

# Data Wrangling

with pandas Cheat Sheet

<http://pandas.pydata.org>

[Pandas API Reference](#) [Pandas User Guide](#)

## 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

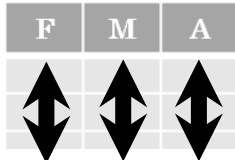
## Method Chaining

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'))
```


## Tidy Data – A foundation for wrangling in pandas

In a tidy data set:



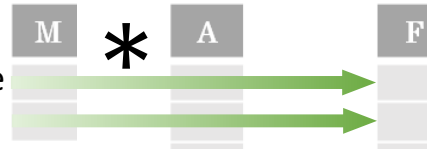
Each **variable** is saved in its own **column**

&



Each **observation** is saved in its own **row**

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



M \* A = F

## Reshaping Data – Change layout, sorting, reindexing, renaming





```
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(columns=['Length', 'Height'])
```

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.sample(frac=0.5)
```

Randomly select fraction of rows.

```
df.sample(n=10)
```

Randomly select n rows.

```
df.nlargest(n, 'value')
```

Select and order top n entries.

```
df.nsmallest(n, 'value')
```

Select and order bottom n entries.

```
df.head(n)
```

Select first n rows.

```
df.tail(n)
```

Select last n rows.

### 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*.

### Using query

`query()` allows Boolean expressions for filtering rows.

```
df.query('Length > 7')
```

```
df.query('Length > 7 and Width < 8')
```

```
df.query('Name.str.startswith("abc")', engine="python")
```

### Subsets - rows and columns

Use `df.loc[]` and `df.iloc[]` to select only rows, only columns or both.

Use `df.at[]` and `df.iat[]` to access a single value by row and column.

First index selects rows, second index columns.

```
df.iloc[10:20]
```

Select rows 10-20.

```
df.iloc[:, [1, 2, 5]]
```

Select columns in positions 1, 2 and 5 (first column is 0).

```
df.loc[:, 'x2': 'x4']
```

Select all columns between x2 and x4 (inclusive).

```
df.loc[df['a'] > 10, ['a', 'c']]
```

Select rows meeting logical condition, and only the specific columns.

```
df.iat[1, 2]
```

Access single value by index

```
df.at[4, 'A']
```

Access single value by label

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

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'

## Group Data



**df.groupby(by="col")**  
Return a GroupBy object, grouped by values in column named "col".

**df.groupby(level="ind")**  
Return a GroupBy object, grouped by values in index level named "ind".

All of the summary functions listed above can be applied to a group. Additional GroupBy functions:

**size()**  
Size of each group.

**agg(function)**  
Aggregate group using function.

## Summarize Data

**df['w'].value\_counts()**  
Count number of rows with each unique value of variable

**len(df)**  
# of rows in DataFrame.

**df.shape**  
Tuple of # of rows, # of columns in DataFrame.

**df['w'].nunique()**  
# of distinct values in a column.

**df.describe()**  
Basic descriptive and statistics for each column (or GroupBy).

**df.info()**  
Prints a concise summary of the DataFrame.

**df.memory\_usage()**  
Prints the memory usage of each column in the DataFrame.

**df.dtypes()**  
Prints a Series with the dtype of each column in the DataFrame.



pandas provides a large set of **summary functions** that operate on different kinds of pandas objects (DataFrame columns, Series, GroupBy, Expanding and Rolling (see below)) and produce single values for each of the groups. When applied to a DataFrame, the result is returned as a pandas Series for each column. Examples:

**sum()**  
Sum values of each object.

**count()**  
Count non-NA/null values of each object.

**median()**  
Median value of each object.

**quantile([0.25, 0.75])**  
Quantiles of each object.

**apply(function)**  
Apply function to each object.

**min()**  
Minimum value in each object.

**max()**  
Maximum value in each object.

**mean()**  
Mean value of each object.

**var()**  
Variance of each object.

**std()**  
Standard deviation of each object.

The examples below can also be applied to groups. In this case, the function is applied on a per-group basis, and the returned vectors are of the length of the original DataFrame.

**shift(1)**  
Copy with values shifted by 1.

**rank(method='dense')**  
Ranks with no gaps.

**rank(method='min')**  
Ranks. Ties get min rank.

**rank(pct=True)**  
Ranks rescaled to interval [0, 1].

**rank(method='first')**  
Ranks. Ties go to first value.

**shift(-1)**  
Copy with values lagged by 1.

**cumsum()**  
Cumulative sum.

**cummax()**  
Cumulative max.

**cummin()**  
Cumulative min.

**cumprod()**  
Cumulative product.

## Handling Missing Data

**df.dropna()**  
Drop rows with any column having NA/null data.

**df.fillna(value)**  
Replace all NA/null data with value.

## Make New Columns



**df.assign(Area=lambda df: df.Length\*df.Height)**  
Compute and append one or more new columns.

**df['Volume'] = df.Length\*df.Height\*df.Depth**  
Add single column.

**pd.qcut(df.col, n, labels=False)**  
Bin column into n buckets.



pandas provides a large set of **vector functions** that operate on all columns of a DataFrame or a single selected column (a pandas Series). These functions produce vectors of values for each of the columns, or a single Series for the individual Series. Examples:

**max(axis=1)**  
Element-wise max.

**clip(lower=-10, upper=10)**  
Trim values at input thresholds

**min(axis=1)**  
Element-wise min.

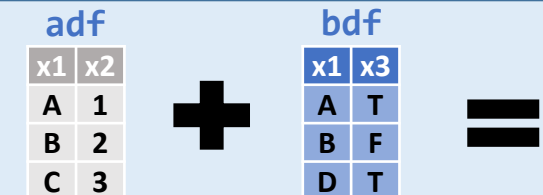
**abs()**  
Absolute value.

## Windows

**df.expanding()**  
Return an Expanding object allowing summary functions to be applied cumulatively.

**df.rolling(n)**  
Return a Rolling object allowing summary functions to be applied to windows of length n.

## Combine Data Sets



### Standard Joins

**pd.merge(adf, bdf, how='left', on='x1')**  
Join matching rows from bdf to adf.

**pd.merge(adf, bdf, how='right', on='x1')**  
Join matching rows from adf to bdf.

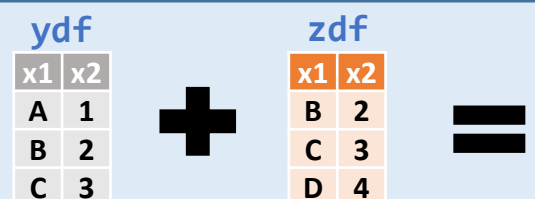
**pd.merge(adf, bdf, how='inner', on='x1')**  
Join data. Retain only rows in both sets.

**pd.merge(adf, bdf, how='outer', on='x1')**  
Join data. Retain all values, all rows.

### Filtering Joins

**adf[adf.x1.isin(bdf.x1)]**  
All rows in adf that have a match in bdf.

**adf[~adf.x1.isin(bdf.x1)]**  
All rows in adf that do not have a match in bdf.



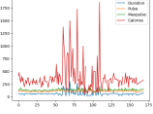
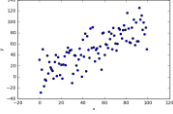
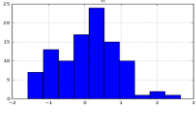

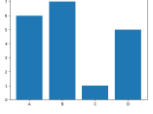
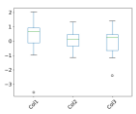
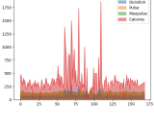
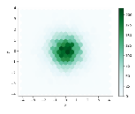
### Set-like Operations

**pd.merge(ydf, zdf)**  
Rows that appear in both ydf and zdf (Intersection).

**pd.merge(ydf, zdf, how='outer')**  
Rows that appear in either or both ydf and zdf (Union).

**pd.merge(ydf, zdf, how='outer', indicator=True)**  
**.query('\_merge == "left\_only"')**  
**.drop(columns=['\_merge'])**  
Rows that appear in ydf but not zdf (Setdiff).

# Plotting

<b>df.plot()</b> Plot a line graph for the DataFrame. 	<b>df.plot.scatter(x='w', y='h')</b> Plot a scatter graph of the DataFrame. 	<b>df.plot.hist()</b> Plot a histogram of the DataFrame. 	<b>df.plot.pie()</b> Plot a pie chart of the DataFrame. 
<b>df.plot.bar()</b> Plot a line graph for the DataFrame. 	<b>df.plot.boxplot()</b> Plot a scatter graph of the DataFrame. 	<b>df.plot.area()</b> Plot an area graph of the DataFrame. 	<b>df.plot.hexbin()</b> Plot a hexbin graph of the DataFrame. 
<b>df.plot(subplots=True)</b> Separate into different graphs for each column in the DataFrame.	<b>df.plot(cumulative=True)</b> Creates a cumulative plot	<b>df.plot(stacked=True)</b> Stacks the data for the columns on top of each other. (bar, barh and area only)	
<b>df.plot(title="Graph of A against B")</b> Sets the title of the graph.	<b>df.plot(bins=30)</b> Set the number of bins into which data is grouped (histograms)	<b>df.plot(alpha=0.5)</b> Sets the transparency of the plot to 50%.	
<b>df.plot(subplots=True, title=['col1', 'col2', 'col3'])</b> Arguments can be combined for more flexibility when graphing, this would plot a separate line graph for of column of a 3-columned DataFrame. The first string in the list of titles applies to the graph of the left-most column.			

## Changing Type

<b>pd.to_numeric(data)</b> Convert non-numeric types to numeric.	<b>df.as_type(type)</b> Convert data to (almost) any given type including categorical
<b>pd.to_datetime(data)</b> Convert non-datetime types to datetime type	<b>df.infer_objects()</b> Attempts to infer a better type for object type data.
<b>pd.to_timedelta(data)</b> Convert non- timedelta types to timedelta	<b>df.convert_dtypes()</b> Convert columns to best possible dtypes

## Datetime

With a Series containing data of type datetime, the dt accessor is used to get various components of the datetime values:

<b>s.dt.year</b> Extract the year	<b>s.dt.day</b> Extract the day (int) from the date.
<b>s.dt.month</b> Extract the month as an integer.	<b>s.dt.quarter</b> Find which quarter the date lies in.
	<b>s.dt.hour</b> Extract the hour.
	<b>s.dt.minute</b> Extract the minute.
	<b>s.dt.second</b> Extract the second.

## Mapping

Apply a mapping to every element in a DataFrame or Series, useful for recategorizing or transforming data.

<b>s.map(lambda x: 2*x)</b> Returns a copy of the series where every entry is doubled
<b>df.apply(lambda s: s.max() - s.min(), axis=1)</b> Returns a Series with the difference of the maximum and minimum values of each row of the DataFrame

## Series String Operations

Similar to python string operations, except these are vectorized to apply to the entire Series efficiently.

<b>s.str.count(pattern)</b> Returns a series with the integer counts in each element.	<b>s.str.cat()</b> Concatenate elements into a single string
<b>s.str.get(index)</b> Returns a series with the data at the given index for each element.	<b>s.str.partition(sep)</b> Splits the string on the first instance of the separator
<b>s.str.join(sep)</b> Returns a series where each element has been concatenated.	<b>s.str.slice(start, stop, step)</b> Slices each string
<b>s.str.title()</b> Converts the first character of each word to be a capital.	<b>s.str.replace(pat, rep)</b> Use regex to replace patterns in each string.
<b>s.str.len()</b> Returns a series with the lengths of each element.	<b>s.str.isalnum()</b> Checks whether each element is alpha-numeric

## Input/Output

Common file types for data input include CSV, JSON, HTML which are human-readable, while the common output types are usually more optimized for performance and scalability such as feather, parquet and HDF.

<b>df = pd.read_csv(filepath)</b> Read data from csv file	<b>df.to_parquet(filepath)</b> Write data to parquet file
<b>df = pd.read_html(filepath)</b> Read data from html file	<b>df.to_feather(filepath)</b> Write data to feather file
<b>df = pd.read_excel(filepath)</b> Read data from xls (and related) files	<b>df.to_hdf(filepath)</b> Write data to HDF file
<b>df = pd.read_sql(filepath)</b> Read data from sql file	<b>df.to_clipboard()</b> Copy object to the system clipboard
<b>pd.read_clipboard()</b> Read text from clipboard	

## Frequently Used Options

Pandas offers some 'options' to globally control how Pandas behaves, display etc. Options can be queried and set via: **pd.options.option\_name** (where *option\_name* is the name of an option). For example:  
**pd.options.display.max\_rows = 20**  
Set the **display.max\_rows** option to 20.

### Functions

<b>get_option(option)</b> Fetch the value of the given option.
<b>set_option(option)</b> Set the value of the given option.
<b>reset_option(options)</b> Reset the values of all given options to default settings.
<b>describe_option(options)</b> Print descriptions of given options.
<b>option_context(options)</b> Execute code with temporary option settings that revert to prior settings after execution.

### Display options

<b>display.max_rows</b> The maximum number of rows displayed in pretty-print.
<b>display.max_columns</b> The maximum number of columns displayed in pretty-print.
<b>display.expand_frame_repr</b> Controls whether the DataFrame representation stretches across pages.
<b>display.large_repr</b> Controls whether a DataFrame that exceeds maximum rows/columns is truncated or summarized
<b>display.precision</b> The output display precision in decimal places.
<b>display.max_colwidth</b> The maximum width of columns, longer cells will be truncated.
<b>display.max_info_columns</b> The maximum number of columns displayed after calling <b>info()</b> .
<b>display.chop_threshold</b> Sets the rounding threshold to zero when displaying a Series/DataFrame.
<b>display.colheader_justify</b> Controls how column headers are justified.