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.

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

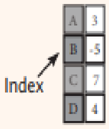
Use the following import convention:

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

Pandas Data Structures

Series

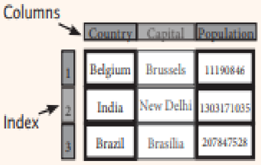
A **one-dimensional** labeled array capable of holding any data type



A	3
B	-5
C	7
D	4

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

DataFrame



	Country	Capital	Population
1	Belgium	Brussels	11190846
2	India	New Delhi	1303171035
3	Brazil	Brasilia	207847528

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
```

Get one element

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

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)]`	Series `s` where value is not >1
`>>> s[(s < -1) | (s > 2)]`	`s` where value is <-1 or >2
`>>> df[df['Population']>1200000000]`	Use filter to adjust DataFrame

Setting

`>>> s['a'] = 6`	Set index `a` of Series `s` to 6

Dropping

`>>> s.drop(['a', 'c'])`	Drop values from rows (axis=o)
`>>> df.drop('Country', axis=1)`	Drop values from columns(axis=1)

Sort & Rank

`>>> df.sort_index()`	Sort by labels along an axis
`>>> df.sort_values(by='Country')`	Sort by the values along an axis
`>>> df.rank()`	Assign ranks to entries

Retrieving Series/DataFrame Information

Basic Information

`>>> df.shape`	(rows,columns)
`>>> df.index`	Describe index
`>>> df.columns`	Describe DataFrame columns
`>>> df.info()`	Info on DataFrame
`>>> df.count()`	Number of non-NA values

Summary

`>>> df.sum()`	Sum of values
`>>> df.cumsum()`	Cummulative sum of values
`>>> df.min()/df.max()`	Minimum/maximum values
`>>> df.idxmin()/df.idxmax()`	Minimum/Maximum index value
`>>> df.describe()`	Summary statistics
`>>> df.mean()`	Mean of values
`>>> df.median()`	Median of values

Applying Functions

```
>>> f = lambda x: x*2
```

`>>> df.apply(f)`	Apply function
`>>> df.applymap(f)`	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)
```

DataCamp
Learn Python for Data Science Interactively

Python For Data Science *Cheat Sheet*

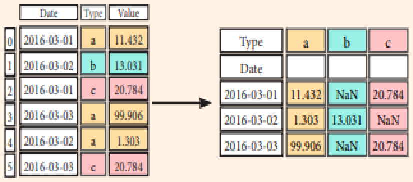
Pandas

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

Reshaping Data

Pivot

`>>> df3= df2.pivot(index='Date',`
`columns='Type',`
`values='Value')`
	Spread rows into columns



	Date	Type	Value
0	2016-03-01	a	11.432
1	2016-03-02	b	13.031
2	2016-03-01	c	20.784
3	2016-03-03	a	99.906
4	2016-03-02	a	1.303
5	2016-03-03	c	20.784

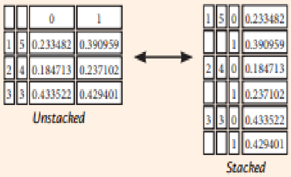
Type	a	b	c
Date			
2016-03-01	11.432	NaN	20.784
2016-03-02	1.303	13.031	NaN
2016-03-03	99.906	NaN	20.784

Pivot Table

`>>> df4 = pd.pivot_table(df2,`
`values='Value',`
`index='Date',`
`columns='Type'])`
	Spread rows into columns

Stack / Unstack

`>>> stacked = df5.stack()`	Pivot a level of column labels
`>>> stacked.unstack()`	Pivot a level of index labels



		0	1
1	5	0.233482	0.390959
2	4	0.184713	0.237102
3	3	0.433522	0.429401

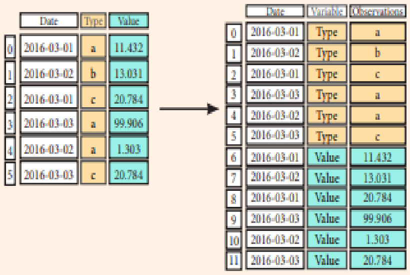
Unstacked

1	5	0	0.233482
		1	0.390959
2	4	0	0.184713
		1	0.237102
3	3	0	0.433522
		1	0.429401

Stacked

Melt

`>>> pd.melt(df2,`
`id_vars=["Date"],`
`value_vars=["Type", "Value"],`
`value_name="Observations")`
	Gather columns into rows



	Date	Type	Value
0	2016-03-01	a	11.432
1	2016-03-02	b	13.031
2	2016-03-01	c	20.784
3	2016-03-03	a	99.906
4	2016-03-02	a	1.303
5	2016-03-03	c	20.784

	Date	Variable	Observations
0	2016-03-01	Type	a
1	2016-03-02	Type	b
2	2016-03-01	Type	c
3	2016-03-03	Type	a
4	2016-03-02	Type	a
5	2016-03-03	Type	c
6	2016-03-01	Value	11.432
7	2016-03-02	Value	13.031
8	2016-03-01	Value	20.784
9	2016-03-03	Value	99.906
10	2016-03-02	Value	1.303
11	2016-03-03	Value	20.784

Iteration

`>>> df.iteritems()`	(Column-index, Series) pairs
`>>> df.iterrows()`	(Row-index, Series) pairs

Advanced Indexing

Also see NumPy Arrays

Selecting

`>>> df3.loc[:,(df3>1).any()]`	Select cols with any vals >1
`>>> df3.loc[:,(df3>1).all()]`	Select cols with vals > 1
`>>> df3.loc[:,df3.isnull().any()]`	Select cols with NaN
`>>> df3.loc[:,df3.notnull().all()]`	Select cols without NaN

Indexing With isin

`>>> df[(df.Country.isin(df2.Type))]`	Find same elements
`>>> df3.filter(items="a","b"])`	Filter on values
`>>> df.select(lambda x: not x%5)`	Select specific elements

Where

`>>> s.where(s > 0)`	Subset the data

Query

`>>> df6.query('second > first')`	Query DataFrame

Setting/Resetting Index

`>>> df.set_index('Country')`	Set the index
`>>> df4 = df.reset_index()`	Reset the index

`>>> df = df.rename(index=str,`
`columns={"Country":"cntry",`
`"Capital":"cptl",`
`"Population":"ppltn"})`
	Rename DataFrame

Reindexing

```
>>> s2 = s.reindex(['a','c','d','e','b'])
```

Forward Filling

```
>>> df.reindex(range(4),
               method='ffill')
    Country    Capital    Population
 0  Belgium    Brussels   11190846
 1    India  New Delhi    1303171035
 2   Brazil    Brasilia   207847528
 3   Brazil    Brasilia   207847528
```

Backward Filling

```
>>> s3 = s.reindex(range(5),
                   method='bfill')
 0    3
 1    3
 2    3
 3    3
 4    3
```

MultiIndexing

```
>>> arrays = [np.array([1,2,3]),
              np.array([5,4,3])]
>>> df5 = pd.DataFrame(np.random.rand(3, 2), index=arrays)
>>> tuples = list(zip(*arrays))
>>> index = pd.MultiIndex.from_tuples(tuples,
                                      names=['first', 'second'])
>>> df6 = pd.DataFrame(np.random.rand(3, 2), index=index)
>>> df2.set_index(["Date", "Type"])
```

Duplicate Data

`>>> s3.unique()`	Return unique values
`>>> df2.duplicated('Type')`	Check duplicates
`>>> df2.drop_duplicates('Type', keep='last')`	Drop duplicates
`>>> df.index.duplicated()`	Check index duplicates

Grouping Data

Aggregation

```
>>> df2.groupby(by=['Date','Type']).mean()
>>> df4.groupby(level=0).sum()
>>> df4.groupby(level=0).agg({'a':lambda x:sum(x)/len(x),
                              'b': np.sum})
```

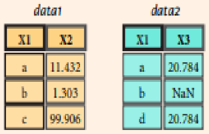
Transformation

```
>>> customSum = lambda x: (x+x%2)
>>> df4.groupby(level=0).transform(customSum)
```

Missing Data

`>>> df.dropna()`	Drop NaN values
`>>> df3.fillna(df3.mean())`	Fill NaN values with a predetermined value
`>>> df2.replace("a", "f")`	Replace values with others

Combining Data



data1

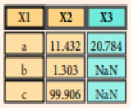
X1	X2
a	11.432
b	1.303
c	99.906

data2

X1	X3
a	20.784
b	NaN
d	20.784

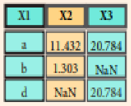
Merge

```
>>> pd.merge(data1,
             data2,
             how='left',
             on='X1')
```



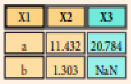
X1	X2	X3
a	11.432	20.784
b	1.303	NaN
c	99.906	NaN

```
>>> pd.merge(data1,
             data2,
             how='right',
             on='X1')
```



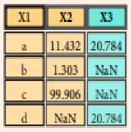
X1	X2	X3
a	11.432	20.784
b	1.303	NaN
d	NaN	20.784

```
>>> pd.merge(data1,
             data2,
             how='inner',
             on='X1')
```



X1	X2	X3
a	11.432	20.784
b	1.303	NaN

```
>>> pd.merge(data1,
             data2,
             how='outer',
             on='X1')
```



X1	X2	X3
a	11.432	20.784
b	1.303	NaN
c	99.906	NaN
d	NaN	20.784

Join

```
>>> data1.join(data2, how='right')
```

Concatenate

Vertical

```
>>> s.append(s2)
```

Horizontal/Vertical

```
>>> pd.concat([s,s2],axis=1, keys=['One','Two'])
>>> pd.concat([data1, data2], axis=1, join='inner')
```

Dates

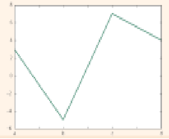
```
>>> df2['Date']= pd.to_datetime(df2['Date'])
>>> df2['Date']= pd.date_range('2000-1-1',
                               periods=6,
                               freq='M')
>>> dates = [datetime(2012,5,1), datetime(2012,5,2)]
>>> index = pd.DatetimeIndex(dates)
>>> index = pd.date_range(datetime(2012,2,1), end, freq='BM')
```

Visualization

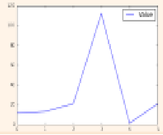
Also see Matplotlib

```
>>> import matplotlib.pyplot as plt
```

```
>>> s.plot()
>>> plt.show()
```



```
>>> df2.plot()
>>> plt.show()
```



DataCamp
Learn Python for Data Science Interactively

Python For Data Science *Cheat Sheet*

NumPy Basics

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

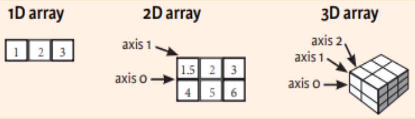
NumPy

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`
`array([[-0.5,  0. ,  0. ],`
`       [-3. , -3. , -3. ]])`
	Subtraction

`>>> np.subtract(a,b)`	Subtraction

`>>> b + a`
`array([[ 2.5,  4. ,  6. ],`
`       [ 5. ,  7. ,  9. ]])`
	Addition

`>>> np.add(b,a)`	Addition

`>>> a / b`
`array([[ 0.66666667, 1.        , 1.        ],`
`       [ 0.25      , 0.4       , 0.5       ]])`
	Division

`>>> np.divide(a,b)`	Division

`>>> a * b`
`array([[  1.5,   4. ,   9. ],`
`       [  4. ,  10. ,  18. ]])`
	Multiplication

`>>> 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)`
`array([[ 7.,  7.],`
`       [ 7.,  7.]])`
	Dot product

Comparison

`>>> a == b`
`array([[False,  True,  True],`
`       [False, False, False]], dtype=bool)`
	Element-wise comparison

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

`>>> 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

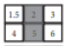
Subsetting

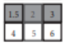
`>>> a[2]`
`3`
		Select the element at the 2nd index

`>>> b[1,2]`
`6.0`
		Select the element at row 0 column 2 (equivalent to `b[1][2]`)

Slicing

`>>> a[0:2]`
`array([1, 2])`
		Select items at index 0 and 1

`>>> b[0:2,1]`
`array([ 2.,  5.])`
		Select items at rows 0 and 1 in column 1

`>>> b[:1]`
`array([[1.5, 2., 3.]])`
		Select all items at row 0 (equivalent to `b[0:1, :]`)

`>>> c[1,...]`
`array([[[ 3.,  2.,  1.],`
`        [ 4.,  5.,  6.]]])`
		Same as `[1,:,:]`

`>>> a[ : :-1]`
`array([3, 2, 1])`
		Reversed array `a`

Boolean Indexing

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

Fancy Indexing

`>>> b[[1, 0, 1, 0],[0, 1, 2, 0]]`
`array([ 4. , 2. , 6. , 1.5])`
	Select elements `(1,0)`,`(0,1)`,`(1,2)` and `(0,0)`

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

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)`
`array([ 1,  2,  3, 10, 15, 20])`
	Concatenate arrays

`>>> np.vstack((a,b))`
`array([[ 1. ,  2. ,  3. ],`
`       [ 1.5,  2. ,  3. ],`
`       [ 4. ,  5. ,  6. ]])`
	Stack arrays vertically (row-wise)

`>>> np.r_[e,f]`	Stack arrays vertically (row-wise)

`>>> np.hstack((e,f))`
`array([[ 7.,  7.,  1.,  0.],`
`       [ 7.,  7.,  0.,  1.]])`
	Stack arrays horizontally (column-wise)

`>>> np.column_stack((a,d))`
`array([[ 1, 10],`
`       [ 2, 15],`
`       [ 3, 20]])`
	Create stacked column-wise arrays

`>>> np.c_[a,d]`	Create stacked column-wise arrays

Splitting Arrays

`>>> np.hsplit(a,3)`
`[array([1]),array([2]),array([3])]`
	Split the array horizontally at the 3rd index

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

DataCamp

Learn Python for Data Science Interactively