UPPSALA
UNIVERSITET

Data Containers: Efficient Memory Storage Using HDF5, And Pandas

Day 5

Advanced Scientific Programming with Python

HDF5

Adapted from:

https://github.com/scopatz/hdf5-is-for-lovers/

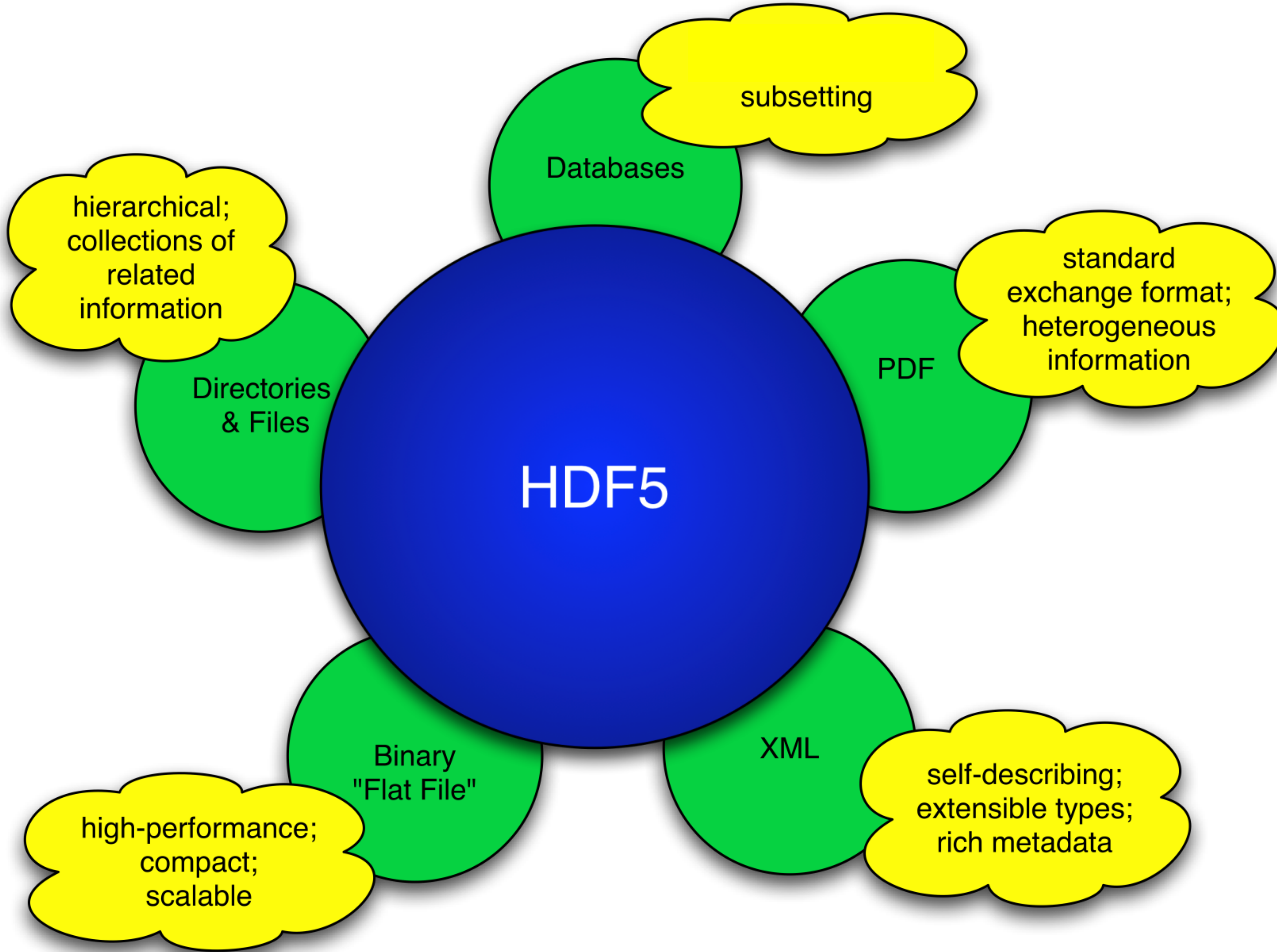
https://support.hdfgroup.org/HDF5/doc/H5.intro.html

https://www.slideshare.net/HDFEOS/introduction-to-hdf5-data-and-programming-models-handson-exercise

What HDF5

- HDF5 stands for (H)eirarchical (D)ata (F)ormat (5)ive.
- It is supported by the HDFGroup.
- At its core HDF5 is binary file type specification.
- However, what makes HDF5 great is the numerous libraries written to interact with files of this type and their extremely rich feature set.
- Can represent complex data objects as well as associated metadata
- A **portable** file format with **no limit** on the number or size of data objects in the collection
- Free software (BSD, MIT kind of license)
- Implements a high-level API with C, C++, Fortran 90, and Java interfaces

HDF5 Has Characteristics Of …

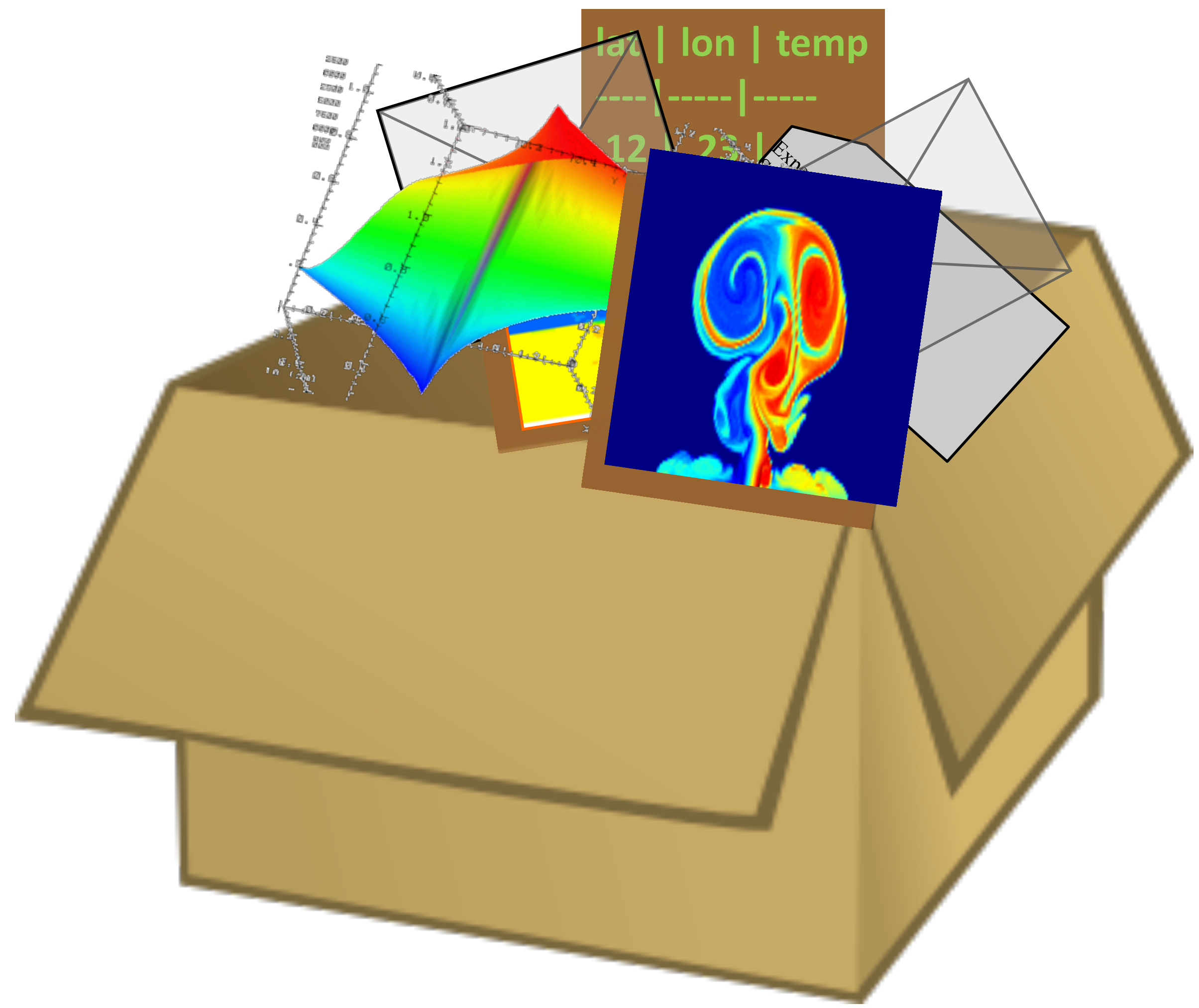


HDF5 Is Designed…

- for small or high volume and/or complex data
- for every size and type of system (portable)
- for flexible, efficient storage and I/O
- to enable applications to evolve in their use of HDF5 and to accommodate new models
- to support long-term data preservation
- Use it as a file format tool kit

HDF5 File

An HDF5 file is a **container** that holds data objects.



Intro to HDF5

- HDF5 files are organized in a hierarchical structure, with two primary structures: groups and datasets.

    1. **HDF5 group:** a grouping structure containing instances of zero or more groups or datasets, together with supporting metadata.

    2. **HDF5 dataset:** a multidimensional array of data elements, together with supporting metadata.

- Working with groups and group members is similar in many ways to working with directories and files in UNIX. As with UNIX directories and files, objects in an HDF5 file are often described by giving their full (or absolute) path names.

`/` signifies the root group.

`/foo` signifies a member of the root group called foo.

`/foo/zoo` signifies a member of the group foo, which in turn is a member of the root group.

- Any HDF5 group or dataset may have an associated attribute list. An HDF5 attribute is a user-defined HDF5 structure that provides extra information about an HDF5 object. Attributes are described in more detail below.

HDF5 Dataset

Metadata

Dataspace

Rank	Dimensions
3	Dim_1 = 4
	Dim_2 = 5
	Dim_3 = 7

Datatype

Integer

Properties

Chunked

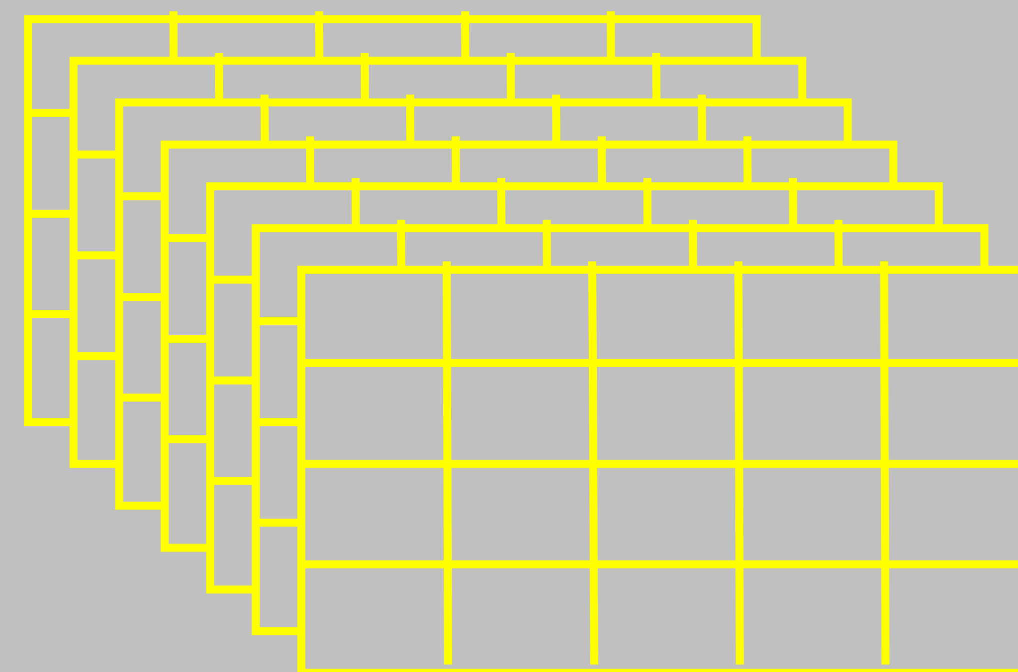
Compressed

(optional) Attributes

Time = 32.4

Pressure = 987

Data



Multi-dimensional array of identically typed data elements

- HDF5 datasets **organize and contain** “raw data values”.
  - HDF5 datatypes describe individual data elements.
  - HDF5 dataspaces describe the logical layout of the data elements.

HDF5 Datasets

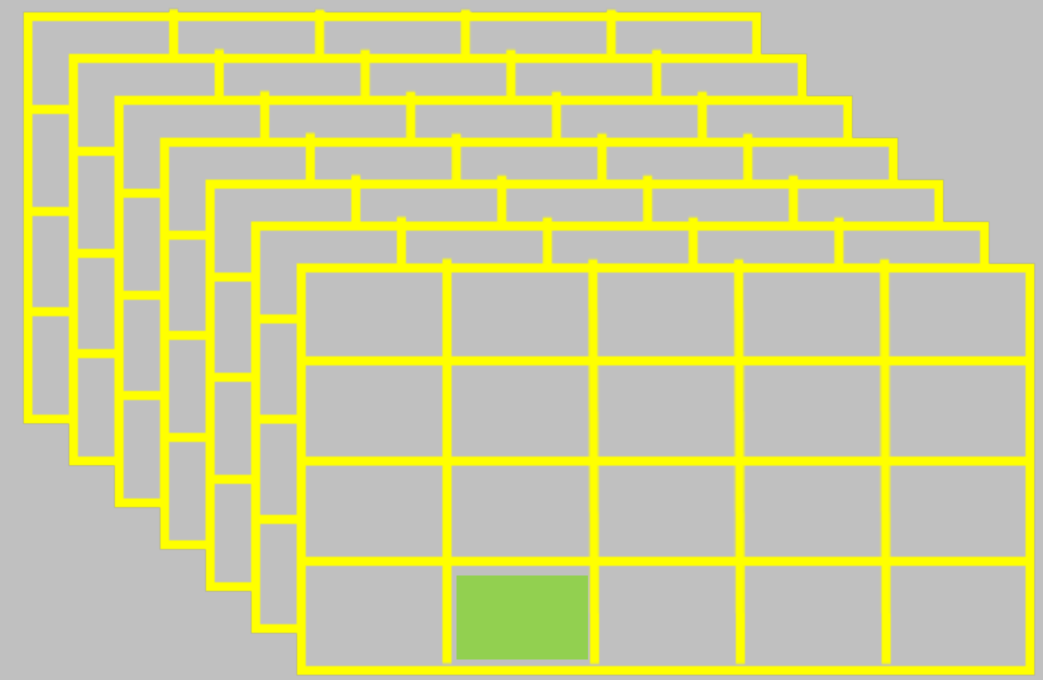
- A dataset is stored in a file in two parts: a header and a data array.
- There are four essential classes of information in any header: name, datatype, dataspace, and storage layout:
- Name. A dataset name is a sequence of alphanumeric ASCII characters.
- Datatype. Defines the kind of data stored in the dataset. Can be one of multiple builtin types (such as int, float, etc…) or user made compound datatypes (akin to a C struct).
- h5py (the main Python APIs for HDF5) supports the NumPy datatypes.

HDF5 Dataset & Datatype

HDF5 Datatype

Integer 32bit LE

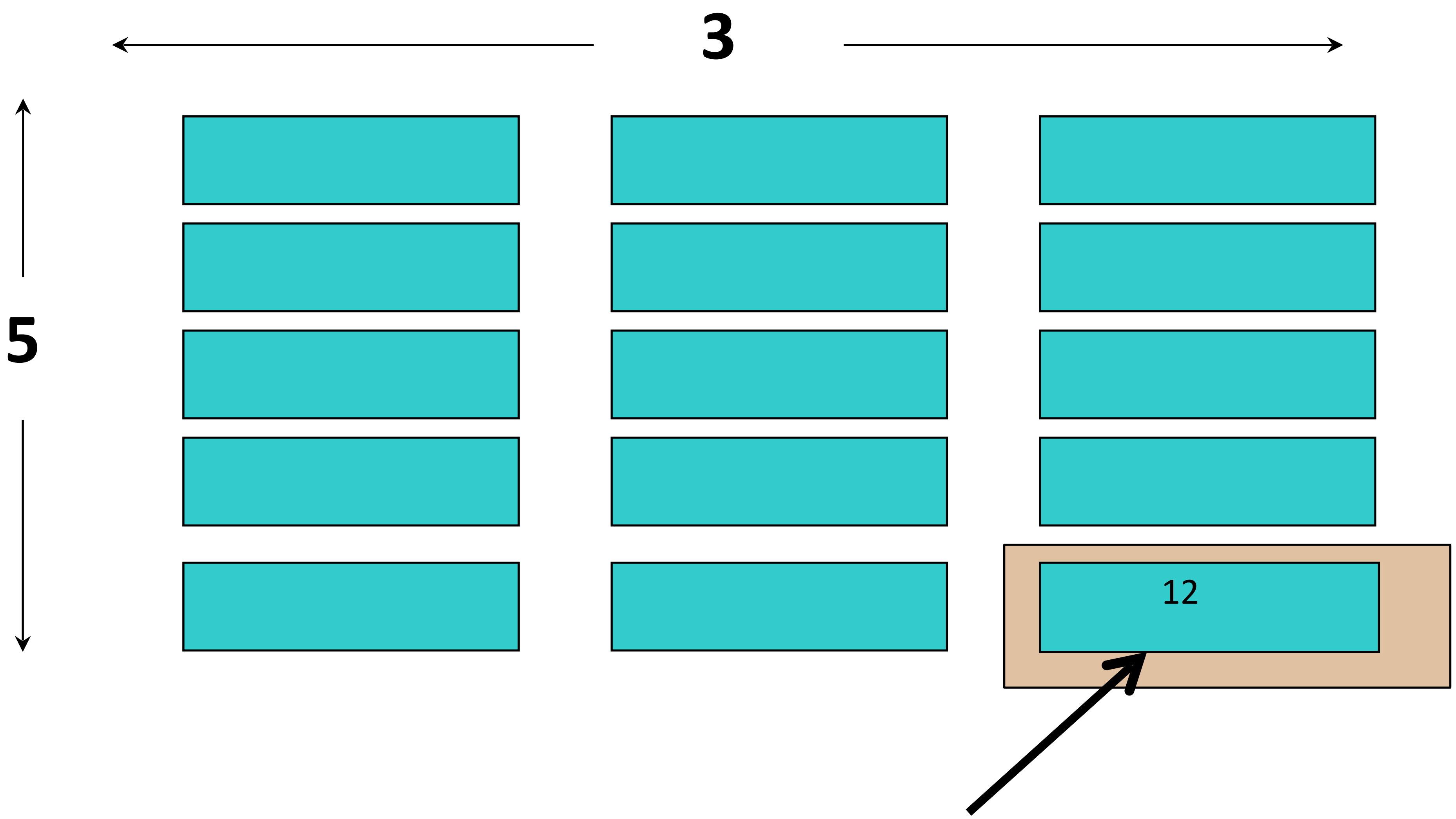
Specifications for single data element



Multi-dimensional array of identically typed data elements

- HDF5 datasets organize and contain “raw data values”.
  - HDF5 datatypes **describe individual data elements.**

HDF5 Dataset



Datatype: **32-bit Integer**

Dataspace: **Rank = 2**

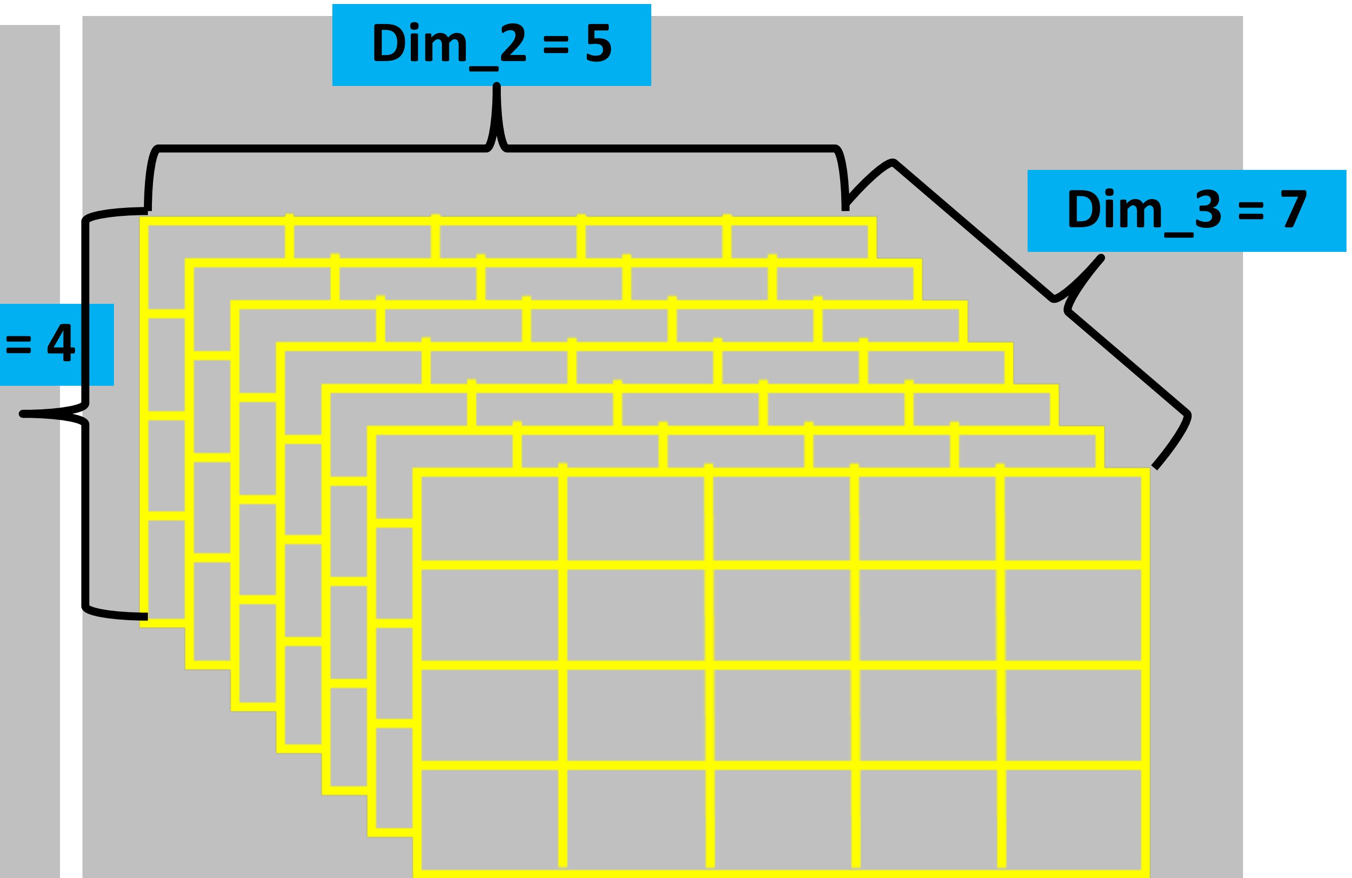
Dimensions = 5 x 3

HDF5 Dataset & Dataspace

HDF5 Dataspace

Rank	Dimensions
3	Dim_1 = 4
	Dim_2 = 5
	Dim_3 = 7

Specifications for array dimensions



Multi-dimensional array of identically typed data elements

- HDF5 datasets organize and contain “raw data values”.

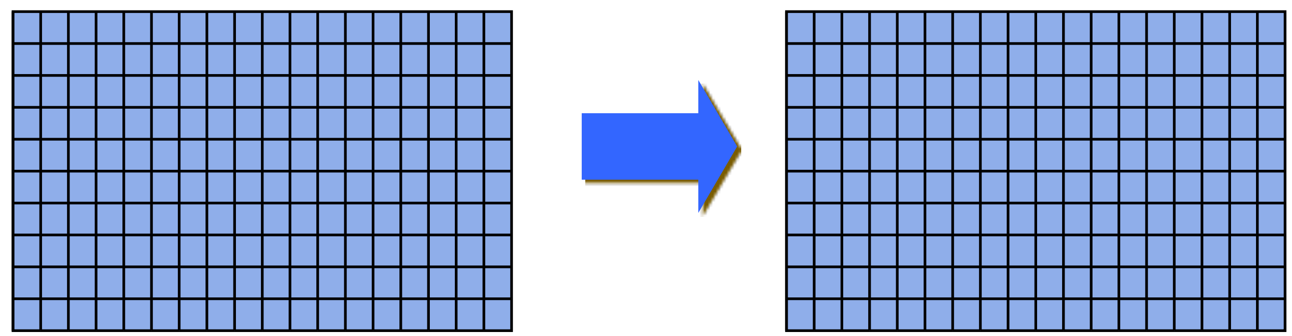
  - HDF5 dataspaces **describe the logical layout of the data elements**

HDF5 Datasets

- Dataspace. A dataset dataspace describes the dimensionality of the dataset. The dimensions of a dataset can be fixed (unchanging), or they may be unlimited, which means that they are extendible (i.e. they can grow larger).
- Storage layout. The layout can be contiguous or chunked. In contiguous, the data is stored in the same linear way that it is organized in memory.
- Chunked storage involves dividing the dataset into equal-sized "chunks" that are stored separately. Chunking has three important benefits.
    1. It makes it possible to achieve good performance when accessing subsets of the datasets, even when the subset to be chosen is orthogonal to the normal storage order of the dataset.
    2. It makes it possible to compress large datasets and still achieve good performance when accessing subsets of the dataset.
    3. It makes it possible efficiently to extend the dimensions of a dataset in any direction.

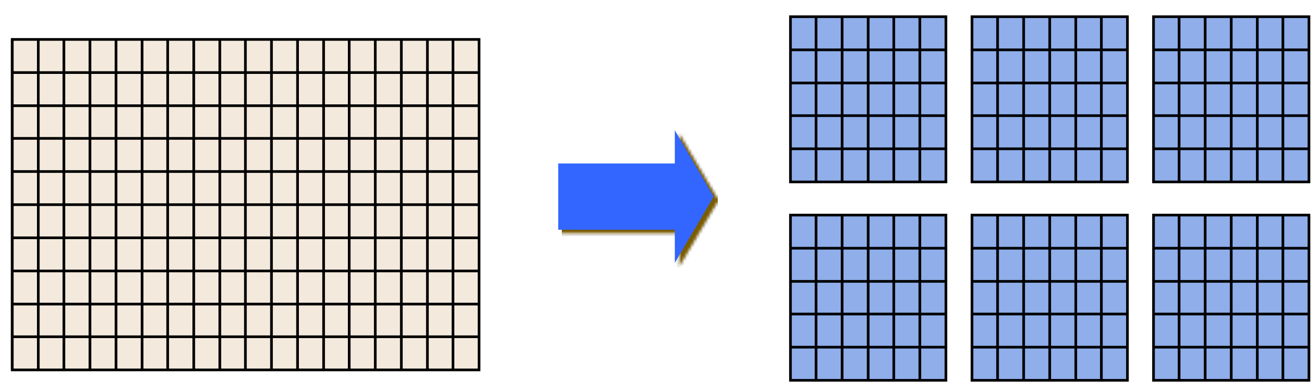
Dataset Storage Properties

Contiguous (default)



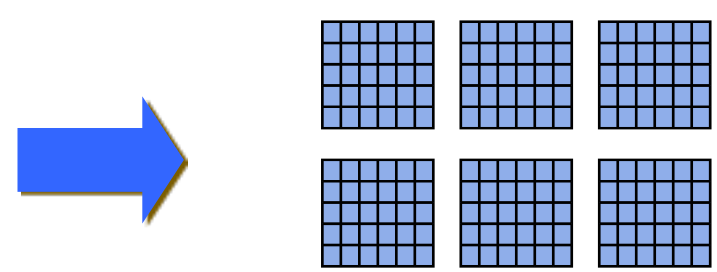
Data elements stored physically adjacent to each other

Chunked



Better access time for subsets; extendible

Chunked & Compressed



Improves storage efficiency, transmission speed

HDF5 Attributes

- Attributes are small named datasets that are attached to primary datasets, groups, or named datatypes.
- Attributes can be used to describe the nature and/or the intended usage of a dataset or group.
- An attribute has two parts: (1) a name and (2) a value. The value part contains one or more data entries of the same datatype.
- When accessing attributes, they can be identified by name or by an index value.
- The use of an index value makes it possible to iterate through all of the attributes associated with a given object.

HDF5 Attributes

- The HDF5 format and I/O library are designed with the assumption that attributes are small datasets.
- They are always stored in the object header of the object they are attached to.
- Because of this, large datasets should **not** be stored as attributes.
- How large is "large" is not defined by the library and is up to the user's interpretation. (Large datasets with metadata can be stored as supplemental datasets in a group with the primary dataset.)

h5py

- **h5py** and PyTables are the two most widely used HDF5 Python APIs.
- We'll start with **h5py**
- In **h5py** HDF5 Groups work like Python dictionaries, and datasets work like NumPy arrays
- Lets see how to create an HDF5 file:

```
>>> import h5py
>>> import numpy as np
>>>
>>> f = h5py.File("mytestfile.hdf5", “w”)
```

- You'll end up with a File object which you can use to for example to create a new dataset:

```
>>> dset = f.create_dataset("mydataset", (100,), dtype='i')
```

h5py

- The object we created isn't an array, but an HDF5 dataset. Like NumPy arrays, datasets have both a shape and a data type:

```
>>> dset.shape
(100,)
>>> dset.dtype
dtype('int32')
```

- They also support array-style slicing. This is how you read and write data from a dataset in the file:

```
>>> dset[...] = np.arange(100)
>>> dset[0]
0
>>> dset[10]
10
>>> dset[0:100:10]
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

Groups and hierarchical organization

- Every object in an HDF5 file has a name, and they're arranged in a POSIX-style hierarchy with /-separators

```
>>> dset.name
u'/mydataset'
```

- Every object in an HDF5 file has a name, and they're arranged in a POSIX-style hierarchy with /-separators

- The "folders" in this system are called **groups**. The File object we created is itself a group, in this case the root group, named **/**:

```
>>> f.name
u'/'
```

- Creating a subgroup is accomplished via **`create_group`**

```
>>> grp = f.create_group("subgroup")
```

Groups and hierarchical organization

- All Group objects also have the **create_*** methods like **File**:

```
>>> dset2 = grp.create_dataset("another_dataset", (50,), dtype='f')
>>> dset2.name
u'/subgroup/another_dataset'
```

- By the way, you don't have to create all the intermediate groups manually. Specifying a full path works just fine:

```
>>> dset3 = f.create_dataset('subgroup2/dataset_three', (10,), dtype='i')
>>> dset3.name
u'/subgroup2/dataset_three'
```

- Groups support most of the Python dictionary-style interface. You retrieve objects in the file using the item-retrieval syntax:

```
>>> dataset_three = f['subgroup2/dataset_three']
```

- Iterating over a group provides the names of its members:

```
>>> for name in f:
...     print name
mydataset
subgroup
subgroup2
```

Groups and hierarchical organization

- Containership testing also uses names:

```
>>> "mydataset" in f
True
>>> "somethingelse" in f
False
```

- You can even use full path names:

```
>>> "subgroup/another_dataset" in f
True
```

- There are also the familiar **`keys()`**, **`values()`**, **`items()`** and **`iter()`** methods, as well as **`get()`**.

- Iterating over an entire file is accomplished with the Group methods **`visit()`** and **`visititems()`**, which takes a function:

```
>>> def printname(name):
...     print name
>>> f.visit(printname)
mydataset
subgroup
subgroup/another_dataset
...
```

Attributes

- One of the best features of HDF5 is that you can store metadata right next to the data it describes.
- All groups and datasets support attached named bits of data called attributes.
- Attributes are accessed through the **`attrs`** proxy object, which again implements the dictionary interface:

```
>>> dset.attrs['temperature'] = 99.5
>>> dset.attrs['temperature']
99.5
>>> 'temperature' in dset.attrs
True
```

h5py Notebook

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$

Adapted from:

Python for Data Analysis

http://shop.oreilly.com/product/0636920023784.do

Goals Of Pandas

- Data structures with labeled axes supporting automatic or explicit data alignment.
- This prevents common errors resulting from misaligned data and working with differently-indexed data coming from different sources.
- Integrated time series functionality.
- The same data structures handle both time series data and non-time series data.
- Arithmetic operations and reductions (like summing across an axis) would pass on the metadata (axis labels).
- Flexible handling of missing data.
- Merge and other relational operations found in popular database databases (SQLbased,for example).

Pandas Data Structures

- To get started with pandas, you will need to get comfortable with its two workhorse data structures: Series and DataFrame
- A Series is a one-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its index.
- The simplest Series is formed from only an array of data:

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

In [6]: obj.values
Out[6]: array([ 4, 7, -5, 3])
In [7]: obj.index
Out[7]: Int64Index([0, 1, 2, 3])
```

Pandas Series

- Often it will be desirable to create a Series with an index identifying each data point:

```
In [8]: obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
In [9]: obj2
Out[9]:
d 4
b 7
a -5
c 3
In [10]: obj2.index
Out[10]: Index([d, b, a, c], dtype=object)

In [11]: obj2['a']
Out[11]: -5
In [12]: obj2['d'] = 6
In [13]: obj2[['c', 'a', 'd']]
Out[13]:
c 3
a -5
d 6
```

Pandas Data Frames

- A DataFrame 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; it can be thought of as a dict of Series (one for all sharing the same index).
- There are numerous ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays

```
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 = 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
```

Pandas Essential Functionality

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

```
In [79]: obj = Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
In [80]: obj
Out[80]:
d 4.5
b 7.2
a -5.3
c 3.6

In [81]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
In [82]: obj2
Out[82]:
a -5.3
b 7.2
c 3.6
d 4.5
e NaN

In [83]: obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)
Out[83]:
a -5.3
b 7.2
c 3.6
d 4.5
e 0.0
```

- Dropping entries from an axis (**drop**)

```
In [94]: obj = Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
In [95]: new_obj = obj.drop('c')
In [96]: new_obj
Out[96]:
a 0
b 1
d 3
e 4
In [97]: obj.drop(['d', 'c'])
Out[97]:
a 0
b 1
e 43

In [98]: data = DataFrame(np.arange(16).reshape((4, 4)),
 ....: index=['Ohio', 'Colorado', 'Utah', 'New York'],
 ....: columns=['one', 'two', 'three', 'four'])
In [99]: data.drop(['Colorado', 'Ohio'])
Out[99]:
          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
```

Pandas Essential Functionality

- Indexing, selection, and filtering
- Series indexing, selection and filtering (obj[...]) works analogously to NumPy arrays, except you can use the Series's index values instead of only integers.

```
In [102]: obj = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
In [103]: obj['b']
Out[103]: 1.0
In [104]: obj[1]
Out[104]: 1.0
```

- DataFrames also behave similarly

Pandas Essential Functionality

```
In [6]: data = DataFrame(np.arange(16).reshape((4, 4)),
   ...: index=['Ohio', 'Colorado', 'Utah', 'New York'],
   ...: columns=['one', 'two', 'three', 'four'])

In [7]: data
Out[7]:
          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

In [8]: data['two']
Out[8]:
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64

In [9]: data[['three', 'one']]
Out[9]:
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
```

Pandas Essential Functionality

```
In [11]:  data[data['three'] > 5]
Out[11]:
          one  two  three  four
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15

In [12]: data < 5
Out[12]:
            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

In [13]: data[data < 5] = 0

In [14]: data
Out[14]:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

Arithmetic And Data Alignment

- One of the most important pandas features is the behavior of arithmetic between objects with different indexes.
- When adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs.

```
In [15]: s1 = Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])

In [16]: s2 = Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f',
'g'])
```

```
In [17]: s1
Out[17]:
a     7.3
c    -2.5
d     3.4
e     1.5
dtype: float64
```

```
In [18]: s2
Out[18]:
a    -2.1
c     3.6
e    -1.5
f     4.0
g     3.1
dtype: float64
```

```
In [19]: s1 + s2
Out[19]:
a     5.2
c     1.1
d     NaN
e     0.0
f     NaN
g     NaN
dtype: float64
```

Arithmetic And Data Alignment

- In arithmetic operations between differently-indexed objects, you might want to fill with a special value, like 0, when an axis label is found in one object but not the other:

```
In [20]: s1.add(s2, fill_value=0)
Out[20]:
a    5.2
c    1.1
d    3.4
e    0.0
f    4.0
g    3.1
dtype: float64
```

- Operations in general exclude missing data

```
In [21]: data
Out[21]:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15

In [22]: data.mean()
Out[22]:
one      5.00
two      6.75
three    7.50
four     8.25
dtype: float64

In [23]: data.mean(1)
Out[23]:
Ohio         0.0
Colorado     4.5
Utah         9.5
New York    13.5
dtype: float64
```

Function Application

- You can apply arbitrary functions to your data

```
In [28]: data.apply(np.sqrt)
Out[28]:
               one       two     three      four
Ohio      0.000000  0.000000  0.000000  0.000000
Colorado  0.000000  2.236068  2.449490  2.645751
Utah      2.828427  3.000000  3.162278  3.316625
New York  3.464102  3.605551  3.741657  3.872983

In [27]: data.apply(np.cumsum)
Out[27]:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8   14     16    18
New York   20   27     30    33

In [30]: data.apply(np.cumsum,1)
Out[30]:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5     11    18
Utah        8   17     27    38
New York   12   25     39    54
```

Function Application

- You can apply arbitrary functions to your data

```
In [29]: data.apply(np.sum)
Out[29]:
one      20
two      27
three    30
four     33
dtype: int64
```

- You can apply NumPy element-wise functions directly

```
In [32]: np.sqrt(data)
Out[32]:
               one       two     three      four
Ohio      0.000000  0.000000  0.000000  0.000000
Colorado  0.000000  2.236068  2.449490  2.645751
Utah      2.828427  3.000000  3.162278  3.316625
New York  3.464102  3.605551  3.741657  3.872983
```

And Much Much More…

- Pandas is a very rich package, on par with NumPy
- There are excellent web resources to learn it
- For example http://pandas.pydata.org/
- I would also highly recommend "Python for Data Analysis" where most of the previous example were taken from.
- Written by the author of Pandas, but it covers much more!