
  - DataFrame: 2D spreadsheet-like structure
  - Panel: 3D labeled array, collection of DataFrames
- SQL-like functionality: GroupBy, joining/merging, etc. Missing data handling.
- Etymology: *pan*el *da*ta *s*tructures

Indexing: The pandas Killer Feature

- Each axis has an index.
- Automatic alignment between differently-indexed objects: makes it nearly impossible to accidentally combine misaligned data
- Hierarchical indexing provides an intuitive way of structuring and working with higher-dimensional data
- Natural way of expressing "group by" and join-type operations
- Claim: considered to be a better integrated and more flexible indexing than anything available in R or MATLAB

Pandas Data Structures

- The development of pandas introduces two new data structures to Python:

  1. Series
  2. DataFrame

- Both are built on top of NumPy (so it's fast!).
- Usually import pandas with the shorthand pd, i.e.

```
import pandas as pd
```

The Series Data Structure

- A Series is a one-dimensional object similar to an array, list, or column in a table. It is a mutable structure.
- Series allows for a labelled index to be assigned to each item.
- As with the List structure, by default, each item in a Series will receive an index label starting from 0.
- The simplest Series is formed from only an array of data:

```
In [3]: obj = pd.Series([4, 7, -5, 3])
Out[3]:
0     4
1     7
2    -5
3     3
dtype: int64
```

Index is shown on the left,
values on the right!

Series – extracting indices and value

- Can extract the indices and values using `obj.index` and `obj.values` respectively, i.e.

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

In [5]:obj.index
Out[5]: Int64Index([0, 1, 2, 3], dtype='int64')
```

- Can also specify the desired indices:

```
In [8]: obj2=pd.Series([4, 7, -5, 3],index=['d', 'b', 'a', 'c'])
Out[8]:

d      4
b      7
a     -5
c      3
dtype: int64
```

Series – selecting values

- Can use values in the index to select values:

```
In [9]: obj2['a']
Out[9]:
-5

In [11]:
obj2['d'] = 6
obj2[['c', 'a', 'd']]

Out[11]:
c 3
a -5
d 6
dtype: int64
```

```
In [12]: obj2[obj2 > 0]
Out[12]:
d 6
b 7
c 3
dtype: int64


In [14]:
obj2*2
Out[14]:
d 12 b 14 a -10 c 6 dtype:
int64
```

- Note that using e.g. NumPy boolean indexing, scalar multiplication or ufuncs (aggregation) will preserve the indexing.

Converting dict{} to Series

- Can think about a Series as a fixed-length, ordered dict, as it is a mapping of index values to data values:

```
In [15]: 'b' in obj2
Out[15]:
True

In [16]: 'e' in obj2
Out[16]:
False
```

```
In [17]: sdata = {'Ohio': 35000,
'Texas': 71000, 'Oregon': 16000,
'Utah': 5000}

In [19]: obj3 = pd.Series(sdata)
         obj3

Out[19]:
Ohio   35000
Oregon 16000
Texas  71000
Utah   5000
```

- Can create a Series from a dictionary by calling the panda.Series() function.

Differently Indexed Data

- Series automatically aligns differently-indexed data in arithmetic operations:

```
In [25]: obj3
Out[25]:

Ohio 35000
Oregon 16000
Texas 71000
Utah 5000
```

```
In [26]: obj4
Out[26]:

California NaN
Ohio 35000
Oregon 16000
Texas 71000
```

```
In [27]: obj3+obj4
Out[27]:

California       NaN
Ohio           70000
Oregon         32000
Texas         142000
Utah             NaN
```

DataFrame

- The DataFrame is designed to be similar to the R dataframe structure.
- Represents a tabular, spreadsheet-like data structure containing an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.)
- The DataFrame has both a row and column index.
- While a DataFrame stores the data internally in a two-dimensional format, can represent much higher-dimensional data in a tabular format using hierarchical indexing.
- Hierarchical indexing (more later in the course!) is a key component in many of the more advanced data-handling features in pandas.

Data inputs to DataFrame

Type	Notes
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
NumPy structured/record array	Treated as the "dict of arrays" case
dict of Series	Each value becomes a column. Indexes from each Series are unioned together to form the result's row index if no explicit index is passed.
dict of dicts	Each inner dict becomes a column. Keys are unioned to form the row index as in the "dict of Series" case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame's column labels
List of lists or tuples	Treated as the "2D ndarray" case
Another DataFrame	The DataFrame's indexes are used unless different ones are passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values become NA/missing in the DataFrame result

DataFrame - Construction

- Many ways to construct a DataFrame. A common way is to convert a dict structure using DataFrame():

```
In [30]: 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)
```

```
Out[30]:
```

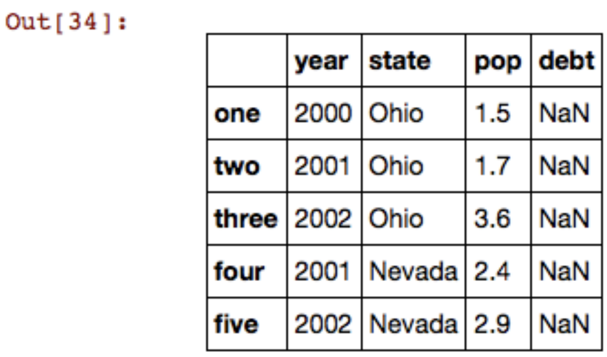
	pop	state	year
0	1.5	Ohio	2000
1	1.7	Ohio	2001
2	3.6	Ohio	2002
3	2.4	Nevada	2001
4	2.9	Nevada	2002

5 rows × 3 columns

DataFrame – Adding columns

- Can specify a sequence of columns and if the column doesn't exist, you will obtain NAs:

```
In [34]:
frame2 = pd.DataFrame(data,columns=['year','state','pop','debt'],
index=['one', 'two', 'three', 'four', 'five'])
```



```
Out[34]:
```

	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

DataFrame – Data Retrieval

- Can retrieve a column by name

```
In [35]: frame2['state']
Out[35]:
one Ohio
two Ohio
three Ohio
four Nevada
five Nevada
Name: state, dtype: object
```

or attribute (use . notation, very R-like!):

```
In [36]: frame2.year
Out[36]:
one 2000
two 2001
three 2002
four 2001
five 2002
Name: year, dtype: int64
```

The column returned when indexing a DataFrame is a view on the underlying data, not a copy.

To create a copy, use pd.Series.copy()

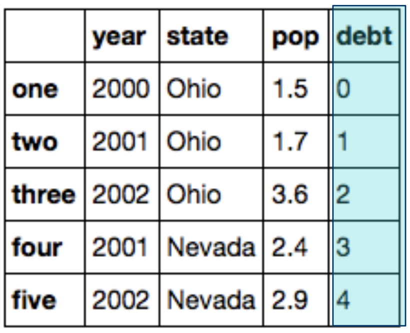
DataFrame – Data Retrieval

- Rows can be retrieved by position, name or other methods, e.g. ix

```
In [37]: frame2.ix['three']

Out[37]:
year 2002
state Ohio
pop 3.6
debt NaN
Name: three, dtype: object
```

- Columns can be easily reassigned values:

```
In [40]: frame2['debt'] = np.arange(5.)
```

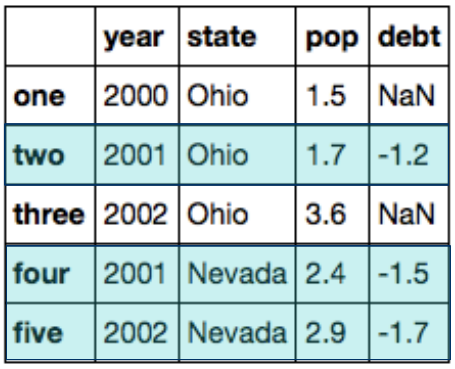


	year	state	pop	debt
one	2000	Ohio	1.5	0
two	2001	Ohio	1.7	1
three	2002	Ohio	3.6	2
four	2001	Nevada	2.4	3
five	2002	Nevada	2.9	4

DataFrame – Inserting Data

- If assigning a list or array to a column in a dataframe, if the sizes do not match exactly, NAs will be used to fill in the gaps:

```
In [41]: val = pd.Series([-1.2, -1.5, -1.7],
                          index=['two', 'four', 'five'])
         frame2['debt'] = val



         Out[42]:
```



	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7

DataFrame – Deleting Columns

- Assigning a column that doesn't exist will create a new column. The `del` keyword will delete columns as with a dict.

```
In [43]: frame2['eastern'] = frame2.state == 'Ohio'
```

```
Out[43]:
```

	year	state	pop	debt	eastern
one	2000	Ohio	1.5	NaN	True
two	2001	Ohio	1.7	-1.2	True
three	2002	Ohio	3.6	NaN	True
four	2001	Nevada	2.4	-1.5	False
five	2002	Nevada	2.9	-1.7	False

```
In [44]: del frame2['eastern']
         frame2.columns

Out[44]:
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Dropping entries from an axis

- Dropping one or more entries from an axis is relatively straightforward. Can use a Series or DataFrame approach:

```
In [83]:
obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
new_obj = obj.drop('c')
```

```
In [85]: obj
Out[85]:
a  0
b  1
c  2
d  3
e  4
```

```
In [86]: new_obj
Out[86]:
a  0
b  1
d  3
e  4
```

Dropping entries from an axis

```
data =
pd.DataFrame(np.arange(16).reshape((4, 4)),
index=['Ohio', 'Colorado', 'Utah', 'New York'],
columns=['one', 'two', 'three', 'four'])
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
data.drop(['Colorado', 'Ohio'])
```

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

```
data.drop(['two', 'four'], axis=1)
```

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

DataFrame – Creating Nested Dicts{}

- Can also create nested dicts:

```
pop = {'Nevada': {2001: 2.4, 2002: 2.9},
       'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
frame3 = pd.DataFrame(pop)
```

```
Out[47]:
```

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

Outer keys will become columns. Inner keys will become rows.

```
frame3.T
```

```
Out[48]:
```

	2000	2001	2002
Nevada	NaN	2.4	2.9
Ohio	1.5	1.7	3.6

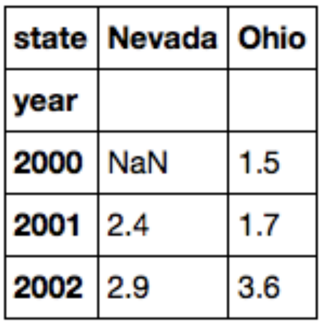
DataFrames can be transposed!

DataFrames – Setting Row/Column Names

- If a DataFrame's index and columns have their name attributes set, these will also be displayed:

```
In [49]:
frame3.index.name = 'year'; frame3.columns.name = 'state'
frame3
```

```
Out[49]:
```



state	Nevada	Ohio
year		
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

- Like Series, can use a `values` attribute (e.g. `frame3.values`) and if the column types are all different, dtype will be: `dtype=object`.

Merging Data Frames (join)

- pandas.merge allows two DataFrames to be joined on one or more keys.
- pandas.merge operates as an inner join.
- Can change this using the option how
- how allows two data frames to be joined with the options left, right, outer and inner, which tells pandas:
  - `left:` use only keys from left frame (SQL: left outer join)
  - `right:` use only keys from right frame (SQL: right outer join)
  - `outer:` use union of keys from both frames (SQL: full outer join)
  - `inner:` use intersection of keys from both frames (SQL: inner join)

Merging Data Frames (join)

```
left_frame = pd.DataFrame({'key': range(5), 'left_value': ['a',
'b', 'c', 'd', 'e']})

right_frame = pd.DataFrame({'key': range(2, 7), 'right_value':
['f', 'g', 'h', 'i', 'j']})

left_frame
right_frame

# inner join
pd.merge(left_frame, right_frame, on='key', how='inner')

# left outer join
pd.merge(left_frame, right_frame, on='key', how='left')


# right outer join
pd.merge(left_frame, right_frame, on='key', how='right')

# full outer join
pd.merge(left_frame, right_frame, on='key', how='outer')
```

Combining Data Frames (concat)

- Similar to the SQL union clause.
- pandas.concat takes a list of Series or DataFrames and returns a Series or DataFrame of the concatenated objects.
- can specify many objects to combine simultaneously (however need at least 2!).

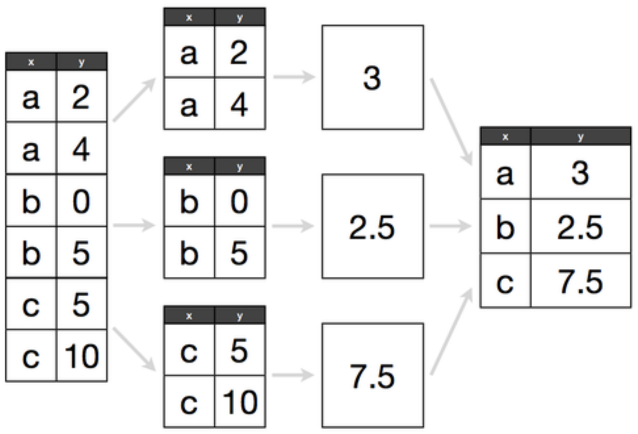
```
pd.concat([left_frame, right_frame])
```

- The default is that the objects are vertically appended.
- Columns with the same name will be combined.
- To combine objects side-by-side, this can be easily specified using the axis option, e.g.

```
pd.concat([left_frame, right_frame], axis=1)
```

Grouping Data Frames (groupby)

- Used to group data in some meaningful way, so that we can perform operations over each separate group (e.g. calculating the average expenditure on customer purchases, grouped by product type (clothing, books, DVDs, electronics, etc.)).
- This much-referenced graphic explains what's happening :)



Hadley Wickham
Data Science in R

- A very useful feature once you get your head around it!

Grouping Data Frames (groupby)

```
import pandas as pd

# Read in the data and check it out
mtcars = pd.read_csv("mtcars.csv")
mtcars.head()
mtcars.shape

# Compute basic descriptive stats over data
mtcars.describe()
mtcars.mean()   # also compute median, std, var, min, max, quantile
mtcars.mean(axis=1)    # compute row means (ie across columns)

# How many automatic transmission cars are there?
mtcars[mtcars["am"]==0].shape
mtcars[mtcars["am"]==0]

# Plot a histogram
mtcars["mpg"].hist()

# Group by number of carburettors and describe
grouped_by_carb = mtcars.groupby("carb")
grouped_by_carb.mean()
```

Grouping Data Frames (groupby)

```
# can group by more than one category
grouped_by_carb_am = mtcars.groupby(["carb","am"])

# compute statistics aggregated over groupings
import numpy as np
grouped_by_carb_am.agg([np.mean, np.std])

# count the number of cars in each combination of carb and am
counts = grouped_by_carb_am['carb'].count()

# plot the counts
import matplotlib.pyplot as plt
%matplotlib inline
df = counts.unstack()
ax = df.plot(kind='bar', stacked=True, figsize=(20, 10),
colormap="BuGn")
ax.set_ylabel("Count")
patches, labels = ax.get_legend_handles_labels()
ax.legend(patches, labels, loc='best')
```

Arithmetic between DataFrames and Series

- Can take advantage of broadcasting to perform operations between DataFrames and Series structures.

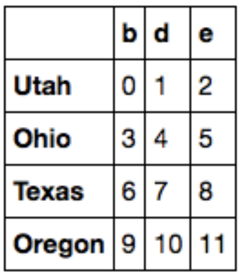
```
In [135]:
arr = np.arange(12.).reshape((3, 4))


Out[135]:
array([[ 0., 1., 2., 3.],
       [ 4., 5., 6., 7.],
       [ 8., 9., 10., 11.]])


In [136]:
arr - arr[0]


Out[136]:
array([[ 0., 0., 0., 0.],
       [ 4., 4., 4., 4.],
       [ 8., 8., 8., 8.]])
```

Arithmetic between DataFrames and Series

- By default, arithmetic between DataFrame and Series matches the index of the Series on the DataFrame's columns, broadcasting down the rows:

```
frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
columns=list('bde'), index=['Utah', 'Ohio', 'Texas', 'Oregon'])
series = frame.ix[0]
```

frame

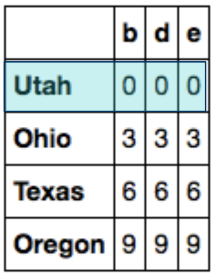


	b	d	e
Utah	0	1	2
Ohio	3	4	5
Texas	6	7	8
Oregon	9	10	11

series

```
b     0
d     1
e     2
Name: Utah, dtype: float64
```

```
In[138]: frame - series
```



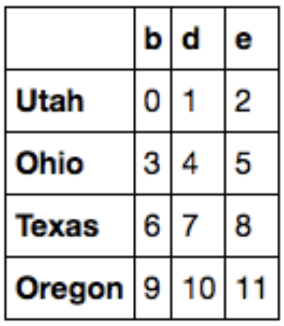
	b	d	e
Utah	0	0	0
Ohio	3	3	3
Texas	6	6	6
Oregon	9	9	9

Arithmetic between DataFrames and Series

- To broadcast down the columns and match on the rows, need to use an arithmetic method:

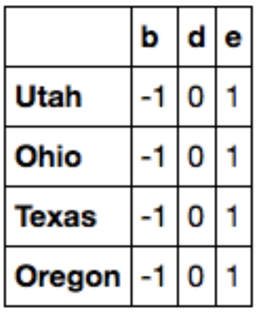
```
series3 = frame['d']
frame.sub(series3,axis=0)
```

frame



	b	d	e
Utah	0	1	2
Ohio	3	4	5
Texas	6	7	8
Oregon	9	10	11

frame – series3



	b	d	e
Utah	-1	0	1
Ohio	-1	0	1
Texas	-1	0	1
Oregon	-1	0	1

A Closer Look At Indexing

- Any array, or other sequence of labels used when constructing a Series or DataFrame, is internally converted to an Index:

```
In [50]: obj = pd.Series(range(3), index=['a', 'b', 'c'])
```

- Index objects are immutable and thus can't be modified by the user:

```
In [51]: obj.index[1] = 'd'

-----------------------------------------------------------
NameError                 Traceback (most recent call last)
.
.
.
<class 'pandas.core.index.Index'>' does not support mutable
operations.
```

- Immutability is important so that Index objects can be safely shared among data structures. However, we can kind of get around this …

Reindexing

- A critical method on pandas objects is `reindex`, which means to create a new object with the data conformed to a new index.

```
In [56]:
obj = pd.Series([4.5, 7.2, -5.3, 3.6],
      index=['d', 'b', 'a', 'c’])
```

- Calling reindex on this Series rearranges the data according to the new index, introducing missing values if any index values were not already present:

```
In [57]:
obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
obj2

Out[57]:
a -5.3
b 7.2
c 3.6
d 4.5
e NaN
```

Reindexing

- Can avoid the usage of NaN when filling in an empty index by using `fill_value`:

```
In [61]:
obj = pd.Series([4.5, 7.2, -5.3, 3.6],
index=['d', 'b', 'a', 'c'])
obj
```

```
Out[61]:
d 4.5
b 7.2
a -5.3
c 3.6
```

- If we re-index, we can initialise all non-existing indices:

```
In [63]:
obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)

Out[63]:
a -5.3
b 7.2
c 3.6
d 4.5
e 0.0
```

Reindexing

- Can also fill empty values associated with indices, using non-obvious (but useful) methods!

```
In [65]: obj3 = pd.Series(['blue', 'purple', 'yellow'],
index=[0, 2, 4])

Out[65]:
0 blue
2 purple
4 yellow
dtype: object

In [66]: obj3 = obj3.reindex(range(6), method='ffill')

Out[66]:
0 blue
1 blue
2 purple
3 purple
4 yellow
5 yellow
```

reindex options

Argument	Description
`ffill` or `pad`	Fill (or carry) values forward
`bfill` or `backfill`	Fill (or carry) values backward

A Closer Look At Reindexing

- reindex can alter either the (row) index, columns, or both.

```
frame = pd.DataFrame(np.arange(9).reshape((3, 3)), index=['a',
'c', 'd'],
columns=['Ohio', 'Texas', 'California'])
```

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

- Specifying just a sequence will re-order the rows (by default).

```
frame2 = frame.reindex(['a', 'b', 'c', 'd'])
```

	Ohio	Texas	California
a	0	1	2
b	NaN	NaN	NaN
c	3	4	5
d	6	7	8

A Closer Look At Reindexing

- The columns can be reindexed using the columns keyword:

```
states = ['Texas', 'Utah', 'California']
frame.reindex(columns=states)
```

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

- Can reindex both simultaneously:

```
frame.reindex(index=['a', 'b', 'c', 'd'],
method='ffill', columns=states)
```

	Texas	Utah	California
a	1	NaN	2
b	1	NaN	2
c	4	NaN	5
d	7	NaN	8

A Closer Look At Reindexing

- reindexing can be done more succinctly by label-indexing with `ix`:

```
frame.ix[['a', 'b', 'c', 'd'], states]
```

reindex function arguments

Argument	Description
`index`	New sequence to use as index. Can be `Index` instance or any other sequence-like Python data structure. An `Index` will be used exactly as is without any copying
`method`	Interpolation (fill) method
`fill_value`	Substitute value to use when introducing missing data by reindexing
`limit`	When forward- or backfilling, maximum size gap to fill
`level`	Match simple Index on level of MultiIndex, otherwise select subset of
`copy`	Do not copy underlying data if new index is equivalent to old index. `True` by default (i.e. always copy data).

Indexing: a few things to watch out for!

- Series indexing works as NumPy indexing, except can use the Series index values in place of just the integers.

```
In[94]:
obj = pd.Series(np.arange(4.),index=['a', 'b', 'c', 'd'])
print(obj[1],' ',obj['b'])

Out[94]:
1.0 1.0
```

- Can use slices, boolean indexing, etc., as seen with NumPy.

- However slicing with labels in pandas takes the endpoint (unlike base Python and NumPy). This matches R's functionality.

```
In[95]:  obj['b':'c']
Out[95]:
b      1
c      2
```

Indexing: a few things to watch out for!

- DataFrame indexing allows the retrieval of one or more columns either with a single value or sequence:

```
data =
pd.DataFrame(np.arange(16).reshape((4, 4)),
index=['Ohio', 'Colorado', 'Utah', 'New York'],
columns=['one', 'two', 'three', 'four'])
```

```
data['two']

Out[103]:
Ohio        1
Colorado   5
Utah       9
New York 13
```

```
data[['two', 'four']]
```

	two	four
Ohio	1	3
Colorado	5	7
Utah	9	11
New York	13	15

- However there are also a few special cases …

Indexing: a few things to watch out for!

- Slicing with labels behaves as with Series slicing. Slicing with row numbers however:

```
In[120]:   data[:2]
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

```
In[121]: data[data['three'] > 5]
```

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

- *Slicing with indices behaves as in base Python!*

- Can also use a pandas structure in a boolean statement:

```
In[126]: data < 5
In[127]: data[data<5]=0 # Try it!
```

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

DataFrame and ix

- Label-indexing of DataFrames can also use the `ix` function

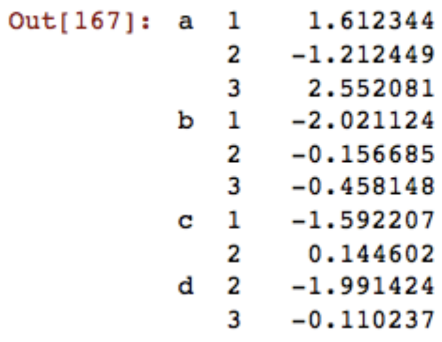
```
In[127]:   data.ix['Colorado', ['two', 'three']]


In [128]:  data.ix[['Colorado','Utah',['two', 'three']]


In[129]:   data.ix[2]


In[130]:   data.ix[:'Utah', 'two']


In[131]:   data.ix[data.three > 5,:3]
```

Note use of . operator to access column variable – very R-like!

Hierarchical Indexing: Series

- Enables multiple (two or more) index levels on an axis.
- Provides a way to work with higher dimensional data at lower dimensions. Mimics R functionality.

```
data = pd.Series(np.random.randn(10),
index=[['a', 'a', 'a', 'b', 'b', 'b', 'c', 'c', 'd', 'd'],
       [1, 2, 3, 1, 2, 3, 1, 2, 2, 3]])
```



```
Out[167]: a  1    1.612344
             2   -1.212449
             3    2.552081
          b  1   -2.021124
             2   -0.156685
             3   -0.458148
          c  1   -1.592207
             2    0.144602
          d  2   -1.991424
             3   -0.110237
```

Hierarchical Indexing: Series

- Makes partial indexing possible.

```
data['b']
```

```
Out[169]: 1   -2.021124
          2   -0.156685
          3   -0.458148
          dtype: float64
```

```
data['b':'c']  # or data.ix[['b', 'c']]
```

```
Out[170]: b  1   -2.021124
             2   -0.156685
             3   -0.458148
          c  1   -1.592207
             2    0.144602
          dtype: float64
```

```
data[:,2]
```

```
Out[172]: a   -1.212449
          b   -0.156685
          c    0.144602
          d   -1.991424
          dtype: float64
```

Hierarchical Indexing: Series
stack and unstack

- Another replication of R functionality.

```
data.stack()  #notice NANs
```

Out[173]:

	1	2	3
a	1.612344	-1.212449	2.552081
b	-2.021124	-0.156685	-0.458148
c	-1.592207	0.144602	NaN
d	NaN	-1.991424	-0.110237

```
data.stack().unstack()
```

```
Out[167]: a  1    1.612344
             2   -1.212449
             3    2.552081
          b  1   -2.021124
             2   -0.156685
             3   -0.458148
          c  1   -1.592207
             2    0.144602
          d  2   -1.991424
             3   -0.110237
```

Hierarchical Indexing: DataFrames

- With DataFrames, either axis can have a hierarchical index.

```
frame = pd.DataFrame(np.arange(12).reshape((4, 3)),index=[['a',
'a', 'b', 'b'], [1, 2, 1, 2]],columns=[['Ohio', 'Ohio',
'Colorado'], ['Green', 'Red', 'Green']])
```

		Ohio		Colorado
		Green	Red	Green
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

Hierarchical Indexing: DataFrames

- With DataFrames, either axis can have a hierarchical index.

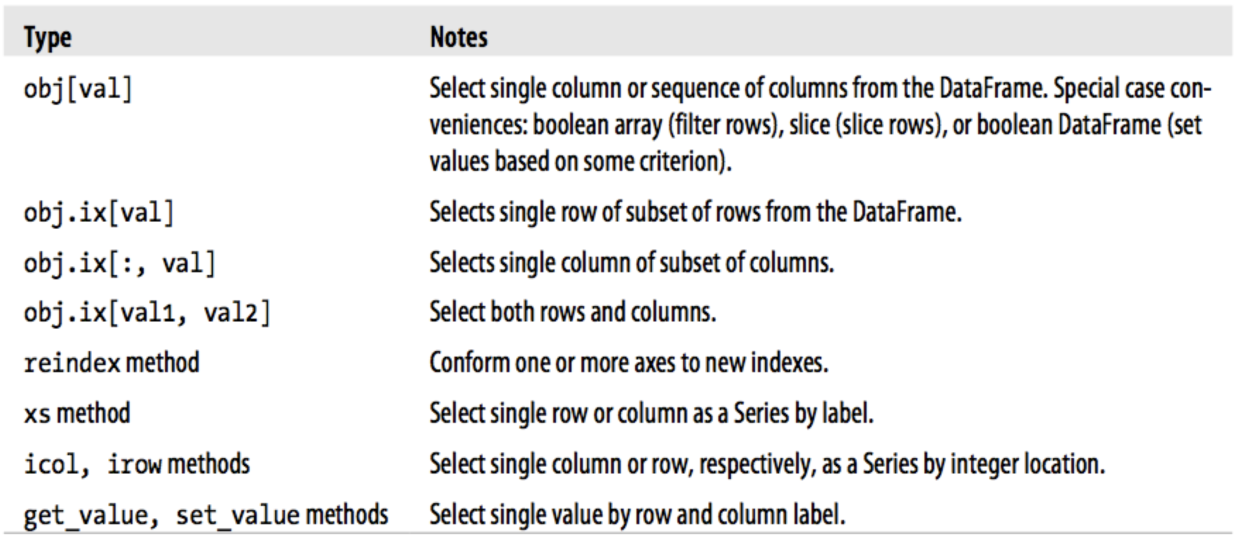
```
frame.index.names = ['key1', 'key2']
frame.columns.names = ['state', 'colour']
```

	state	Ohio		Colorado
	colour	Green	Red	Green
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

```
frame['Ohio'] # partial indexing
```

	colour	Green	Red
key1	key2		
a	1	0	1
	2	3	4
b	1	6	7
	2	9	10

DataFrame Indexing Options



Type	Notes
obj[val]	Select single column or sequence of columns from the DataFrame. Special case conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion).
obj.ix[val]	Selects single row of subset of rows from the DataFrame.
obj.ix[:, val]	Selects single column of subset of columns.
obj.ix[val1, val2]	Select both rows and columns.
reindex method	Conform one or more axes to new indexes.
xs method	Select single row or column as a Series by label.
icol, irow methods	Select single column or row, respectively, as a Series by integer location.
get_value, set_value methods	Select single value by row and column label.

Function mapping: apply

- In R, there is much usage of a suite of apply functions (apply, sapply, lapply…). We'll see these in the next lecture!
- Python-defined ufuncs 'automatically' apply themselves to each element in a pandas DataFrame. However user-defined functions do not.
- pandas replicates the R functionality in part by providing an apply function for this purpose.

Function mapping: apply

```
frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

	b	d	e
Utah	-1.284567	-0.724012	-0.410331
Ohio	-1.028510	0.117753	-0.048079
Texas	-1.210197	-2.220176	-1.135762
Oregon	0.479340	-0.005978	-1.975791

```
f = lambda x: x.max() - x.min()
frame.apply(f)
b 2.142073
d 2.710013
e 0.349985

frame.apply(f,axis=1)
Utah   2.550449
Ohio   1.386588
Texas  0.905827
Oregon 1.248064
```

Many of the most common array statistics (e.g. sum, mean) are DataFrame methods, so using apply is not necessary. (in R this is not the case!)

Descriptive Statistics (pandas)

Method	Description
`count`	Number of non-NA values
`describe`	Compute set of summary statistics for Series or each DataFrame column
`min, max`	Compute minimum and maximum values
`argmin, argmax`	Compute index locations (integers) at which minimum or maximum value obtained, respectively
`idxmin, idxmax`	Compute index values at which minimum or maximum value obtained, respectively
`quantile`	Compute sample quantile ranging from 0 to 1
`sum`	Sum of values
`mean`	Mean of values
`median`	Arithmetic median (50% quantile) of values
`mad`	Mean absolute deviation from mean value
`var`	Sample variance of values
`std`	Sample standard deviation of values
`skew`	Sample skewness (3rd moment) of values
`kurt`	Sample kurtosis (4th moment) of values

Descriptive Statistics (pandas)

cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute 1st arithmetic difference (useful for time series)
pct_change	Compute percent changes

apply and applymap

```
def f(x):  # returns a series
    return Series([x.min(), x.max()], index=['min', 'max'])

frame.apply(f)
       b                 d            e
min -0.555730       0.281746     -1.296221
max  1.246435       1.965781      1.393406


format = lambda x: # returns a formatted string
    '%.2f' % x

frame.applymap(format)
       b      d      e
Utah -0.20 0.48 -0.52
Ohio -0.56 1.97  1.39
Texas 0.09 0.28  0.77
Oregon 1.25 1.01 -1.30
```

applymap is used for element-wise function application

Missing Data

- We can fill in, or filter out missing data (NAs). The functionalities available are:

Argument	Description
`dropna`	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
`fillna`	Fill in missing data with some value or using an interpolation method such as `'ffill'` or `'bfill'`.
`isnull`	Return like-type object containing boolean values indicating which values are missing / NA.
`notnull`	Negation of `isnull`.

```
from numpy import nan as NA
data = Series([1, NA, 3.5, NA, 7])
data.dropna()   # data[data.notnull()] gives the same result

0 1
2 3.5
4 7
```

Detecting Missing Data in Series

- Can specify indices when passing a dict structure in Series():

```
In [22]: states = ['California', 'Ohio', 'Oregon', 'Texas']
         obj4 = pd.Series(sdata, index=states)

Out[22]:
California NaN
Ohio 35000
Oregon 16000
Texas 71000
dtype: float64
```

Introduces a missing number NaN (indicates NA values)

- Use `isnull` and `notnull` to detect missing data:

```
In [26]: pd.isnull(obj4)
Out[26]:
California    True
Ohio          False
Oregon        False
Texas         False
```

```
In [27]: pd.notnull(obj4)
Out[27]:
California    False
Ohio          True
Oregon        True
Texas         True
```

Filling In Missing Data in Series

- We can use fillna() to do the job.
- Recall: `states = ['California', 'Ohio', 'Oregon', 'Texas']`
  `obj4 = pd.Series(sdata, index=states)`
- Use `fillna()` to replace the NaNs.

```
In [8]:obj4.fillna(0)

Out[8]: California       0
       Ohio          35000
       Oregon        16000
       Texas         71000
```

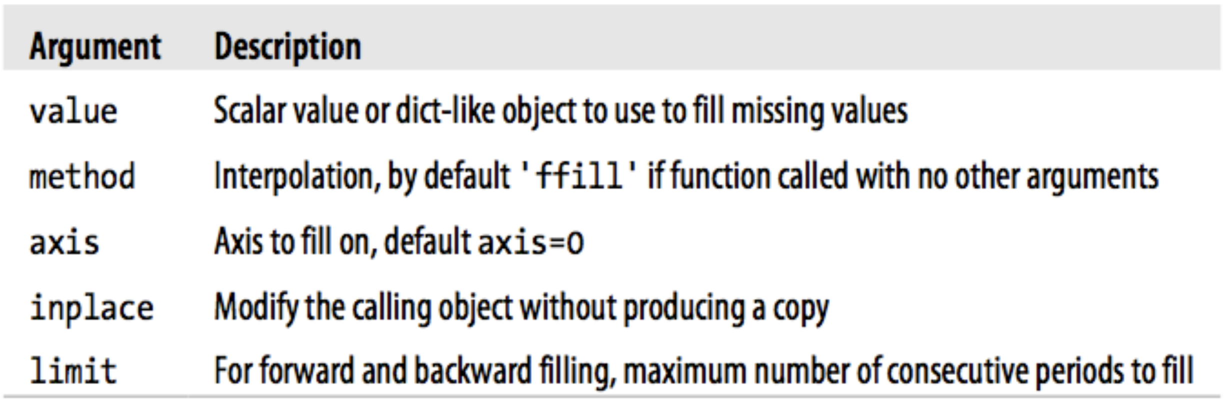
- **Caution!** Simply replacing NaNs because they are inconvenient is very poor practice - you should never alter data for ease. You should only fill in missing values when you know why they occur and how to replace them.
- Otherwise we are better off ignoring the rows of data containing NAs and focusing on data we can trust.

Filling In Missing Data

- Can use `fillna()` in a number of ways:

```
df.fillna(0)                   # fill with 0s

df.fillna({1: 0.5, 3: -1})     # fill with a dict.
                               # Non-existant indices ignored


_ = df.fillna(0, inplace=True) # fill in place (no copy created)


df.fillna(method='bfill')      # can use the reindexing methods


data = Series([1., NA, 3.5, NA, 7])    # fill with the mean
data.fillna(data.mean())
```

Filling In Missing Data



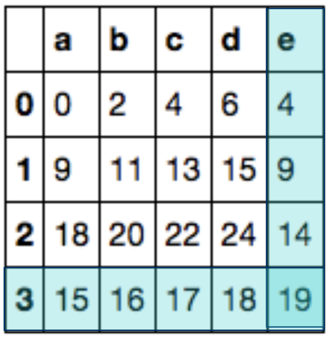
Argument	Description
`value`	Scalar value or dict-like object to use to fill missing values
`method`	Interpolation, by default `'ffill'` if function called with no other arguments
`axis`	Axis to fill on, default `axis=0`
`inplace`	Modify the calling object without producing a copy
`limit`	For forward and backward filling, maximum number of consecutive periods to fill

DataFrame – Inserting Missing Data

- To avoid issues with arithmetic operations, may wish to fill NaNs with a special value (e.g. 0) in these cases.

```
df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),columns=list('abcd'))
df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),columns=list('abcde'))

df1.add(df2, fill_value=0)
```

```
Out[134]:
```



	a	b	c	d	e
0	0	2	4	6	4
1	9	11	13	15	9
2	18	20	22	24	14
3	15	16	17	18	19

- Can use `fill_value` with `add`, `sub`, `div` and `mul`.

Filtering Missing Data

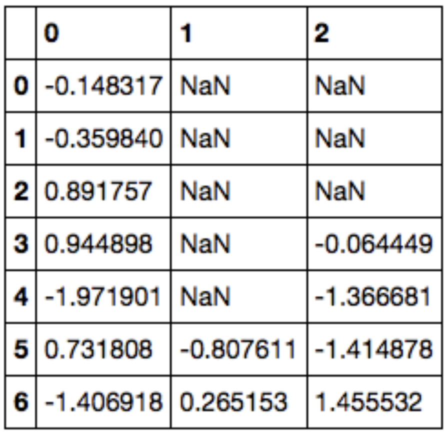
- DataFrames are a bit more complex.

```
data = DataFrame([[1., 6.5, 3.], [1., NA, NA], [NA, NA, NA],
[NA, 6.5, 3.]])
data
        0     1     2
0       1.   6.5   3.
1       1    NaN   NaN
2     NaN    NaN   NaN
3     NaN    6.5   3

cleaned = data.dropna()

        0     1     2
0       1.   6.5   3.

cleaned = data.dropna(how='all') # returns NaN-only rows
cleaned = data.dropna(axis=1,how='all') # drops columns
```

Filtering Missing Data

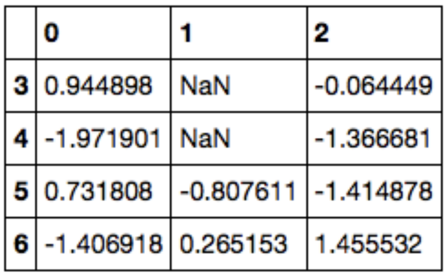
- Can also use a threshold argument `thresh`

```
df = pd.DataFrame(np.random.randn(7, 3))
df.ix[:4, 1] = NA; df.ix[:2, 2] = NA
df
```



	0	1	2
0	-0.148317	NaN	NaN
1	-0.359840	NaN	NaN
2	0.891757	NaN	NaN
3	0.944898	NaN	-0.064449
4	-1.971901	NaN	-1.366681
5	0.731808	-0.807611	-1.414878
6	-1.406918	0.265153	1.455532

```
df.dropna(thresh=2)
```



	0	1	2
3	0.944898	NaN	-0.064449
4	-1.971901	NaN	-1.366681
5	0.731808	-0.807611	-1.414878
6	-1.406918	0.265153	1.455532