Data Science In Python 2

Instructor: Imam Jabar Shidiq R, S.Kom.
Faishal Wahiduddin, S.Kom.

What you'll learn today

- Learning Python libraries for numerical and scientific computing that extend Python's basic math module such as **NumPy/SciPy** and **Pandas**.
- **NumPy/SciPy** – numerical and scientific function libraries.
- **Pandas** – high-performance data structures and data analysis tools.

We Will Learn

For Data Wrangling

Numpy: Library For Scientific Computing

Introduction

NumPy

- Numpy is the core library for scientific computing in Python. It provides a **high-performance multidimensional array object and tools** for working with **arrays.**

- Introduces objects for **multidimensional arrays and matrices**, as well as functions that allow to **easily perform advanced mathematical and statistical** operations on those objects

- Provides vectorization of mathematical operations on arrays and matrices which significantly improves the performance.

- Many other python libraries are built on NumPy.

SciPy

- SciPy contains **more fully-featured versions** of the linear algebra modules, as well as many other numerical algorithms.
- Collection of algorithms for linear algebra, differential equations, numerical integration, optimization, statistics and more.
- Built on NumPy.

SciPy

- SciPy's functionality is implemented in a number of specific sub-modules. These include:
  - Special mathematical functions (scipy.special) -- airy, elliptic, bessel, etc.
  - Integration (scipy.integrate)
  - Optimization (scipy.optimize)
  - Interpolation (scipy.interpolate)
  - Fourier Transforms (scipy.fftpack)
  - Signal Processing (scipy.signal)
  - Linear Algebra (scipy.linalg)
  - Compressed Sparse Graph Routines (scipy.sparse.csgraph)
  - Spatial data structures and algorithms (scipy.spatial)
  - Statistics (scipy.stats)
  - Multidimensional image processing (scipy.ndimage)
  - etc

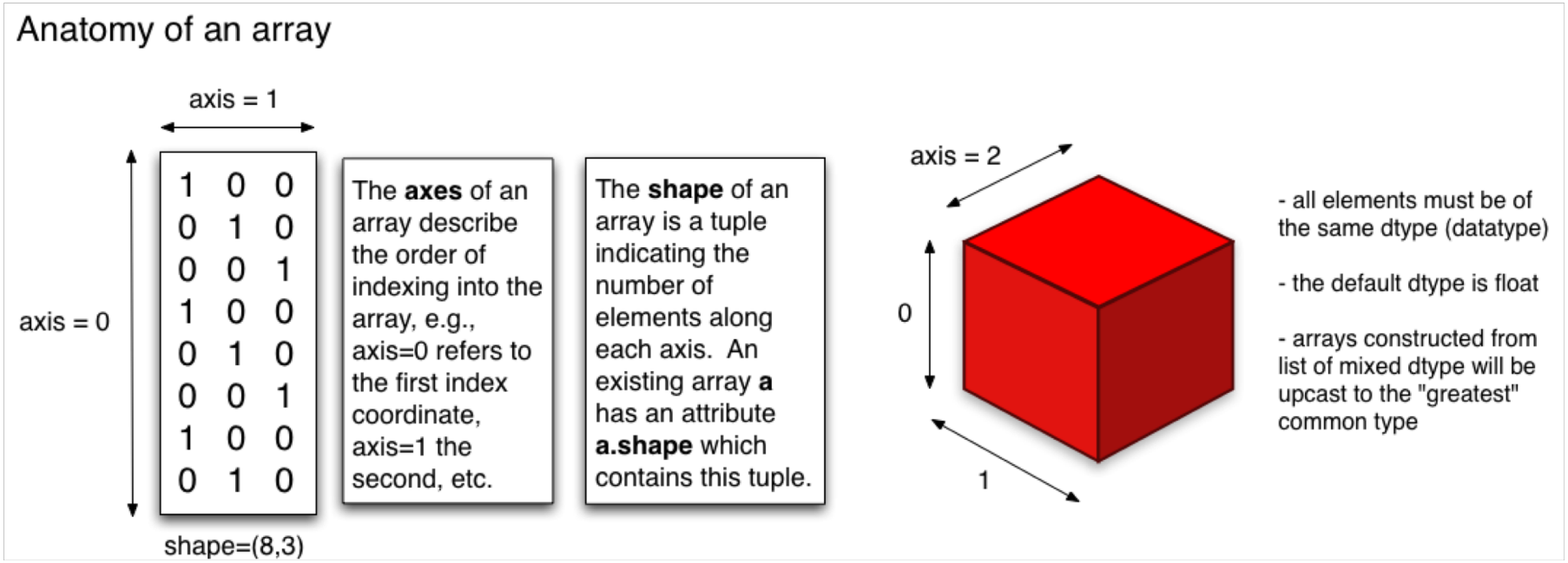
NumPy Array

- NumPy Array is a powerful N-dimensional array object that contain a grid of values, all of the same type which is in the form of rows and columns.

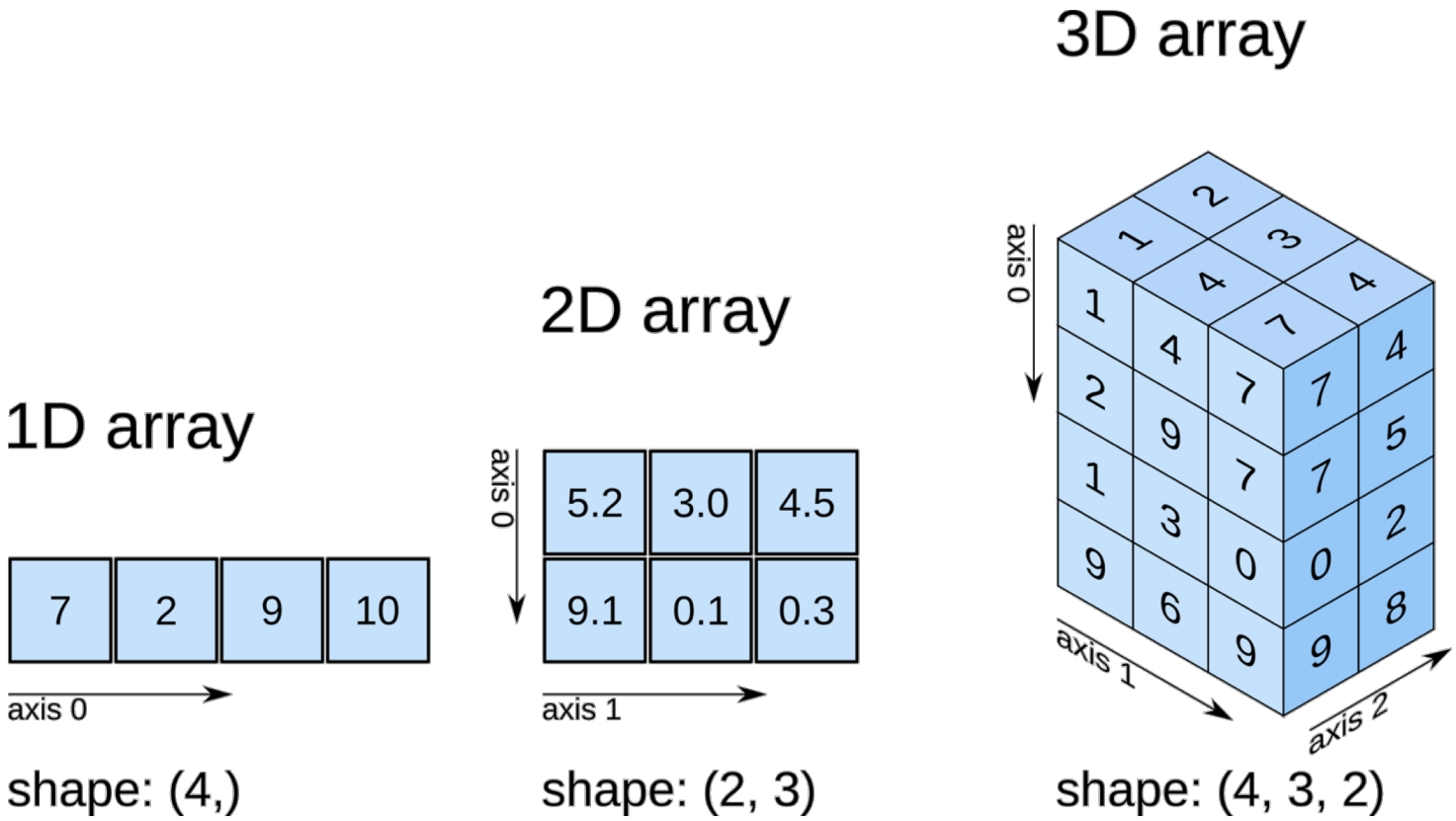
- We can initialize numpy arrays from nested Python lists and access it elements. In order to perform numpy operations.

- There are a couple of mechanisms for creating arrays in NumPy:
  - Conversion from other Python structures (e.g., lists, tuples).
  - Built-in NumPy array creation (e.g., arange, ones, zeros, etc.).
  - Reading arrays from disk, either from standard or custom formats (e.g. reading in from a CSV file).
  - and others ...

Anatomy Of An Array



Anatomy Of An Array



NumPy Array

- In general, any numerical data that is stored in an array-like container can be converted to an ndarray through use of the array() function. The most obvious examples are sequence types like lists and tuples.

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])                  # Single-dimensional array
array([1, 2, 3])

>>> b = np.array([(1, 2, 3),(4,5,6)])        # Multi-dimensional Array
array([[1, 2, 3], [4, 5, 6]])
```

NumPy Array Create Functions

- Numpy also provides built-in functions to create arrays:

- **zeros(shape)** – creates an array filled with – values with the specified shape. The default dtype is float64.

```
>>> np.zeros((2,3))
array([[0., 0., 0.], [0., 0., 0.]])
```

- **ones(shape)** – creates an array filled with 1 values
- **arange()** – creates arrays with regularly incrementing values.

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(2, 10, dtype=np.float)
array([2., 3., 4., 5., 6., 7., 8., 9.])
>>> np.arange(2, 3, 0.1)
array([2., 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])
```

NumPy Array Create Functions

- **full(shape, number)** – creates a constant array

```
>>> np.full((2,2),7)
array([[7, 7], [7, 7]])
```

- **eye(row)** – creates an identity matrix
- **linspace()** – creates arrays with a specified number of elements, and spaced equally between the specified beginning and end values.

```
>>> np.linspace(1., 4., 6)
array([ 1. , 1.6, 2.2, 2.8, 3.4, 4. ])
```

- **random.random(shape)** – creates arrays with random floats over the interval

```
>>> np.random.random(2,3)
array([[ 0.75688597, 0.41759916, 0.35007419],
       [ 0.77164187, 0.05869089, 0.98792864]])
```

NumPy Array Printing

- Printing an array can be done with the print statement

```
>>> import numpy as np
>>> a = np.arange(3)
>>> print a
[0 1 2]
>>> a
array([0, 1, 2])
>>> b = np.arange(9).reshape(3,3)
>>> print b
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

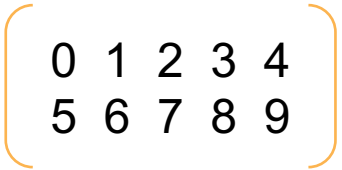
NumPy Array Indexing

- Single-dimension indexing is accomplished as usual.

```
>>> a = np.arange(10)
>>> a[2]
2
>>> a[-2]
8
```

- Multi-dimensional arrays support multi-dimensional indexing.

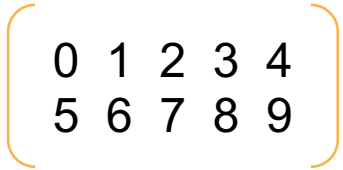
```
>>> b.shape = (2,5) # now b is 2-dimensional
>>> x[1,3]
8
>>> x[1,-1]
9
```



NumPy Array Indexing

- Using fewer dimensions to index will result in a subarray

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



NumPy Array Indexing

- Slicing: Similar to Python lists, numpy arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array.

```
>>> a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[2:5]
array([2, 3, 4])
>>> a[:-7]
array([0, 1, 2])
>>> a[1:7:2]
array([1, 3, 5])
```

NumPy Array Indexing

- Slicing at 2 dimensional array

```
>>> b = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
>>> b[:2, 1:3]
array([[ 2, 3], [6, 7]])
>>> b[1:3, ::2]
array([[ 5, 7], [9, 11]])
```

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

NumPy Array Indexing

Be Careful!

- A slice of an array is a view into the same data, so modifying it will modify the original array.

```
>>> print (b[0,1])
2
>>> b[0,1] = 70
>>> print (b[0,1])
70
```

NumPy Array Indexing

- Boolean array indexing: Boolean array indexing lets you pick out arbitrary elements of an array. Frequently this type of indexing is used to select the elements of an array that satisfy some condition.

```
>>> a = np.array([[1,2], [3, 4], [5, 6]])
>>> bool_idx = a > 2
array([[False, False], [True, True], [True, True]])
>>> print(a[bool_idx])
[3 4 5 6]
>>> print(a[a > 2])
[3 4 5 6]
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

We use boolean array indexing to construct a rank 1 array

NumPy Array Operations

```
>>> a = np.arange(5)
>>> b = np.arange(5)
>>> a + b
array([0, 2, 4, 6, 8])
>>> a - b
array([0, 0, 0, 0, 0])
>>> a ** 2
array([ 0,  1,  4,  9, 16])
>>> a > 3
array([False, False, False, False,  True], dtype=bool)
>>> 10 * np.sin(a)
array([ 0., 8.41470985, 9.09297427, 1.41120008, -7.56802495])
>>> a * b
array([ 0,  1,  4,  9, 16])
```

- Basic operations apply element-wise. The result is a new array with the resultant elements.
- Operations like *= and += will modify the existing array.

a: [0 1 2 3 4]

b: [0 1 2 3 4]

NumPy Array Operations

- NumPy has fast built-in aggregation and math functions for working on arrays

- Mathematics Function:

- `np.add()`
- `np.subtract()`
- `np.multiply()`
- `np.divide()`
- `np.exp()`
- `np.sqrt()`
- `np.log()`
- `np.dot()`

- Aggregation Function:

- `sum()`
- `min()`
- `max(axis=0)`
- `mean()`
- `std()`
- `np.median(a)`

NumPy Array Operations

```
>>> a = np.arange(5)
>>> b = np.arange(5)
>>> np.add(a,b)
array([0, 2, 4, 6, 8])
>>> np.multiply(a,b)
array([0, 1, 4, 9, 16])
>>> np.exp(a)
array([1., 2.71828183, 7.3890561, 20.08553692, 54.59815003])
>>> np.sqrt(a+b)
array([0., 1.41421356, 2., 2.44948974, 2.82842712])
>>> np.log(a)
array([-inf, 0., 0.69314718, 1.09861229, 1.38629436])
>>> np.dot(a,b)
30
```

a: 0 1 2 3 4

b: 0 1 2 3 4

NumPy Array Operations

```
>>> a = np.random.random((2,3))
>>> a
array([[ 0.68166391, 0.98943098, 0.69361582],
       [ 0.78888081, 0.62197125, 0.40517936]])
>>> a.sum()
4.1807421388722164
>>> a.min()
0.4051793610379143
>>> a.max(axis=0)
array([ 0.78888081, 0.98943098, 0.69361582])
>>> a.min(axis=1)
array([ 0.68166391, 0.40517936])
```

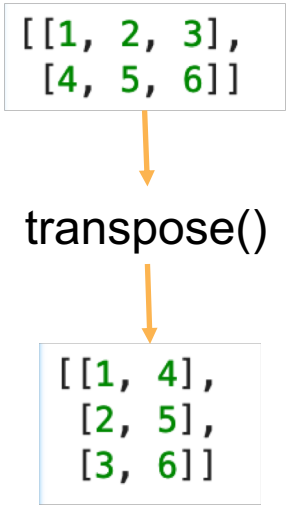
Array Manipulation

Introduction and Examples

Transposing

- Transposing your arrays actually does is permuting the dimensions of it. Or, in other words, you switch around the shape of the array.

```
>>> x = np.array([[1,2,3],[4,5,6]])
>>> x
array([[1, 2, 3], [4, 5, 6]])
>>> x.T
array([[1, 4], [2, 5], [3, 6]])
>>> x.transpose()
array([[1, 4], [2, 5], [3, 6]])
```

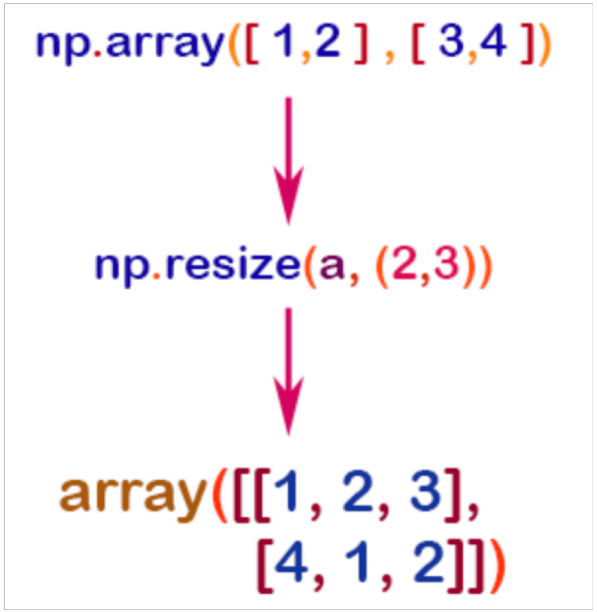


Resize & Reshape

- Both reshape and resize change the shape of the numpy array. The difference is that using **resize will affect the original array** while using **reshape create a new reshaped instance of the array**.

- **Resize**

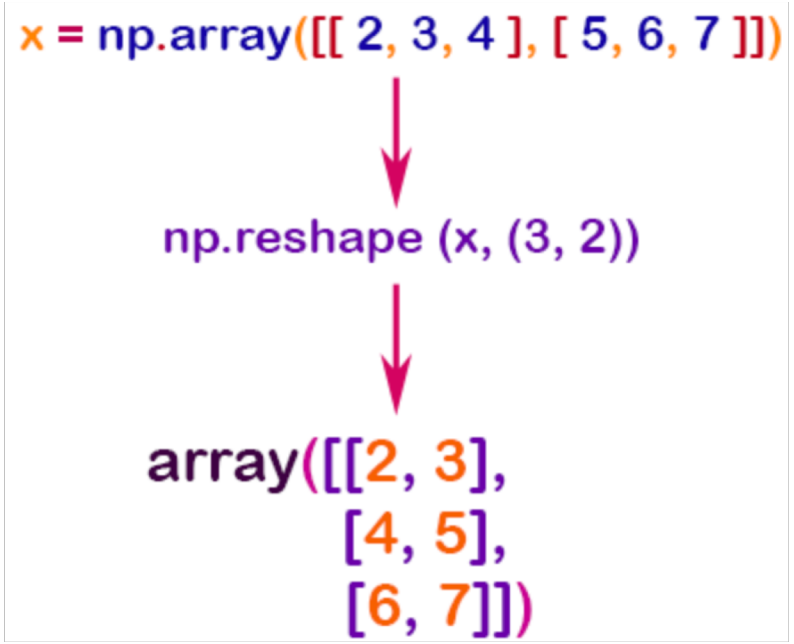
```
>>> x = np.array([[1,2],[3,4]])
>>> x.shape
(2, 2)
>>> x.resize(2,3)
array([[1, 2, 3], [4, 1, 2]])
```



Resize & Reshape

- Reshape

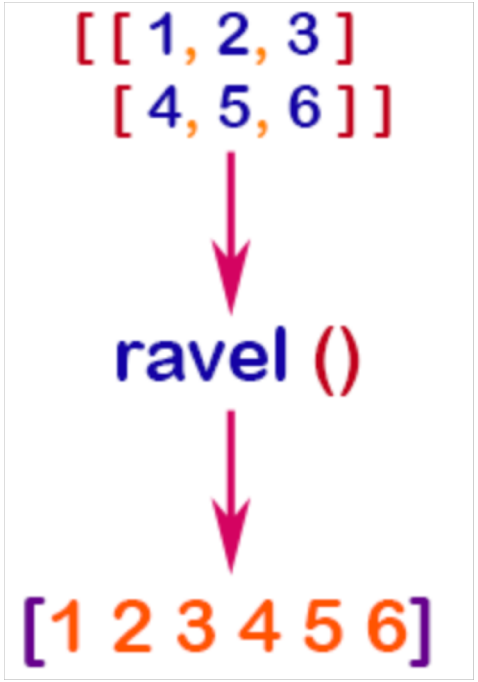
```
>>> x = np.array([[2,3,4], [5,6,7]])
>>> x.size
6
>>> x.reshape((3,2))
array([[2, 3], [4, 5], [6, 7]])
```



Resize & Reshape

- **Ravel:** The ravel() function is used to create a contiguous flattened array.

```
>>> x = np.array([[2,3,4], [5,6,7]])
>>> x.size
6
>>> x.ravel()
array([2, 3, 4, 5, 6, 7])
```

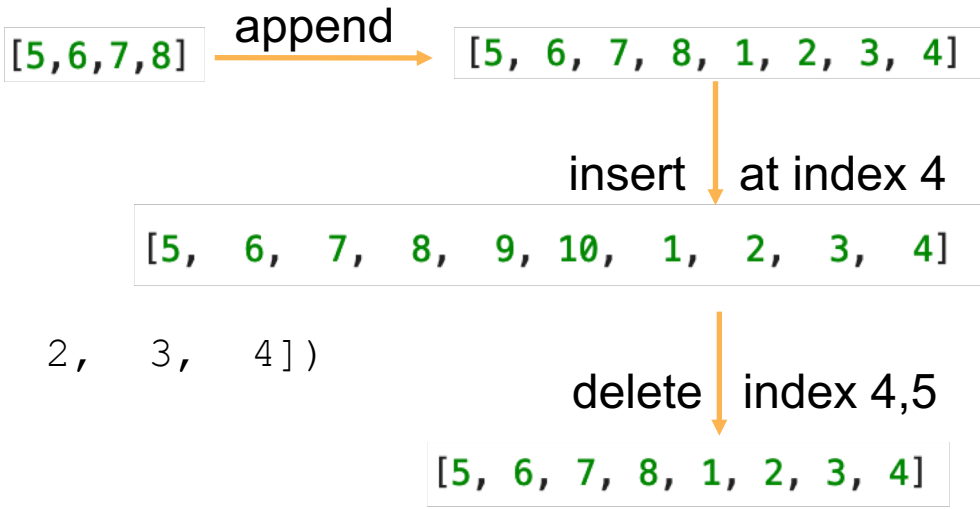


Adding & Removing

- When append arrays to the original array, they are “glued” to the end of that original array.
- The insert function is used to insert values wherever we want inside the array.
- We can also delete array elements using delete function.

- **Single-Dimension:**

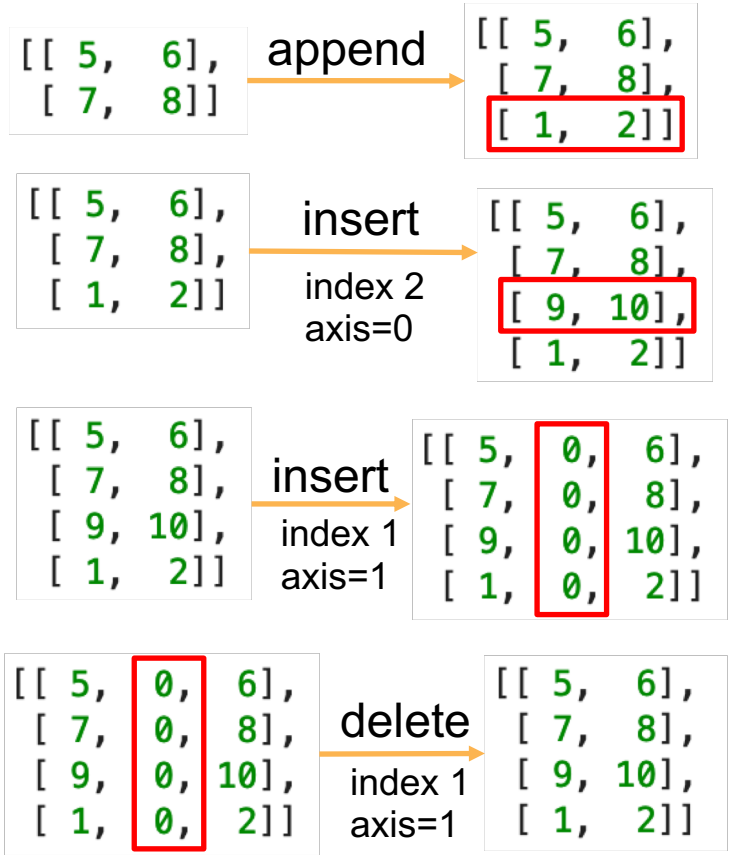
```
>>> x = np.array([5,6,7,8])
>>> x
array([5,6,7,8])
>>> x = np.append(x,[1,2,3,4])
array([5, 6, 7, 8, 1, 2, 3, 4])
>>> x = np.insert(x, 4, [9,10])
array([5,  6,  7,  8,  9, 10,  1,  2,  3,  4])
>>> x = np.delete(x,[4,5])
array([5, 6, 7, 8, 1, 2, 3, 4])
```



Adding & Removing

- Multi-Dimension:

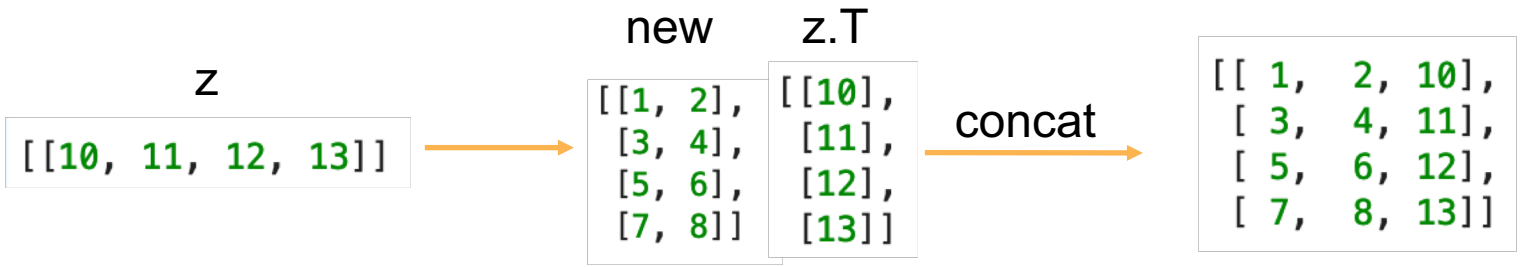
```
>>> z = np.array([[5,6],[7,8]])
>>> z
array([[5,6],[7,8]])
>>> z = np.append(z,[[1,2]], axis=0)
array([[5,6],[7,8],[1,2])
>>> z = np.insert(z, 2, [9,10], axis=0)
array([[5,6],[7,8],[9,10],[1,2])
>>> z = np.insert(z, 1, [0,0,0,0], axis=1)
array([[5,0,6],[7,0,8],[9,0,10],[1,0,2]])
z = np.delete(z,[1], axis=1)
array([[5,6],[7,8],[9,10],[1,2]])
```



Join & Split

- Concatenate Arrays

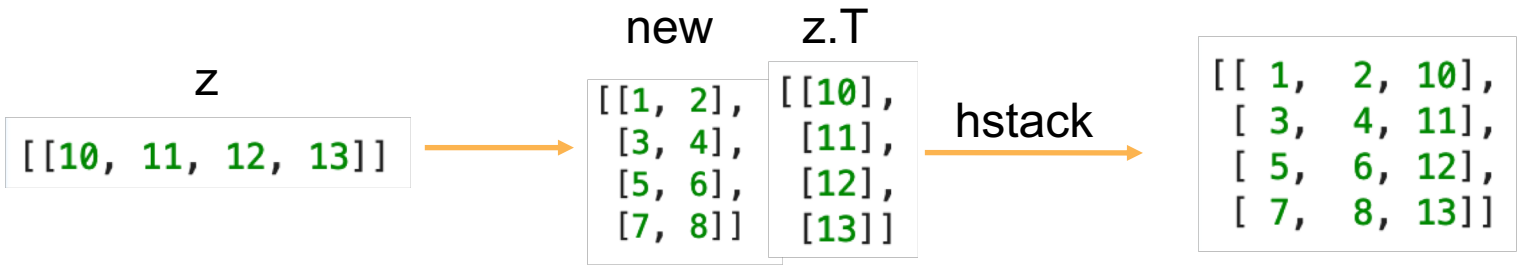

```
>>> x = np.array([[1,2],[3,4],[5,6]])
>>> y = np.array([[7,8]])
>>> new = np.concatenate((x,y), axis=0)
>>> new
array([[1, 2], [3, 4], [5, 6], [7, 8]])

>>> z = np.array([[10,11,12,13]])
>>> np.concatenate((new,z.T), axis=1)
array([[ 1, 2, 10], [ 3, 4, 11], [ 5, 6, 12], [ 7, 8, 13]])
```



Join & Split

- Vstack (row wise) & Hstack (column wise)

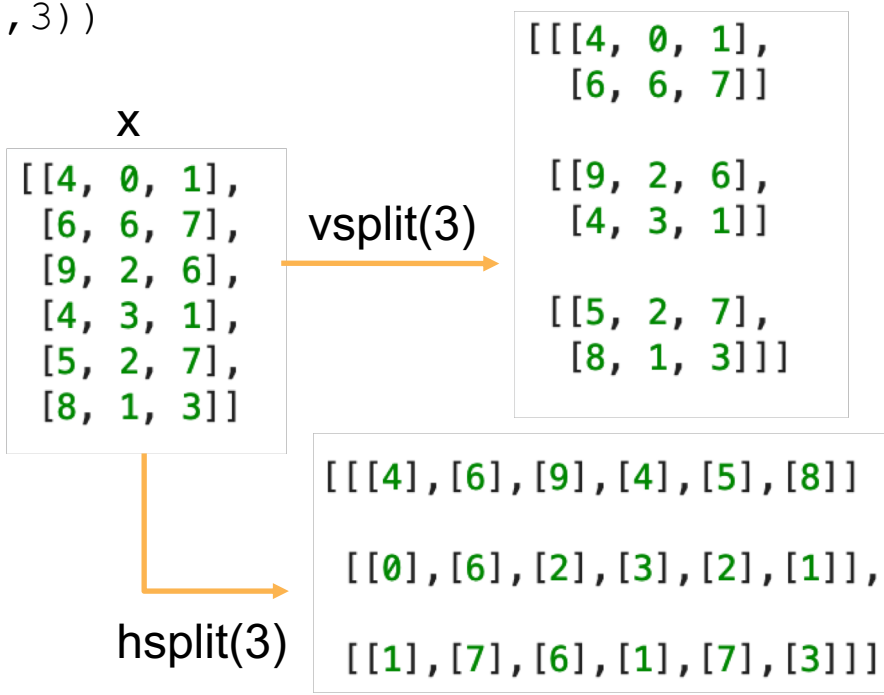

```
>>> x = np.array([[1,2],[3,4],[5,6]])
>>> y = np.array([[7,8]])
>>> new = np.vstack((x,y))     #axis=0
>>> new
array([[1, 2], [3, 4], [5, 6], [7, 8]])

>>> z = np.array([[10,11,12,13]])
>>> np.hstack((new,z.T))       #axis=1
array([[ 1, 2, 10], [ 3, 4, 11], [ 5, 6, 12], [ 7, 8, 13]])
```



Join & Split

- Vsplit (split vertical evenly) & Hsplit (split horizontal evenly)

```
>>> x = np.random.randint(10, size=(6,3))
array([[4, 0, 1], [6, 6, 7],
       [9, 2, 6], [4, 3, 1],
       [5, 2, 7], [8, 1, 3]])
>>> print(np.vsplit(x,3))
[array([[4, 0, 1], [6, 6, 7]]),
 array([[9, 2, 6], [4, 3, 1]]),
 array([[5, 2, 7], [8, 1, 3]])]
>>> print(np.hsplit(x,3))
[array([[4],[6],[9],[4],[5],[8]]),
 array([[0],[6],[2],[3],[2],[1]]),
 array([[1],[7],[6],[1],[7],[3]])]
```



Inspecting Your Array

- **Shape**: Array dimensions
- **Size**: Number of array elements
- **Len**: Length of array
- **Dtype**: Data type of array elements
- **Astype**: Convert an array to a different type

```
>>> x = np.random.randint(10, size=(6,3))
array([[4, 0, 1], [6, 6, 7], [9, 2, 6],
       [4, 3, 1], [5, 2, 7], [8, 1, 3]])
>>> x.shape
(6, 3)
>>> x.size
18
>>> len(x)
6
>>> x.dtype
dtype('int64')
>>> x.astype('float')
```

x

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

Shape: (6,3)
Size: 18
Len: 6
Dtype: int64

NumPy Cheatsheet

Python For Data Science *Cheat Sheet*

NumPy Basics

Learn Python for Data Science Interactively at www.DataCamp.com

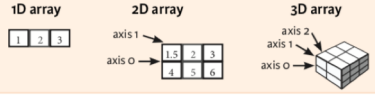
NumPy

The **NumPy** library is the core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.

Use the following import convention:

```
>>> import numpy as np
```

NumPy Arrays



Creating Arrays

```
>>> a = np.array([1,2,3])
>>> b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
>>> c = np.array([[(1.5,2,3), (4,5,6)], [(3,2,1), (4,5,6)]],
                 dtype = float)
```

Initial Placeholders

```
>>> np.zeros((3,4))                  Create an array of zeros
>>> np.ones((2,3,4),dtype=np.int16)  Create an array of ones
>>> d = np.arange(10,25,5)           Create an array of evenly
                                     spaced values (step value)
>>> np.linspace(0,2,9)               Create an array of evenly
                                     spaced values (number of samples)
>>> e = np.full((2,2),7)             Create a constant array
>>> f = np.eye(2)                    Create a 2X2 identity matrix
>>> np.random.random((2,2))          Create an array with random values
>>> np.empty((3,2))                  Create an empty array
```

I/O

Saving & Loading On Disk

```
>>> np.save('my_array', a)
>>> np.savez('array.npz', a, b)
>>> np.load('my_array.npy')
```

Saving & Loading Text Files

```
>>> np.loadtxt("myfile.txt")
>>> np.genfromtxt("my_file.csv", delimiter=',')
>>> np.savetxt("myarray.txt", a, delimiter=" ")
```

Data Types

```
>>> np.int64       Signed 64-bit integer types
>>> np.float32     Standard double-precision floating point
>>> np.complex     Complex numbers represented by 128 floats
>>> np.bool        Boolean type storing TRUE and FALSE values
>>> np.object      Python object type
>>> np.string_     Fixed-length string type
>>> np.unicode_    Fixed-length unicode type
```

Inspecting Your Array

```
>>> a.shape          Array dimensions
>>> len(a)           Length of array
>>> b.ndim           Number of array dimensions
>>> e.size           Number of array elements
>>> b.dtype          Data type of array elements
>>> b.dtype.name     Name of data type
>>> b.astype(int)    Convert an array to a different type
```

Asking For Help

```
>>> np.info(np.ndarray.dtype)
```

Array Mathematics

Arithmetic Operations

```
>>> g = a - b                    Subtraction
  array([[-0.5,  0. ,  0. ],
         [-3. , -3. , -3. ]])
>>> np.subtract(a,b)             Subtraction
>>> b + a                        Addition
  array([[ 2.5,  4. ,  6. ],
         [ 5. ,  7. ,  9. ]])
>>> np.add(b,a)                  Addition
>>> a / b                        Division
  array([[ 0.66666667,  1.        ,  1.        ],
         [ 0.25      ,  0.4       ,  0.5       ]])
>>> np.divide(a,b)               Division
>>> a * b                        Multiplication
  array([[  1.5,   4. ,   9. ],
         [  4. ,  10. ,  18. ]])
>>> np.multiply(a,b)             Multiplication
>>> np.exp(b)                    Exponentiation
>>> np.sqrt(b)                   Square root
>>> np.sin(a)                    Print sines of an array
>>> np.cos(b)                    Element-wise cosine
>>> np.log(a)                    Element-wise natural logarithm
>>> e.dot(f)                     Dot product
  array([[ 7.,  7.],
         [ 7.,  7.]])
```

Comparison

```
>>> a == b                       Element-wise comparison
  array([[False,  True,  True],
         [False, False, False]], dtype=bool)
>>> a < 2                        Element-wise comparison
  array([True, False, False], dtype=bool)
>>> np.array_equal(a, b)         Array-wise comparison
```

Aggregate Functions

```
>>> a.sum()                      Array-wise sum
>>> a.min()                      Array-wise minimum value
>>> b.max(axis=0)                Maximum value of an array row
>>> b.cumsum(axis=1)             Cumulative sum of the elements
>>> a.mean()                     Mean
>>> b.median()                   Median
>>> a.corrcoef()                 Correlation coefficient
>>> np.std(b)                    Standard deviation
```

Copying Arrays

```
>>> h = a.view()                 Create a view of the array with the same data
>>> np.copy(a)                   Create a copy of the array
>>> h = a.copy()                 Create a deep copy of the array
```

Sorting Arrays

```
>>> a.sort()                     Sort an array
>>> c.sort(axis=0)               Sort the elements of an array's axis
```

Subsetting, Slicing, Indexing

Also see Lists

Subsetting

```
>>> a[2]                         Select the element at the 2nd index
  3
>>> b[1,2]                       Select the element at row 1 column 2
  6.0                            (equivalent to b[1][2])
```

Slicing

```
>>> a[0:2]                       Select items at index 0 and 1
  array([1, 2])
>>> b[0:2,1]                     Select items at rows 0 and 1 in column 1
  array([ 2.,  5.])
>>> b[:1]                        Select all items at row 0
  array([[1.5, 2., 3.]])         (equivalent to b[0:1, :])
>>> c[1,...]                     Same as [1,:,:]
  array([[[ 3.,  2.,  1.],
          [ 4.,  5.,  6.]]])
>>> a[ : :-1]                    Reversed array a
  array([3, 2, 1])
```

Boolean Indexing

```
>>> a[a<2]                       Select elements from a less than 2
  array([1])
```

Fancy Indexing

```
>>> b[[1, 0, 1, 0],[0, 1, 2, 0]]  Select elements (1,0),(0,1),(1,2) and (0,0)
  array([ 4. , 2. , 6. , 1.5])
>>> b[[1, 0, 1, 0]][:,[0,1,2,0]]  Select a subset of the matrix's rows
  array([[ 4. ,  5. ,  6. ,  4. ],   and columns
         [ 1.5,  2. ,  3. ,  1.5],
         [ 4. ,  5. ,  6. ,  4. ],
         [ 1.5,  2. ,  3. ,  1.5]])
```

Array Manipulation

Transposing Array

```
>>> i = np.transpose(b)          Permute array dimensions
>>> i.T                          Permute array dimensions
```

Changing Array Shape

```
>>> b.ravel()                    Flatten the array
>>> g.reshape(3,-2)              Reshape, but don't change data
```

Adding/Removing Elements

```
>>> h.resize((2,6))              Return a new array with shape (2,6)
>>> np.append(h,g)               Append items to an array
>>> np.insert(a, 1, 5)           Insert items in an array
>>> np.delete(a,[1])             Delete items from an array
```

Combining Arrays

```
>>> np.concatenate((a,d),axis=0)  Concatenate arrays
  array([ 1,  2,  3, 10, 15, 20])
>>> np.vstack((a,b))             Stack arrays vertically (row-wise)
  array([[ 1. ,  2. ,  3. ],
         [ 1.5,  2. ,  3. ],
         [ 4. ,  5. ,  6. ]])
>>> np.r_[e,f]                   Stack arrays vertically (row-wise)
>>> np.hstack((e,f))             Stack arrays horizontally (column-wise)
  array([[ 7.,  7.,  1.,  0.],
         [ 7.,  7.,  0.,  1.]])
>>> np.column_stack((a,d))       Create stacked column-wise arrays
  array([[ 1, 10],
         [ 2, 15],
         [ 3, 20]])
>>> np.c_[a,d]                   Create stacked column-wise arrays
```

Splitting Arrays

```
>>> np.hsplit(a,3)               Split the array horizontally at the 3rd
  [array([1]),array([2]),array([3])]  index
>>> np.vsplit(c,2)               Split the array vertically at the 2nd index
[array([[[ 1.5,  2. ,  1. ],
         [ 4. ,  5. ,  6. ]]]),
 array([[[ 3.,  2.,  3.],
         [ 4.,  5.,  6.]]])]
```

Pandas: High-Performance Data Manipulation Tools

Introduction

Pandas

- Pandas is used for data manipulation, analysis and cleaning. Python pandas is well suited for different kinds of data.

- Pandas provides high-level data structures and functions designed to make working with structured or tabular data fast, easy, and expressive.

- The primary objects in pandas that will be used today are the DataFrame, a tabular, column-oriented data structure with both row and column labels, and the Series, a one-dimensional labeled array object.

- For users of the R language for statistical computing, the DataFrame name will be familiar, as the object was named after the similar R data.frame object

Pandas – Data Frame & Series

- Pandas DataFrames extend NumPy two-dimensional arrays by giving labels to the columns and also to the rows. **(Spreadsheet-like)**

- Pandas DataFrame column has the same data type, but the row ("record") datatype may be heterogenous (a tuple of different types), while the column datatype must be homogenous. **(heterogenous)**

- Pandas Series is a **one-dimensional labeled array** capable of holding any data type with axis labels or index. An example of a Series object is one column from a DataFrame.

VS NumPy Array

- A numpy array (ndarray) is a multidimensional array type containing items of the **same type and size. (Numpy array is homogeneous)**

Loading Pandas Library

In [ ]:
```
#Import Python Libraries
import pandas as pd
```

Press `Shift+Enter` to execute the *jupyter* cell

Pandas - Reading Data

- In order to read csv data, we'll need to use the **pandas.read_csv** function. This function will take in a csv file and return a DataFrame.

```
#Read csv file
df = pd.read_csv("data/salaries.csv")
```

Note: The above command has many optional arguments to fine-tune the data import process.

- There is a number of pandas commands to read other data formats:

```
pd.read_excel('myfile.xlsx', 'Sheet1', index_col=None, na_values=['NA'])
pd.read_stata('myfile.dta')
pd.read_sas('myfile.sas7bdat')
pd.read_hdf('myfile.h5','df')
etc..
```

Pandas - Exploring Data Frames

- We'll use the head method to see what's in the first N rows of a DataFrame.
- Use the tail method to see what's in the first N rows of a DataFrame.

```
#List first 5 records
df.head()
```

```
#List last 5 records
df.tail()
```

	sepal length	sepal width	petal length	petal width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

	sepal length	sepal width	petal length	petal width	class
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Data Frames - Data Types

- We can check data types for all the columns or a particular column.

```
#Check a particular column type
df['sepal length'].dtypes
```

```
dtype('float64')
```

```
#Check types for all the columns
df.dtypes
```

```
sepal length    float64
sepal width     float64
petal length    float64
petal width     float64
class            object
dtype: object
```

Data Frames (Python) - Attibutes

- Python objects have attributes and methods.

df.attribute	description
dtypes	list the types of the columns
columns	list the column names
axes	list the row labels and column names
ndim	number of dimensions
size	number of elements
shape	return a tuple representing the dimensionality
values	numpy representation of the data

Data Frames – Basic Methods

- All attributes and methods can be listed with a dir() function: dir(df)

df.method()	description
head(n), tail(n)	first/last n rows
describe()	generate descriptive statistics (for numeric columns only)
max(), min()	return max/min values for all numeric columns
mean(), median()	return mean/median values for all numeric columns
std()	standard deviation
sample(n)	returns a random sample of the data frame
dropna()	drop all the records with missing values

Data Frame – *groupby* Method

- Using "group by" method we can:
  - Split the data into groups based on some criteria
  - Calculate statistics (or apply a function) to each group

```
#Group data using class
df_class = df.groupby(['class'])
```

```
#Calculate mean value for each numeric column per each group
df_class.mean()
```

class	sepal length	sepal width	petal length	petal width
Iris-setosa	5.006	3.418	1.464	0.244
Iris-versicolor	5.936	2.770	4.260	1.326
Iris-virginica	6.588	2.974	5.552	2.026

Data Frame – *groupby* Method

- Once groupby object is create we can calculate various statistics for each group:

```
#Group data using class
df.groupby('class')[['sepal length']].mean()
```

class	sepal length
Iris-setosa	5.006
Iris-versicolor	5.936
Iris-virginica	6.588

Note: If single brackets are used to specify the column (e.g. sepal length), then the output is Pandas Series object. When double brackets are used the output is a Data Frame

Data Frame – Filtering

- To subset the data we can apply Boolean indexing. This indexing is commonly known as a filter. For example if we want to subset the rows in which the sepal length value is greater than 5.0 and sepal width value is greater than 3.5.

```
#View data with filtering
df_length = df[(df['sepal length']>5)&(df['sepal width']>3.5)]
```

- Any Boolean operator can be used to subset the data:
  - `>` greater; `>=` greater or equal;
  - `<` less; `<=` less or equal;
  - `==` equal; `!=` not equal;

```
#View data with filtering
df_setosa = df[df['class'] == 'Iris-setosa']
```

Data Frame - Slicing

- There are a number of ways to subset the Data Frame:
  - one or more columns
  - one or more rows
  - a subset of rows and columns

- Rows and columns can be selected by their position or label (iloc, loc)

Data Frame - Slicing

- When selecting one column, it is possible to use single set of brackets, but the resulting object will be a Series (not a DataFrame):

```
#Select column sepal length:
df['sepal length']
```

- When we need to select more than one column and/or make the output to be a DataFrame, we should use double brackets:

```
#Select column sepal length:
df[['sepal length','class']]
```

Data Frame – Selecting Rows

- If we need to select a range of rows, we can specify the range using ":"

```
#Select rows by their position:
df[10:20]
```

- Notice that the first row has a position 0, and the last value in the range is omitted:

- So for 0:10 range the first 10 rows are returned with the positions starting with 0 and ending with 9

Data Frame – Method *loc*

- If we need to select a range of rows, using their labels we can use method loc:

```
#Select rows by their labels:
df.loc[10:15,['sepal length','class']]
```

	sepal length	class
10	5.4	Iris-setosa
11	4.8	Iris-setosa
12	4.8	Iris-setosa
13	4.3	Iris-setosa
14	5.8	Iris-setosa
15	5.7	Iris-setosa

Data Frame – Method *iloc*

- If we need to select a range of rows and/or columns, using their positions we can use method iloc:

```
#Select rows by their labels:
df.iloc[10:15,[0, 1, 4]]
```

	sepal length	sepal width	class
10	5.4	3.7	Iris-setosa
11	4.8	3.4	Iris-setosa
12	4.8	3.0	Iris-setosa
13	4.3	3.0	Iris-setosa
14	5.8	4.0	Iris-setosa

Data Frame – Method *iloc* (summary)

```
df.iloc[0]  # First row of a data frame
df.iloc[i]  #(i+1)th row
df.iloc[-1] # Last row

df.iloc[:, 0]  # First column
df.iloc[:, -1] # Last column

df.iloc[0:7]       #First 7 rows
df.iloc[:, 0:2]    #First 2 columns
df.iloc[1:3, 0:2]  #Second through third rows and first 2 co
lumns
df.iloc[[0,5], [1,3]] #1st and 6th rows and 2nd and 4th columns
```

Data Frame – Sorting

- We can sort the data by a value in the column. By default the sorting will occur in ascending order and a new data frame is return.

```
# Create a new data frame from the original sorted by the column sepal length
df_sorted = df.sort_values( by ='sepal length')
df_sorted.head()
```

	sepal length	sepal width	petal length	petal width	class
13	4.3	3.0	1.1	0.1	Iris-setosa
42	4.4	3.2	1.3	0.2	Iris-setosa
38	4.4	3.0	1.3	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
41	4.5	2.3	1.3	0.3	Iris-setosa

Data Frame – Sorting

- We can sort the data using 2 or more columns:

```
df_sorted = df.sort_values(by =['sepal length','petal length'],
                           ascending = [True, True])
df_sorted.head()
```

	sepal length	sepal width	petal length	petal width	class
13	4.3	3.0	1.1	0.1	Iris-setosa
38	4.4	3.0	1.3	0.2	Iris-setosa
42	4.4	3.2	1.3	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
41	4.5	2.3	1.3	0.3	Iris-setosa

Pandas – Missing Values

- Missing values are marked as NaN

```
# Read a dataset with missing values
df = pd.read_csv("data/titanic_data.csv")
```

```
# Select the rows that have at least one missing value
df[df.isnull().any(axis=1)].head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
5	6	0	3	Moran, Mr. James	male	NaN	0	0	330877	8.4583	NaN	Q
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3	1	349909	21.0750	NaN	S

Pandas – Missing Values

- There are a number of methods to deal with missing values in the data frame:

df.method()	description
dropna()	Drop missing observations
dropna(how='all')	Drop observations where all cells is NA
dropna(axis=1, how='all')	Drop column if all the values are missing
dropna(thresh = 5)	Drop rows that contain less than 5 non-missing values
fillna(0)	Replace missing values with zeros
isnull()	returns True if the value is missing
notnull()	Returns True for non-missing values

Pandas – Missing Values

- When summing the data, missing values will be treated as zero
- If all values are missing, the sum will be equal to NaN
- cumsum() and cumprod() methods ignore missing values but preserve them in the resulting arrays
- Missing values in GroupBy method are excluded (just like in R)
- Many descriptive statistics methods have skipna option to control if missing data should be excluded . This value is set to True by default (unlike R)

Pandas – Aggregation Functions

- Aggregation - computing a summary statistic about each group, i.e.
  - compute group sums or means
  - compute group sizes/counts

- Common aggregation functions:
  - min, max
  - count, sum, prod
  - mean, median, mode, mad
  - std, var

Pandas – Aggregation Functions

- agg() method are useful when multiple statistics are computed per column:

```
df[['Age','Fare']].agg(['min','mean','max'])
```

	Age	Fare
min	0.420000	0.000000
mean	29.699118	32.204208
max	80.000000	512.329200

Pandas – Basic Decriptive Statistics

df.method()	description
describe	Basic statistics (count, mean, std, min, quantiles, max)
min, max	Minimum and maximum values
mean, median, mode	Arithmetic average, median and mode
var, std	Variance and standard deviation
sem	Standard error of mean
skew	Sample skewness
kurt	kurtosis

Pandas Cheatsheet

Python For Data Science *Cheat Sheet*

Pandas Basics

Learn Python for Data Science Interactively at www.DataCamp.com

Pandas

The **Pandas** library is built on NumPy and provides easy-to-use **data structures** and **data analysis** tools for the Python programming language.

Use the following import convention:

```
>>> import pandas as pd
```

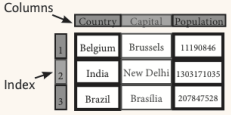
Pandas Data Structures

Series

A **one-dimensional** labeled array capable of holding any data type


```
>>> s = pd.Series([3, -5, 7, 4], index=['a', 'b', 'c', 'd'])
```

DataFrame



A **two-dimensional** labeled data structure with columns of potentially different types

```
>>> data = {'Country': ['Belgium', 'India', 'Brazil'],
            'Capital': ['Brussels', 'New Delhi', 'Brasilia'],
            'Population': [11190846, 1303171035, 207847528]}

>>> df = pd.DataFrame(data,
                      columns=['Country', 'Capital', 'Population'])
```

Asking For Help

```
>>> help(pd.Series.loc)
```

Selection

Also see NumPy Arrays

Getting

```
>>> s['b']
 -5

>>> df[1:]
   Country    Capital  Population
1   India  New Delhi  1303171035
2  Brazil   Brasilia  207847528
```

Get one element

Get subset of a DataFrame

Selecting, Boolean Indexing & Setting

By Position

```
>>> df.iloc([0],[0])
'Belgium'
>>> df.iat([0],[0])
'Belgium'
```

Select single value by row & column

By Label

```
>>> df.loc([0], ['Country'])
'Belgium'
>>> df.at([0], ['Country'])
'Belgium'
```

Select single value by row & column labels

By Label/Position

```
>>> df.ix[2]
Country      Brazil
Capital    Brasilia
Population  207847528
```

Select single row of subset of rows

```
>>> df.ix[:,'Capital']
0     Brussels
1    New Delhi
2     Brasilia
```

Select a single column of subset of columns

```
>>> df.ix[1,'Capital']
'New Delhi'
```

Select rows and columns

Boolean Indexing

```
>>> s[~(s > 1)]
>>> s[(s < -1) | (s > 2)]
>>> df[df['Population']>1200000000]
```

Series `s` where value is not >1
`s` where value is <-1 or >2
Use filter to adjust DataFrame

Setting

```
>>> s['a'] = 6
```

Set index `a` of Series `s` to 6

Dropping

```
>>> s.drop(['a', 'c'])
>>> df.drop('Country', axis=1)
```

Drop values from rows (axis=0)
Drop values from columns(axis=1)

Sort & Rank

```
>>> df.sort_index(by='Country')
>>> s.order()
>>> df.rank()
```

Sort by row or column index
Sort a series by its values
Assign ranks to entries

Retrieving Series/DataFrame Information

Basic Information

```
>>> df.shape
>>> df.index
>>> df.columns
>>> df.info()
>>> df.count()
```

(rows,columns)
Describe index
Describe DataFrame columns
Info on DataFrame
Number of non-NA values

Summary

```
>>> df.sum()
>>> df.cumsum()
>>> df.min()/df.max()
>>> df.idmin()/df.idmax()
>>> df.describe()
>>> df.mean()
>>> df.median()
```

Sum of values
Cummulative sum of values
Minimum/maximum values
Minimum/Maximum index value
Summary statistics
Mean of values
Median of values

Applying Functions

```
>>> f = lambda x: x*2
>>> df.apply(f)
>>> df.applymap(f)
```

Apply function
Apply function element-wise

Data Alignment

Internal Data Alignment

NA values are introduced in the indices that don't overlap:

```
>>> s3 = pd.Series([7, -2, 3], index=['a', 'c', 'd'])
>>> s + s3
  a     10.0
  b      NaN
  c      5.0
  d      7.0
```

Arithmetic Operations with Fill Methods

You can also do the internal data alignment yourself with the help of the fill methods:

```
>>> s.add(s3, fill_value=0)
  a    10.0
  b    -5.0
  c     5.0
  d     7.0
>>> s.sub(s3, fill_value=2)
>>> s.div(s3, fill_value=4)
>>> s.mul(s3, fill_value=3)
```

I/O

Read and Write to CSV

```
>>> pd.read_csv('file.csv', header=None, nrows=5)
>>> pd.to_csv('myDataFrame.csv')
```

Read and Write to Excel

```
>>> pd.read_excel('file.xlsx')
>>> pd.to_excel('dir/myDataFrame.xlsx', sheet_name='Sheet1')
```

Read multiple sheets from the same file

```
>>> xlsx = pd.ExcelFile('file.xls')
>>> df = pd.read_excel(xlsx, 'Sheet1')
```

Read and Write to SQL Query or Database Table

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///:memory:')
>>> pd.read_sql("SELECT * FROM my_table;", engine)
>>> pd.read_sql_table('my_table', engine)
>>> pd.read_sql_query("SELECT * FROM my_table;", engine)
```

`read_sql()` is a convenience wrapper around `read_sql_table()` and `read_sql_query()`

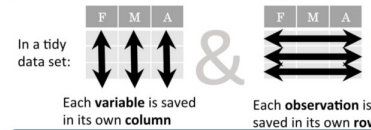
```
>>> pd.to_sql('myDf', engine)
```

Pandas Cheatsheet

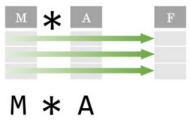
Data Wrangling with pandas Cheat Sheet

http://pandas.pydata.org

Tidy Data – A foundation for wrangling in pandas



Tidy data complements pandas's vectorized operations. pandas will automatically preserve observations as you manipulate variables. No other format works as intuitively with pandas.



Syntax – Creating DataFrames

	a	b	c
1	4	7	10
2	5	8	11
3	6	9	12

```
df = pd.DataFrame(
        {"a" : [4 ,5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
       index = [1, 2, 3])
```
Specify values for each column.

```
df = pd.DataFrame(
    [[4, 7, 10],
     [5, 8, 11],
     [6, 9, 12]],
    index=[1, 2, 3],
    columns=['a', 'b', 'c'])
```
Specify values for each row.

		a	b	c
n	v			
d	1	4	7	10
	2	5	8	11
e	2	6	9	12

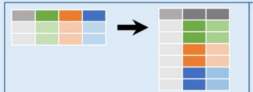
```
df = pd.DataFrame(
        {"a" : [4 ,5, 6],
         "b" : [7, 8, 9],
         "c" : [10, 11, 12]},
index = pd.MultiIndex.from_tuples(
        [('d',1),('d',2),('e',2)],
          names=['n','v'])))
```
Create DataFrame with a MultiIndex

Method Chaining

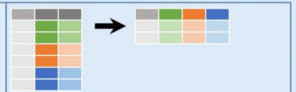
Most pandas methods return a DataFrame so that another pandas method can be applied to the result. This improves readability of code.

```
df = (pd.melt(df)
        .rename(columns={
                'variable' : 'var',
                'value' : 'val'})
        .query('val >= 200')
      )
```

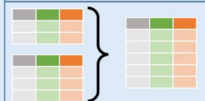
Reshaping Data – Change the layout of a data set



pd.melt(df)
Gather columns into rows.



df.pivot(columns='var', values='val')
Spread rows into columns.



pd.concat([df1,df2])
Append rows of DataFrames

pd.concat([df1,df2], axis=1)
Append columns of DataFrames

df.sort_values('mpg')
Order rows by values of a column (low to high).

df.sort_values('mpg',ascending=False)
Order rows by values of a column (high to low).

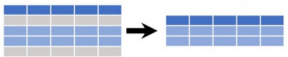
df.rename(columns = {'y':'year'})
Rename the columns of a DataFrame

df.sort_index()
Sort the index of a DataFrame

df.reset_index()
Reset index of DataFrame to row numbers, moving index to columns.

df.drop(['Length','Height'], axis=1)
Drop columns from DataFrame

Subset Observations (Rows)



df[df.Length > 7]
Extract rows that meet logical criteria.

df.drop_duplicates()
Remove duplicate rows (only considers columns).

df.head(n)
Select first n rows.

df.tail(n)
Select last n rows.

df.sample(frac=0.5)
Randomly select fraction of rows.

df.sample(n=10)
Randomly select n rows.

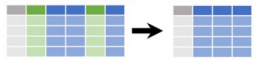
df.iloc[10:20]
Select rows by position.

df.nlargest(n, 'value')
Select and order top n entries.

df.nsmallest(n, 'value')
Select and order bottom n entries.

Logic in Python (and pandas)			
<	Less than	!=	Not equal to
>	Greater than	df.column.isin(*values*)	Group membership
==	Equals	pd.isnull(*obj*)	Is NaN
<=	Less than or equals	pd.notnull(*obj*)	Is not NaN
>=	Greater than or equals	&,|,~,^,df.any(),df.all()	Logical and, or, not, xor, any, all

Subset Variables (Columns)



df[['width','length','species']]
Select multiple columns with specific names.

df['width'] *or* **df.width**
Select single column with specific name.

df.filter(regex='*regex*')
Select columns whose name matches regular expression *regex*.

regex (Regular Expressions) Examples	
'\.'	Matches strings containing a period '.'
'Length$'	Matches strings ending with word 'Length'
'^Sepal'	Matches strings beginning with the word 'Sepal'
'^x[1-5]$'	Matches strings beginning with 'x' and ending with 1,2,3,4,5
'^(?!Species$).*'	Matches strings except the string 'Species'

df.loc[:,'x2':'x4']
Select all columns between x2 and x4 (inclusive).

df.iloc[:,[1,2,5]]
Select columns in positions 1, 2 and 5 (first column is 0).

df.loc[df['a'] > 10, ['a','c']]
Select rows meeting logical condition, and only the specific columns .

Let’s Dig Into Data!