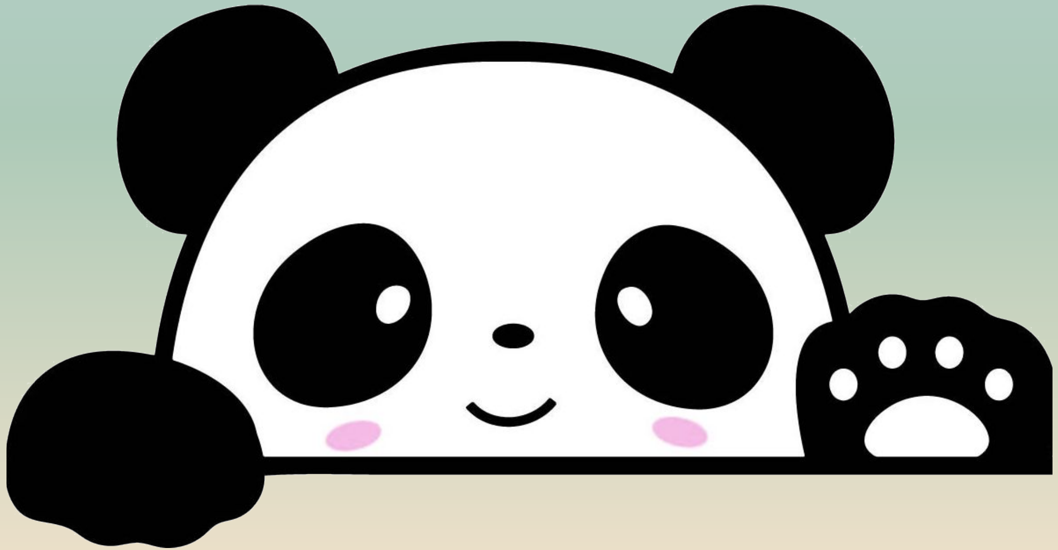
Intro to Pandas

Rebecca Pitts

Lund University

LTH Faculty of Engineering



Course Outline

Pandas

- Object classes: Series & DataFrames
- Basic input/output & inspection
- Selecting and filtering data
- Handling missing values
- Operations
- Merging, joining, & concatenating
- GroupBy & hierarchical DataFrames
- Advanced I/O

Seaborn

- The Pandas-Seaborn interface
- Scatter & density plots
- Paired data grids
- Heatmaps & matrices
- Categorical data plots
- Built-in regression methods
- Style & formatting with Matplotlib

Course Outline

Day 1

- What is Pandas? Why use it?
- Object classes & data types
- Basic input/output
- **Inspection & cleaning**: selection & filtering, handling missing data
- **Basic Operations:** stats, binary, vectorized math & string methods
- Sorting, reindexing, & merging

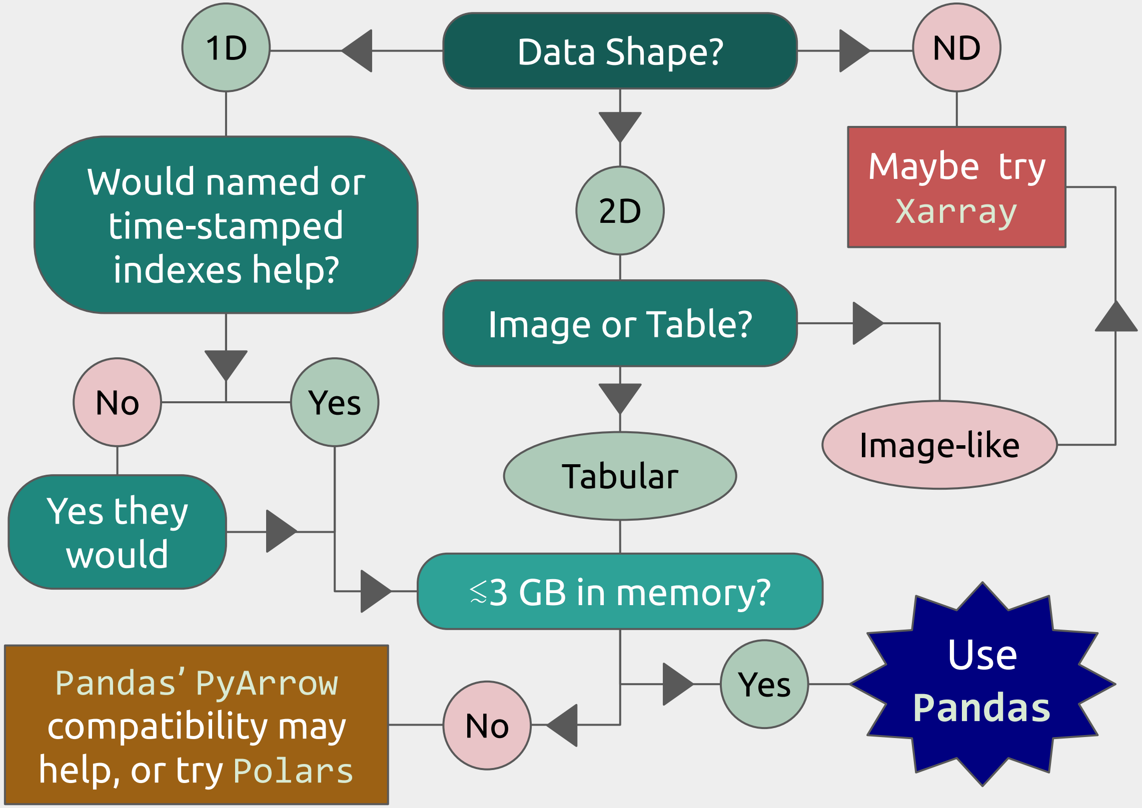
Day 2

- Intro to GroupBy objects
- **More Operations:** comparing data, complex &/or user-defined functions, windowing, & iteration
- Built-in plotting methods
- Time series functionality
- **Advanced topics:** ML prep, memory-saving data types, MultiIndexing & hierarchical DataFrames

What is Pandas? Is it right for my data?

Pandas = PANel Data AnalysiS

- Python data library for cleaning, organizing, & statistically analyzing large data sets
- Originally developed for analyzing & modelling financial records (panel data) over time



Installation

Pandas

Anaconda distribution now comes with pandas and its dependencies by default.

If you use PyPI, make sure `NumPy`, `SciPy`, `python-dateutil`, `pytz`, & `tzdata` are installed, & then run

`pip install pandas`

If you work with HDF files, you will also need `pytables`

Seaborn

Anaconda distribution now comes with seaborn and its dependencies by default.

If you use PyPI, make sure `NumPy`, `pandas`, `Matplotlib`, & `SciPy` are installed, & run

`pip install seaborn` or

`pip install seaborn[stats]`

if you need `statsmodels` for regression

Pros and Cons of Pandas

Pros

- Explicit, automatic data alignment: all entries have corresponding row & column labels/indexes
- Easy to add, remove, transform, compare, broadcast, & aggregate data within & across data structures
- Data structures support any mix of numerical, string, list, Boolean, & `datetime` datatypes
- I/O interface supports wide variety of text, binary, & database formats, including Excel, JSON, HDF5, NetCDF, & SQL
- Hundreds of built-in functions for statistical analysis, time series analysis, grouping & hierarchical (re)organization, missing data handling, & basic plotting, plus support for user-defined functions
- Simple interface with Seaborn & increasingly also Matplotlib

Cons

- Syntax feels inconsistent
- Multi-indexing for >2 dimensions is complicated, limits functionality (try `Xarray` for ND data)
- Parallelization support via `PyArrow` & interoperability with `Polars` is still experimental

Definitions to know before we get started

When you see a Python function described in documentation in the form:

`module.fxn_name(*args,**kwargs)`

You need to know what args & kwargs are:

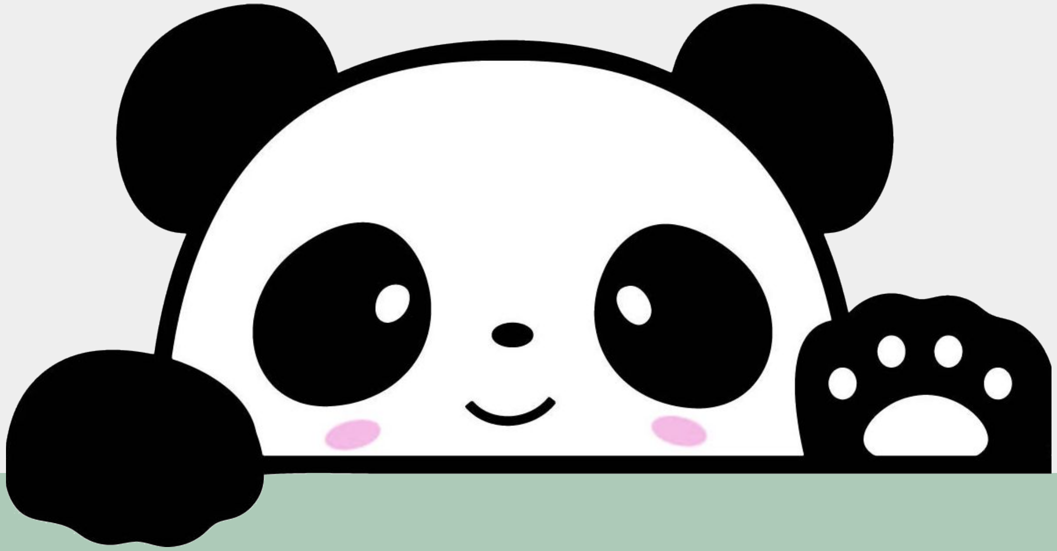
- **args** = positional **arg**uments; usually mandatory
- **kwargs** = **k**ey**w**ord **arg**uments; usually optional

You also need to know what classes, methods, & attributes are →

Classes are templates to make Python objects, with methods & attributes

Methods associate functions with the class & allow quick evaluation for each class instance. **Syntax:** `obj.method()` or `obj.method(*args, **kwargs)`

Attributes let you automatically compute & store values that can be derived for any instance of the class. **Syntax:** `obj.attribute`



Pandas Object Classes & Data Types

Main Pandas object classes: Series & DataFrame

`pandas.Series(data, index=None, name=None, **kwargs)`

- 1D array with customizable indexes (labels) attached to every entry for easy access, & optionally a name for later addition to a DataFrame
    - Indexes can be numbers (integer or float), strings, or `datetime` objects; defaults to 0-based integer indexing

`pandas.DataFrame(data, columns=None, index=None, **kwargs)`

- 2D array where every column is a Series: all entries accessible by column & row labels
- Any function that works with a DataFrame will work with a Series unless the function specifically requires column or index arguments
- Column labels & row indexes/labels can be safely (re)assigned as needed

API reference in documentation shows *hundreds* of methods for each.

Quick aside on row & column labels

Pandas documentation has inconsistent nomenclature for row & column labels:

- “**Indexes**” may refer to just the row labels, or both row & column labels
- “**Columns**” may refer to the labels & contents of columns collectively, or only the labels
- “**Keys**” may be used to refer to column labels, or occasionally both column & row labels, particularly in commands designed to mimic SQL functionality
- A column label may be called a “**name**”, after the optional Series label

I try to stick with “**labels**” to avoid confusion with strictly numerical indexes for array-like slicing, & Index data types, but I may slip into the convention of using “indexes” for row labels since these are numerical & 0-based by default.

Basic Series & DataFrame Attributes (minimal list)

- **`df.index`—type `Index`, list of row labels**
- **`df.columns`—type `Index`, list of column labels**
- `df.axes`—nested list of row & column labels or generators for numerical indexes
- `df.dtypes`—list of datatypes by column
- `df.empty`—bool, True if `df` is empty
- `df.ndim`—number of axes (2 for DataFrame, 1 for Series)
- `df.shape`—tuple, length of `df` along each axis
- `df.size`—int, total number of data entries
- `df.values`—returns `df` converted to a NumPy array (also applicable to `.columns` & `.index`)

```
dummy_df = pd.DataFrame(np.linspace(0.5,10,20).reshape(5,4),
                        columns=['a','b','c','d'])
print(dummy_df,'\n')
print(dummy_df.ndim, dummy_df.shape, dummy_df.size,'\n')
dummy_df.axes
```

```
     a    b    c     d
0  0.5  1.0  1.5   2.0
1  2.5  3.0  3.5   4.0
2  4.5  5.0  5.5   6.0
3  6.5  7.0  7.5   8.0
4  8.5  9.0  9.5  10.0

2 (5, 4) 20


[RangeIndex(start=0, stop=5, step=1),
 Index(['a', 'b', 'c', 'd'], dtype='object')]
```

Here the `RangeIndex()` part stores the row labels, & the `Index()` part stores the column labels

Pandas data types in Series & DataFrames

Basic

- **`float64`**, **`int64`**: standard numeric data types
- **`object`**: str, Bool, list/tuple, & mixed data types (malformed data)

Time Series

- **`datetime64`**`[ns(,tz)]`: timestamps formatted like `datetime` objects
- **`timedelta64`**`[ns]`: time increments (or decrements) relative to a fixed timestamp (anchor point defaults to 0)
- **`period`**`[<unit>]`: time increments defined by specifying a divisible timespan (e.g. a particular month) & the units of subdivision (e.g. days)

Other

- **`Categorical`**: set-like type for non-numeric data with few unique values; drastically reduces memory usage & good for GroupBy ops
- **`Interval`**: tuples of bin edges, all of which must be open/closed on the same side; usually output by `pd.cut()` or `pd.qcut()`

Index-class objects & sub-classes

`Index`-class objects, like those returned by `df.columns` & `df.index`, are *immutable, hashable* sequences used to align data for easy access. Subclasses include:

Basic

`RangeIndex`: for monotonic integer sequences; default Index type if no row Indexes assigned by user

Time Series

`DatetimeIndex`: for timestamps as (row) indexes

`TimedeltaIndex`: for time increments as (row) indexes

`PeriodIndex`: for time intervals as (row) indexes

Other

`CategoricalIndex`: stores full list of Indexes plus set of unique Index values in `.categories` attribute

`MultiIndex`: multi-level indexes for GroupBy objects (later) & hierarchical DataFrames

Indexes have many Series-like attributes & set methods, but Index methods only return copies.

Important Index attributes (minimal list)

Index objects share many attributes & some methods with Series & NumPy arrays, & a few methods with DataFrames

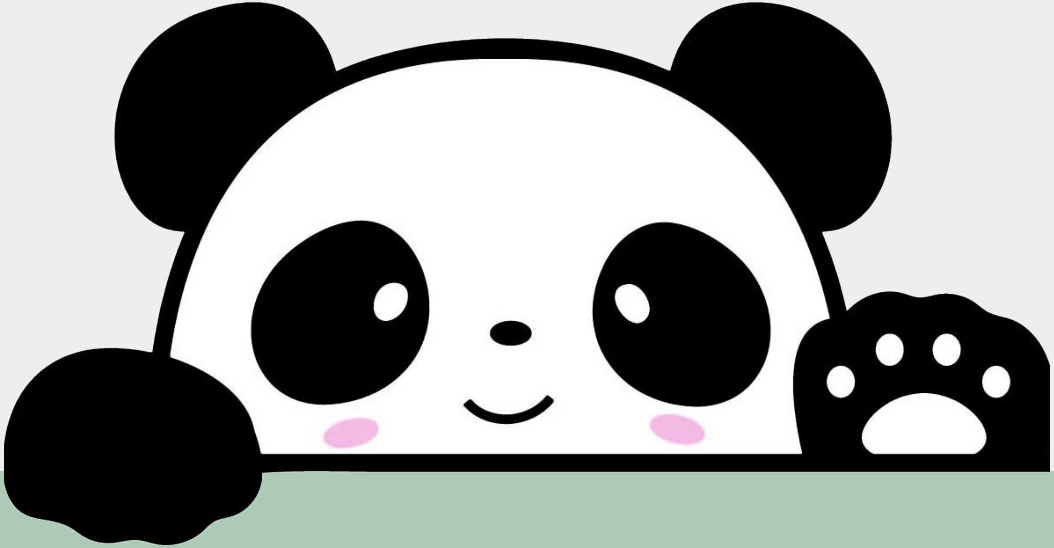
- Index.**hasnans**—True if None/NaN in Index
- Index.**has_duplicates**—True if any duplicates in Index, not what/where they are
  - Inverse: Index.**is_unique**—True only if *all* indexes are unique
  - Index.**duplicated**(keep='first'|'last') returns bool mask where duplicates are True, excluding first or last depending on kwarg keep
- Index.**dtype**—get data type of Index (also available: lots of .is_<type> attributes)
- Index.**is_monotonic_increasing | decreasing**—True if values vary monotonically or repeat consecutively

```
idummy = pd.Index([1,2,3,3,3, 4,5,6,6,6, 7,8,9,9,9])
print(idummy.duplicated(keep='last'),',', idummy.is_monotonic_increasing)
```

```
[False False  True  True False False False  True  True False False False
  True  True False] , True
```

Important Index methods (*very* minimal list)

- `Index.asof(label)`—if `Index` is sorted, returns label itself if it exists, nearest label before it otherwise; NaN if not sorted or if all indexes are later
- `Index.append(other)`—returns copy of `Index` with other (`Index` or list thereof) added
- `Index.insert(loc,label)`—insert new `Index` label at integer position (index) `loc`
- `Index.drop_duplicates(keep='first'|'last')`—returns copy of `Index` with duplicates removed, keeping either first or last (does not change source Series or DataFrame)
  - `Index.drop('label')`—returns copy of `Index` with specific labels deleted
- `Index.argmin | argmax()`—get integer position of [smallest | largest] index label
- `Index.union | intersection | difference | symmetric_difference(other)`— returns set of index labels in [`Index` or `other` | `Index` & `other` | `Index` & not `other` | `Index` or `other` but not both]
- `Index.isin(values)`—check if values (list) are in `Index` (also a Series/DataFrame method)
- `Index.where(cond, other=None)`—replace values where condition is False (also a Series/DataFrame method), with ‘other’ if given



Input/Output

Basic I/O (not a complete list)

CSV and Excel are most-used formats

Type	Data Description	Reader	Writer
text	CSV (or any ASCII text with a standard delimiter)	`read_csv(path_or_url, sep=',', **kwargs)`	`to_csv()`
text	Fixed-Width Text File	`read_fwf()`	N/A
text	JSON	`read_json()`	`to_json()`
text	HTML	`read_html()`	`to_html()`
text	LaTeX	N/A	`Styler.to_latex()`
text	XML	`read_xml()`	`to_xml()`
text	Local clipboard	`read_clipboard()`	`to_clipboard()`
SQL	SQL	`read_sql()`	`to_sql()`
SQL	Google BigQuery	`read_gbq()`	`to_gbq()`
binary	Python Pickle Format	`read_pickle()`	`to_pickle()`
binary	MS Excel	`read_excel(path_or_url, sheet_name=0, **kwargs)`	`to_excel(path, sheet_name=...)`
binary	OpenDocument	`read_excel(path_or_url, sheet_name=0, **kwargs)`	`to_excel(path, engine="odf")`
binary	HDF5 Format	`read_hdf()`	`to_hdf()`
binary	Apache Parquet	`read_parquet()`	`to_parquet()`

Internal data structure conversion

To Pandas DataFrame from…

- **List or array:** accepted directly by `pd.Series()` or `pd.DataFrame()`
- **Dictionary or dict of dicts:** use `pd.DataFrame.from_dict()`
  - Keys will be column labels unless kwarg `orient='index'`
- **Structured/record array**, or **sequence of tuples or dicts:** use `pd.DataFrame.from_records()`

From Pandas DataFrame/Series to…

- **NumPy array:** use `df.to_numpy()` (will lose row & column labels)
- **Dictionary:** use `df.to_dict()`
  - Use `orient='list'` to get dict with only columns as keys
- **NumPy record array:** use `df.to_records()` method
- **Strings:** use `df.to_str()` method (mostly same kwargs as `.to_html()`)

Basic I/O - some common kwargs

Text data (txt, csv, xlsx, html, clipboard, etc) often contain a row of column names. Many have a column of data that would make better indexes than arbitrarily assigned numbers.

- Text reader functions assume top row is for column names (`header=0`); set **`header`** to another index if column labels are lower, or override column labels with kwarg **`columns`**
- Save memory: **`usecols`** lets you load only essential columns (accepts indexes or labels)
- Text reader functions assign 0-based row indexes from top to bottom; can instead set **`index_col`** to the index of a data column (0-based, left-to-right), or override the defaults by setting `index` to a list or array of choice
    - Can use `df.set_index('column')` to set existing column as index later
- Can use **`converters`** to fix problematic input text patterns, if you know about them
- For users of commas as decimal markers: set **`decimal=','`** (default is `decimal='.'`)

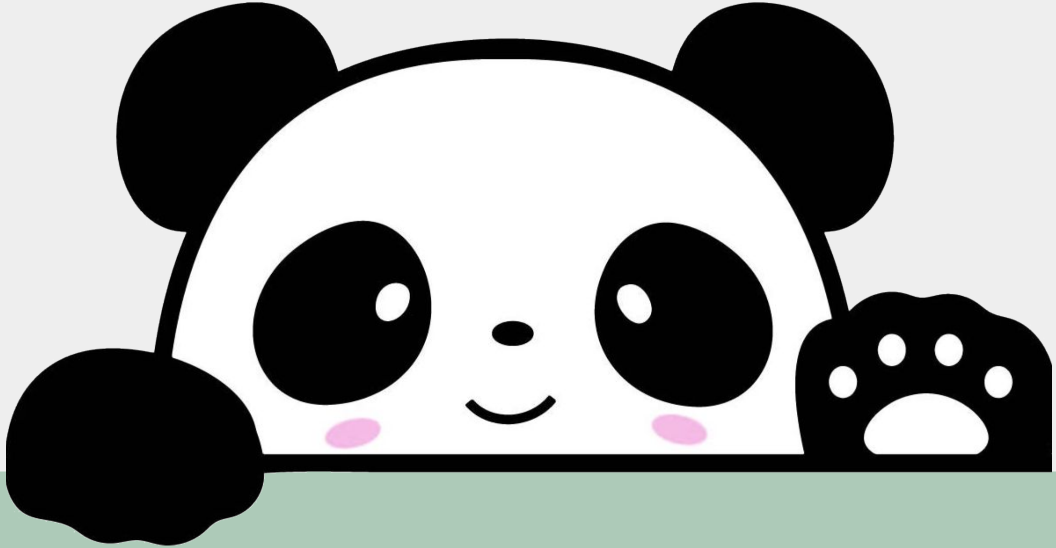
Basic I/O Example

```
df = pd.read_csv('exoplanets_5250_EarthUnits.csv',index_col=0)
df.head()
```

#name	distance	star_mag	planet_type	discovery_yr	mass_ME	radius_RE	orbital_radius_AU	orbital_period_yr
11 Comae Berenices b	304.0	4.72307	Gas Giant	2007	6169.20	12.096	1.290000	0.892539
11 Ursae Minoris b	409.0	5.01300	Gas Giant	2009	4687.32	12.208	1.530000	1.400000
14 Andromedae b	246.0	5.23133	Gas Giant	2008	1526.40	12.88	0.830000	0.508693
14 Herculis b	58.0	6.61935	Gas Giant	2002	2588.14	12.544	2.773069	4.800000
16 Cygni B b	69.0	6.21500	Gas Giant	1996	566.04	13.44	1.660000	2.200000

Write-out example:

```
df.to_csv('exoplanets_5250_EarthUnits.txt', sep='|',
          decimal=',', index=True)
```



Inspection & Cleaning

Data inspection convenience functions

- `df.head()` prints first 5 rows of data with row & column labels; `df.tail()` does same for last 5 rows. Both accept integer arg to print more or fewer rows
- `df.info()` prints # of rows & first & last index values; titles, index #s, valid data counts, & datatypes of columns; & size of `df` in memory.
- `df.describe()` prints summary statistics for all numerical columns (not always useful)
- `df.nunique()` prints count of unique values in each column

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 5250 entries, 11 Comae Berenices b to YZ Ceti d
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   distance            5233 non-null   float64
 1   star_mag            5089 non-null   float64
 2   planet_type         5250 non-null   object
 3   discovery_yr        5250 non-null   int64
 4   mass_ME             5227 non-null   float64
 5   radius_RE           5233 non-null   float64
 6   orbital_radius_AU   4961 non-null   float64
 7   orbital_period_yr   5250 non-null   float64
 8   eccentricity        5250 non-null   float64
 9   detection_method    5250 non-null   object
dtypes: float64(7), int64(1), object(2)
memory usage: 580.2+ KB
```

Type “object” is used for strings, boolean values, & mixed data types

Data Inspection Cont.

```
df.describe()
```

	distance	star_mag	discovery_yr	mass_ME	radius_RE	orbital_radius_AU	orbital_period_yr	eccentricity
count	5233.000000	5089.000000	5250.000000	5227.000000	5233.000000	4961.000000	5.250000e+03	5250.000000
mean	2167.168737	12.683738	2015.732190	460.035267	5.627083	6.962942	4.791509e+02	0.063924
std	3245.522087	3.107571	4.307336	3761.458727	5.315522	138.673600	1.680445e+04	0.141402
min	4.000000	0.872000	1992.000000	0.020000	0.296000	0.004400	2.737850e-04	0.000000
25%	389.000000	10.939000	2014.000000	3.970000	1.760000	0.053000	1.259411e-02	0.000000
50%	1371.000000	13.543000	2016.000000	8.470000	2.732800	0.102800	3.449692e-02	0.000000
75%	2779.000000	15.021000	2018.000000	159.000000	11.715200	0.286000	1.442163e-01	0.060000
max	27727.000000	44.610000	2023.000000	239136.000000	77.280000	7506.000000	1.101370e+06	0.950000

Note: integer columns are treated as floats, & Boolean values are treated as object-type, so these results are not always useful

The memory_usage() function

`df.memory_usage()` prints table of sizes of each column of `df` in memory, in bytes, with 1 BIG CAVEAT:

- Numerical & boolean data are fixed size in bytes—stored within `df` in memory
- Object-type data (strings) are NOT fixed in size—memory stores only *pointers* at location of `df`; string *values* are elsewhere & are usually much larger in memory
- Must use `memory_usage(deep=True)` to estimate memory used by string values instead of just pointers (reported values will be upper bounds, but more realistic)

Size of `df` in memory reported by `df.info()` is the sum of values returned by `memory_usage(deep=False)`: **don't rely on `df.info()` to monitor memory use!**

compare:

```
df.memory_usage()#deep=True)

Index                174136
distance              42000
star_mag              42000
planet_type           42000
discovery_yr          42000
mass_ME               42000
radius_RE             42000
orbital_radius_AU     42000
orbital_period_yr     42000
eccentricity          42000
detection_method      42000
dtype: int64
```

```
df.memory_usage(deep=True)

Index                491638
distance              42000
star_mag              42000
planet_type          355545
discovery_yr          42000
mass_ME               42000
radius_RE             42000
orbital_radius_AU     42000
orbital_period_yr     42000
eccentricity          42000
detection_method     348608
dtype: int64
```

Data Selection (& Assignment) Syntax

To Access…	Syntax
1 column	`df['col_name']`
1 named row	`df.loc['row_name']`
1 row by index	`df.iloc[index]`
1 column by index (rarely used)	`df.iloc[:,index]`
1 cell by row & column labels	`df.at['row_name','col_name']` or `df.at[index,'col_name']`
1 cell by row & column indexes	`df.iat[row_index, col_index]`
subset of columns	`df[['col0', 'col1', 'col2']]`
subset of named rows	`df.loc[['rowA','rowB','rowC']]`
subset of rows by index	`df.iloc[i_m:i_n]` or `df.take([i_m, …, i_n])` where *i_m* & *i_n* are the m^{th} & n^{th} integer indexes
1+ rows & columns by name	**`df.loc['row','col']` or `df.loc[['rowA','rowB', ...],['col0', 'col1', ...]]`**
1+ rows & columns by index	**`df.iloc[i_m:i_n, j_p:j_q]` where *i* & *j* are row & column indexes, respectively**
columns by name & rows by index	**`df[['col0', 'col1', 'col2']].iloc[i_m:i_n]`**

Selection Syntax Example

```
print(df[['planet_type','mass_ME']].iloc[25:35])
```

```
                planet_type  mass_ME
#name
51 Eridani b      Gas Giant   636.00
51 Pegasi b       Gas Giant   146.28
55 Cancri b       Gas Giant   264.13
55 Cancri c       Gas Giant    54.51
55 Cancri d       Gas Giant  1233.20
55 Cancri e     Super Earth     7.99
55 Cancri f       Gas Giant    44.84
61 Virginis b  Neptune-like     5.10
61 Virginis c  Neptune-like    18.20
61 Virginis d  Neptune-like    22.90
```

Beware chain indexing—use .loc[...] instead!

Printing the result of chain indexing, i.e. `df[row_selector][col_selector]` will often return what you'd expect. BUT BE WARNED: whether the return value is a copy or a view of the original data is hard to predict or determine.

Pandas will usually raise `SettingWithCopy` if you use chain indexing in assignment, but it doesn't catch everything.

`df.loc[row_selector, col_selector]` is overwhelmingly preferred.

```
print(df.loc[(df.index.str.contains('PSR')) &
             (df['discovery_yr'] < 2000), 'planet_type'])
print('\n...looks the same as...\n')
print(df[(df.index.str.contains('PSR')) &
         (df['discovery_yr'] < 2000)]['planet_type'])
print("\n...but only use the first version!")
```

```
#name
PSR B1257+12 b    Terrestrial
PSR B1257+12 c    Super Earth
PSR B1257+12 d    Super Earth
Name: planet_type, dtype: object

...looks the same as...

#name
PSR B1257+12 b    Terrestrial
PSR B1257+12 c    Super Earth
PSR B1257+12 d    Super Earth
Name: planet_type, dtype: object

...but only use the first version!
```

Conditional Selection

Any binary comparison operator (>, <, ==, =>, =<, !=) and most logical operators can be used inside `[]` of `df[...]`, `df.loc[...]`, and `df.iloc[...]` with some conditions:

- Bitwise logical operators (`&`, `|`, `^`, `~`) must be used in lieu of plain-English counterparts (`and`, `or`, `xor`, `not`)
- When 2+ conditions are specified, each condition must be bracketed by `()` or code will raise `TypeError`
- The "`is`" operator does not work within `.loc[]`. Use **`.isna()`** or **`.notna()`** to check for invalid data, and **`.isin()`**, **`.notin()`**, or **`.str.contains()`** to check for the presence of substrings

```
print(df.loc[ (df['discovery_yr'] < 2007) &
              (df['planet_type'] != 'Gas Giant'),
      'planet_type'])
```

```
#name
55 Cancri e               Super Earth
GJ 436 b                 Neptune-like
GJ 581 b                 Neptune-like
GJ 876 d                 Neptune-like
HD 160691 d              Neptune-like
HD 190360 c              Neptune-like
HD 4308 b                Neptune-like
HD 49674 b               Neptune-like
HD 69830 b               Neptune-like
HD 69830 c               Neptune-like
HD 69830 d               Neptune-like
HD 99492 b               Neptune-like
OGLE-2005-BLG-169L b     Neptune-like
OGLE-2005-BLG-390L b     Neptune-like
PSR B1257+12 b            Terrestrial
PSR B1257+12 c            Super Earth
PSR B1257+12 d            Super Earth
Name: planet_type, dtype: object
```

The .query() method

`df.query()` lets you select* & filter rows via a string passed to implicit `exec()`

- Allows plain-English logical operators**
- Reduces number of `[]` and `()` needed
- Built-in variable `index` if you can't or don't want to name the index
- Insert variables by prefixing them with `@`
- **Important:** row & column names must be set between grave accents (), not single or double quotes

```
y = 2009
df.query("index.str.contains('PSR') and \
        `discovery_yr` < @y")['planet_type']
```

```
#name
PSR B1257+12 b     Terrestrial
PSR B1257+12 c     Super Earth
PSR B1257+12 d     Super Earth
PSR B1620-26 b       Gas Giant
Name: planet_type, dtype: object
```

*`df.query()` allows column-wise *filtering* but not column *selection*—must use chain indexing, ergo cannot use `df.query()` to set values

**binary operators are not implemented for all data types—use == and != instead of `is` or `is not`

Finding & handling missing/invalid data

- Check for missing data with `.isna()` & `.notna()` DataFrame methods
  - In `datetime64`-, `timedelta64`-, & `period`-type data, `pd.NaT` takes place of NaN.
- No Pandas equivalent of `np.isinf()`: if you need locations, use `np.isinf(copy.to_numpy())` where `copy` is a copy of column/row to search
- Use `.dropna(axis=axis)` to remove whole rows or columns containing invalid entries (recommend keeping `inplace=False`).
- Use `.fillna()` to replace NaNs with a fixed value, or `.interpolate()` to fill gaps based on surrounding data. Any interpolation algorithm allowed as the method kwarg of `scipy.interpolate()` is accepted.
  - Most math & stats functions exclude NaNs by default, so they can usually be left alone; can include by setting `skipna=False` (warning: this will propagate NaNs).

Finding invalid data example

```
df[df['orbital_radius_AU'].isna()].iloc[:5,2:7]
```

	planet_type	discovery_yr	mass_ME	radius_RE	orbital_radius_AU
#name					
CI Tauri b	Gas Giant	2019	3688.80	12.432	NaN
CoRoT-7 d	Neptune-like	2022	17.14	4.3008	NaN
DS Tucanae A b	Neptune-like	2019	413.40	5.7008	NaN
EPIC 201238110 b	Super Earth	2019	4.16	1.87	NaN
EPIC 201427007 b	Super Earth	2021	2.86	1.5	NaN

Cleaning bad (but not NaN) or duplicate data

- If 2+ rows are identical, use `df.drop_duplicates()` to return duplicate-free copy of the DataFrame (default) or remove duplicates in-place (`inplace=True`).
    - Use `subset` kwarg to remove duplicates by specific columns (more aggressive).
- Use `df.drop(data, axis=axis)` to get rid of unneeded columns (`axis=1`) or rows (`axis=0`) by name or index; set `inplace=True` to modify `df` in-place.
- To mask bad *numeric* data (or infinities), use `df.mask(condition, other=None)`, where `other` (default NaN) passes scalars or Series/DataFrame to replace values with
- If data are predictably malformed, use `df.replace(to_replace=old, value=new)` where `old` &/or `new` can be `str`, `regex`, `list`, `dict`, `Series`, `int`, `float`, or `None`.
    - If `to_replace` is a `dict`, key-value pairs are interpreted as `old: new` unless `value` kwarg is supplied; then key-value pairs are interpreted as `in_column: old`

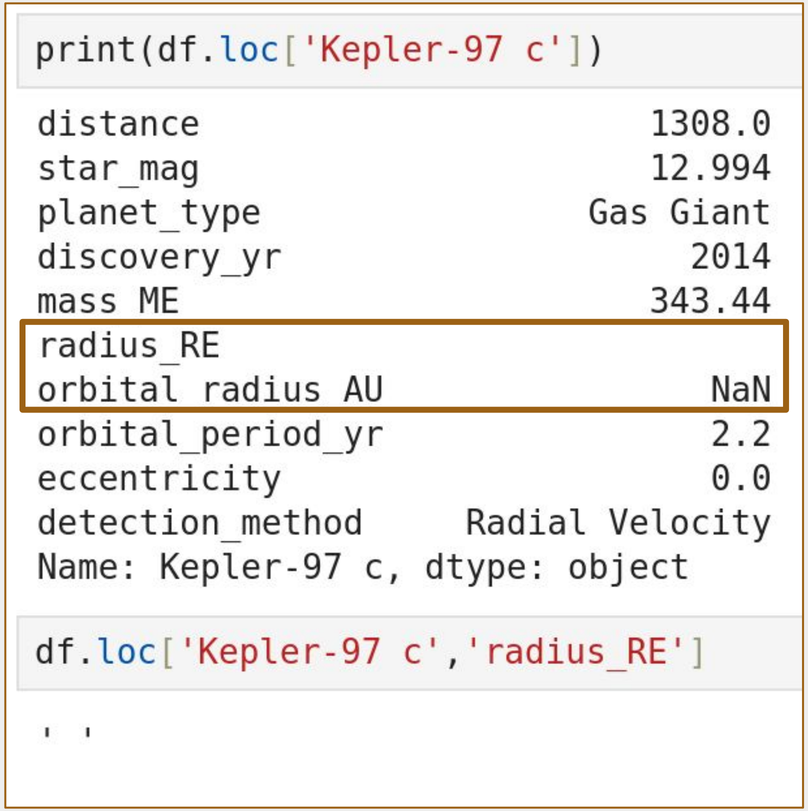
NaNs vs Whitespace

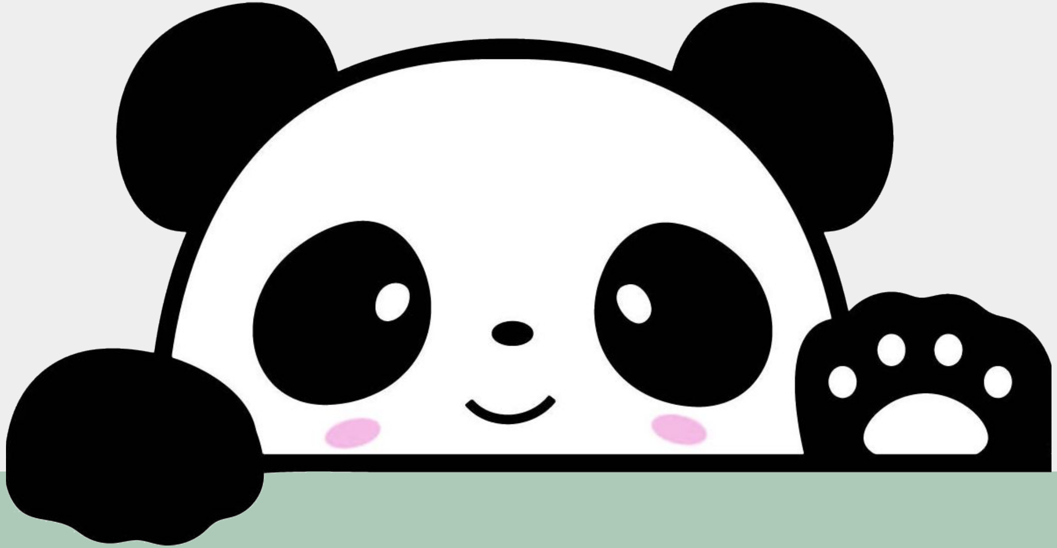
Pandas assumes all whitespace is intentional (in case numbers are zip codes, currencies, etc.); therefore:

- `.isna()` ignores spaces
- Numeric columns containing spaces are cast as `object` type (like strings)

Fixing a numeric column containing spaces might look like this:

```
df['col'] = df['col'].replace(' ',
np.nan).astype('float64')
```





Basic Operations

Vectorized String Methods

Most built-in string methods can be applied column-wise to Pandas data structures using `.str.<method>()`

- `.str.replace()`—this version does not accept `dict` input where keys are existing substrings & values are replacements; try without `.str`
- `.str.upper()/.lower()`
- `.str.strip()/.rstrip()`
- `.str.split()/.rsplit()`

Kwargs of `.str.split(" ", n=None, expand=False)` (& rstrip counterpart) change output data structure:

- Default: Series of lists of substrings
- With `expand=True`: DataFrame with as many columns as substrings in the longest string
- With `expand=True` & `n>1`: DataFrame with *n* columns where longer substring sequences are truncated & `None`s fill out rows for shorter sequences

```python
dummy1 = pd.Series(['Pandas are cute!',
                    'I like trains.',
                    'Hello, world?',
                    'Hams.'])
print('Original:\n', dummy1)
```

```
Original:
 0    Pandas are cute!
1      I like trains.
2       Hello, world?
3               Hams.
dtype: object
```

```python
dummy2 = dummy1.str.split()
print('Split and get:\n',dummy2.str[:2])
```

```
Split and get:
 0       [Pandas, are]
1          [I, like]
2     [Hello,, world?]
3            [Hams.]
dtype: object
```

```python
dummy3 = dummy1.str.split(expand=True)
print('Expand=True:\n', dummy3)
```

```
Expand=True:
         0       1        2
0  Pandas     are    cute!
1       I    like  trains.
2  Hello,  world?     None
3   Hams.    None     None
```

Statistics & related math methods

Series & DataFrame objects have most NumPy stats methods & few SciPy ones:

- NumPy-like methods: `.abs()`, `.count()`, `.max()`, `.min()`, `.mean()`, `.median()`, `.mode()`, `.prod()`, `.quantile()`, `.sum()`, `.std()`, `.var()`, `.cumsum()`, `.cumprod()`
    - Pandas adds `.cummax()` & `.cummin()`
- SciPy (m)stats-like methods: `.sem()`, `.skew()`, `.kurt()`, `.corr()`
- Can randomly sample data with `.sample(n=n, replace=True, **kwargs)`

Common behaviors:

- NaNs excluded by default, but can include with kwarg `skipna=True` (not recommended)
- Methods require no args/kwargs for Series, but for DataFrame, you must specify **`numeric_only=True`** & `axis`
    - `axis=0` or "`index`" (default): columns preserved, indexes collapse
    - `axis=1` or "`columns`": indexes preserved, columns collapse

Broadcasting basic arithmetic

Vectorized arithmetic with normal operators (`+`, `-`, `*`, `/`, `**`, and `%`) is possible between a Series/DataFrame & a scalar or 2 Series/DataFrames of the same shape.

- E.g.: `df/100.`, `df**-1.5`, `dfA+dfB`, …

Otherwise, use `.add()`, `.sub()`, `.mul()`, `.div()`, `.pow()`, & `.mod()` methods to broadcast `+`, `-`, `*`, `/`, `**`, and `%`, respectively

- Broadcast according to `axis` kwarg (same syntax as stats methods)
- Reverse-order counterparts are prefixed with `r` (`.radd()`, `.rsub()`, …)

```
dfA = pd.DataFrame(np.arange(12).reshape([4,3]),
                   columns = ['a','b','c'])
print(dfA,'\n\n',dfA.div([4.,3.,2., 1.], axis='index'),
      '\n\n', dfA.rdiv([4.,3.,2., 1.], axis='index'))
```

```
   a   b   c           4.0
0  0   1   2      /    3.0
1  3   4   5           2.0
2  6   7   8           1.0
3  9  10  11

     a          b          c
0  0.0   0.250000   0.500000
1  1.0   1.333333   1.666667
2  3.0   3.500000   4.000000
3  9.0  10.000000  11.000000

          a         b         c
0       inf  4.000000  2.000000
1  1.000000  0.750000  0.600000
2  0.333333  0.285714  0.250000
3  0.111111  0.100000  0.090909
```

div()

rdiv()

Broadcasting arithmetic continued

- Series & Index objects have **`divmod()`** method to return DataFrame of integer quotients & remainders (see right)

Performance tip: for scalar or element-wise arithmetic, install & import `numexpr`, & wrap expression `expr` with `pd.eval(expr, engine='numexpr')`

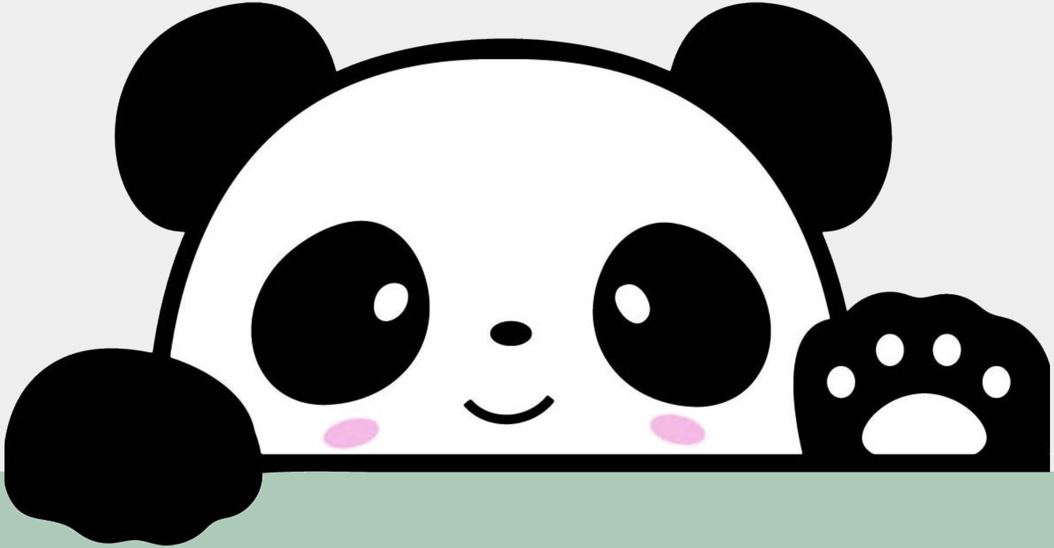
- `numexpr` accelerates computation with multi-threading & smart chunking
- **Note**: only pure int64 or float64 Series/DataFrames benefit

```
ser3 = pd.Series(np.linspace(0,9,11))
rems,mods=ser3.divmod(3)
print(pd.concat((ser3.rename('nums'),
                 rems.rename('rems'),
                 mods.rename('mods')), axis=1))
```

```
    nums  rems  mods
0    0.0   0.0   0.0
1    0.9   0.0   0.9
2    1.8   0.0   1.8
3    2.7   0.0   2.7
4    3.6   1.0   0.6
5    4.5   1.0   1.5
6    5.4   1.0   2.4
7    6.3   2.0   0.3
8    7.2   2.0   1.2
9    8.1   2.0   2.1
10   9.0   3.0   0.0
```

Comparative methods

- Compare Series or DataFrame to single scalar values with normal operators (>=, !=, etc)
- Compare 1 Series or DataFrame element-wise to another of the same shape (as the arg) using `.gt()`, `.lt()`, `.ge()`, `.le()`, `.eq()`, or `.ne()` (>, <, >=, <=, =, and !=, respectively).
- **Boolean reduction:** for any of the above comparisons (to scalars or other pandas data structures), add `.any()` or `.all()` once to collapse the column axis, twice to get 1 value.
- Use `df1.compare(df2)` to find & print differences between 2 **identically indexed** Series or DataFrames (both objects must have the same row & column labels <u>in the same order</u>)
  - `.compare()` will not show data type differences if the values are equal; for that, use `pd.testing.assert_frame_equal(df1, df2)` or `pd.testing.assert_series_equal(df1, df2)` to see if it raises `AssertionError`



(Re)organizing & Merging Data

Sorting

Two methods to sort both Series & DataFrames: `.sort_values(by=row_or_col, axis=0, key=None, kind='quicksort')` & `.sort_index(axis=0, key=None)`

- Both sorting functions return copies unless `inplace=True`
- `axis` kwarg refers to direction along which values will be shifted, not the fixed axis
- `key` kwarg lets you apply a vectorized function (more on this soon) to the index before sorting. ***Note:*** *this alters what the sorting algorithm sees, not the indexes as they will be printed*
- `.sort_values(by=row_or_col, axis=0, kind='quicksort')` sorts Series or DataFrame by value of column(s)/row(s) passed to `by` kwarg (optional for Series)
  - If `by` is type list, order of sort may depend on algorithm given for `kind`.
  - If `by` is a row label, `axis=1` is mandatory

```python
dummy = pd.DataFrame(np.random.randint(0,high=9,size=(4,3)),
                     columns = ['B','a','C'],
                     index = ['h','i','j','k'])
print("Sorted by column C\n",dummy.sort_values('C',axis=0))
print("Sorted by row j\n",dummy.sort_values('j',axis=1))
```

```
Sorted by column C
    B  a  C
i   2  6  1
j   2  7  1
k   8  1  3
h   3  5  8
Sorted by row j
    C  B  a
h   8  3  5
i   1  2  6
j   1  2  7
k   3  8  1
```

Upper & lower-case letters are normally treated separately, with **a** coming after **Z**.

```python
print("Columns sorted alphabetically\n",
      dummy.sort_index(axis=1))
```

```
Columns sorted alphabetically
    B  C  a
h   3  8  5
i   2  1  6
j   2  1  7
k   8  3  1
```

```python
print("Columns sorted alphabetically with key\n",
      dummy.sort_index(axis=1,key=lambda c: c.str.lower()))
```

```
Columns sorted alphabetically with key
    a  B  C
h   5  3  8
i   6  2  1
j   7  2  1
k   1  8  3
```

Need to use str.lower() to internally treat upper- & lower-case letters equivalently

Reindexing

If indexes or columns are missing, `.reindex(labels, index=rows, columns=cols)` can add & sort them in the order of `labels` simultaneously

- Can also change indexes, but only by reassignment; even with `copy=False`, in-place modification is not possible
- Can use `.reindex()` to select data when you aren't sure all the given labels exist, without raising exceptions

`df1.reindex_like(df2)` makes empty DataFrame with same row & column labels as `df2` & inserts values from `df1` at row & column indexes shared with `df2`

Series method `.searchsorted(values)` returns indexes at which to insert values to maintain order

```
dummy = pd.DataFrame(np.random.randint(0,high=9,
                                        size=(4,3)),
                     columns = ['a','b','c'],
                     index = [38,42,36,48])
print(dummy)
print(dummy.reindex(np.arange(36,50,2),
                    axis='index'))
```

```
    a  b  c
38  0  3  0
42  1  1  2
36  6  3  4
48  2  7  1
      a    b    c
36  6.0  3.0  4.0
38  0.0  3.0  0.0
40  NaN  NaN  NaN
42  1.0  1.0  2.0
44  NaN  NaN  NaN
46  NaN  NaN  NaN
48  2.0  7.0  1.0
```

```
dummy2 = pd.DataFrame(
    np.arange(0,9).reshape(3,3),
    columns = ['b','c','d'],
    index = [38,43,48])
print(dummy2.reindex_like(dummy))
```

```
     a    b    c
38 NaN  0.0  1.0
42 NaN  NaN  NaN
36 NaN  NaN  NaN
48 NaN  6.0  7.0
```

Combining data structures

Pandas functions:

1. `.concat()`: combine ≥2 DataFrames or Series along a shared column or index
2. `.merge(left_df, right_df, how='inner')`: combine 2 DataFrames/Series on columns with SQL-style logic
3. `.merge_ordered(fill_method=None)`: combine 2 sorted DataFrames/Series with optional interpolation over gaps
4. `.merge_asof()`: left-join 2 sorted DataFrames/Series by nearest value of index instead of matching values

DataFrame (& Series) methods:

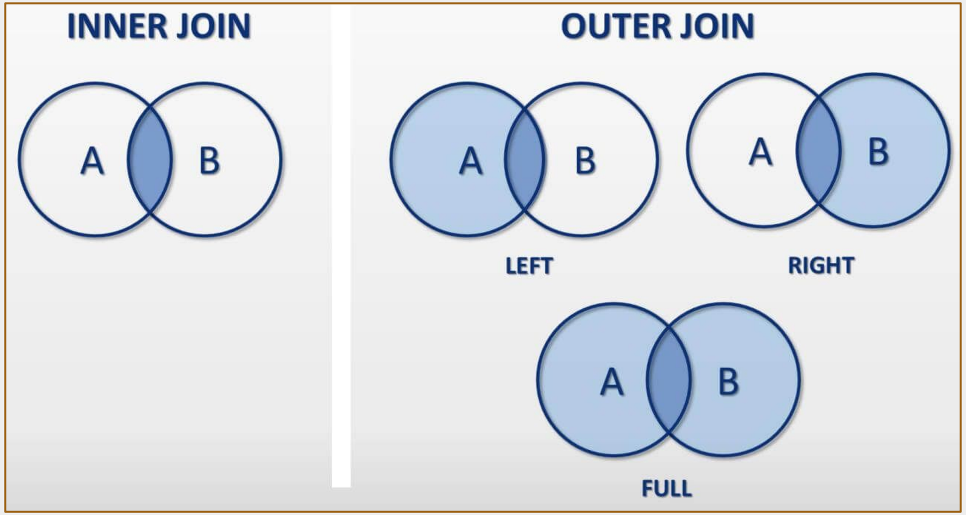
1. `df1.combine_first(df2)`: update missing values of DataFrame `df1` with fill values from DataFrame `df2` at shared index locations
2. `df1.combine(df2, func)`: merge 2 DataFrames column-wise based on given function `func` that takes 2 Series & returns either Series or scalars
3. `df1.join(df2)` (uses `.merge()` internally): join 2 DataFrames/Series on given index(es) or column(s) (also an `Index` method)

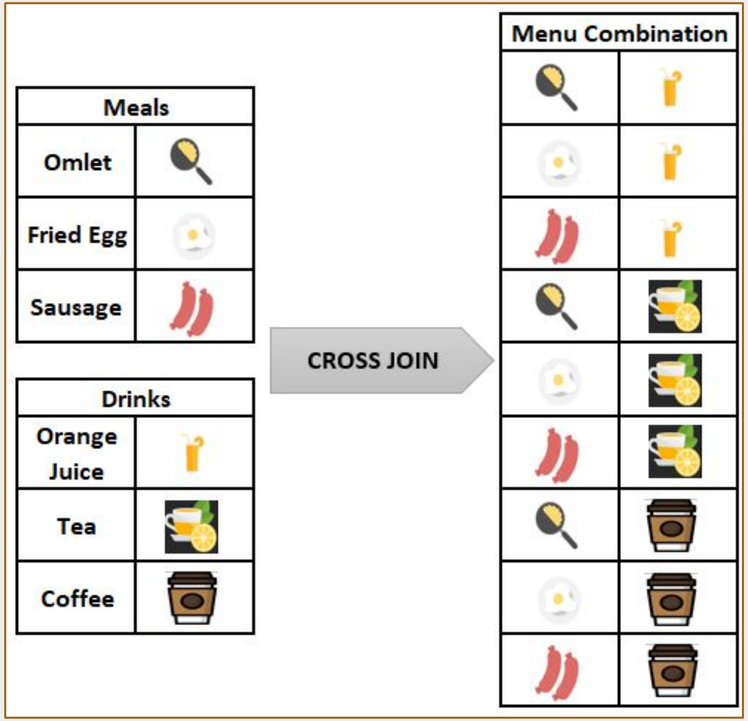
SQL joining styles used by .merge() & .join()

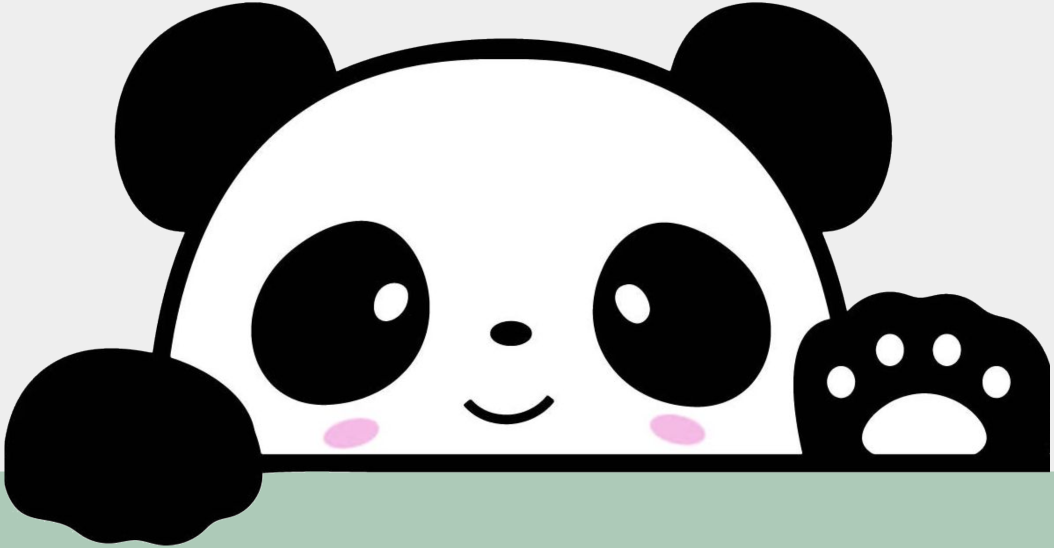
The `.merge()`, `.join()`, & derivatives of merge have a `how` kwarg to toggle different SQL-like set logics, & an `on` kwarg to select subsets of columns to preserve.

1. `'inner'` (default): take intersection of 2 DataFrames in terms of row *positions*, like SQL `inner join`, while preserving all columns unless otherwise specified with `left_on`, `right_on`, `left_index=True`, or `right_index=True`
2. `'outer'`: align on shared data but keep all rows & columns, like SQL `full outer join`, with NaNs where row-column-pairs are not associated with any existing data
3. `'left'`: keep all contents of left (1st) DataFrame, plus any data from right (2nd) that share row & column indexes with left DataFrame, like SQL `left outer join`.
4. `'right'`: keep all contents of right (2nd) DataFrame, plus any data from left (1st) that share row & column indexes with right DataFrame, like SQL `right outer join`.
5. `'cross'`: take Cartesian product of 2 DataFrames (take every non-redundant pairing of every cell in one with every cell in the other), like SQL `cross join`.

SQL join logic in pictures







GroupBy objects

Intro to GroupBy Objects

One of the most powerful Pandas tools, it lets you organize (& sort) data hierarchically & run variable statistical analyses on different subsets of data simultaneously.

- **Syntax**: `grouped = df.groupby(['col1', 'col2', …])` or `grouped = df.groupby(by='col')`
    - To group by rows, take transpose of DataFrame first with `df.T`
- **Most DataFrame methods & attributes can be called on GroupBy objects, but aggregate methods will be evaluated for every group separately**
- GroupBy objects have `.nth()` method to retrieve n^{th} row of every group; *n* can be <0.
- Groups in GroupBy objects can be selected by category name with `.get_group('cat')` or `.get_group(('cat1', 'cat2', …))`, & accessed as iterable with `.groups`
- Separate functions can be broadcast to each group in 1 command

GroupBy example

```
grouped1=df.groupby(['planet_type'])
grouped1.nth(-1)
```

	distance	star_mag	planet_type	discovery_yr	mass_ME	radius_RE	orbital_radius_AU	orbital_period_yr	eccentricity
#name									
LkCa 15 c	516.0	12.025	Unknown	2015	NaN	NaN	18.60000	0.999316	0.00
Wolf 503 b	145.0	10.270	Neptune-like	2018	6.26	2.043	0.05706	0.016427	0.41
YSES 2 b	357.0	10.885	Gas Giant	2021	2003.40	12.768	115.00000	1176.500000	0.00
YZ Ceti b	12.0	12.074	Terrestrial	2017	0.70	0.913	0.01634	0.005476	0.06
YZ Ceti d	12.0	12.074	Super Earth	2017	1.09	1.030	0.02851	0.012868	0.07

GroupBy example continued

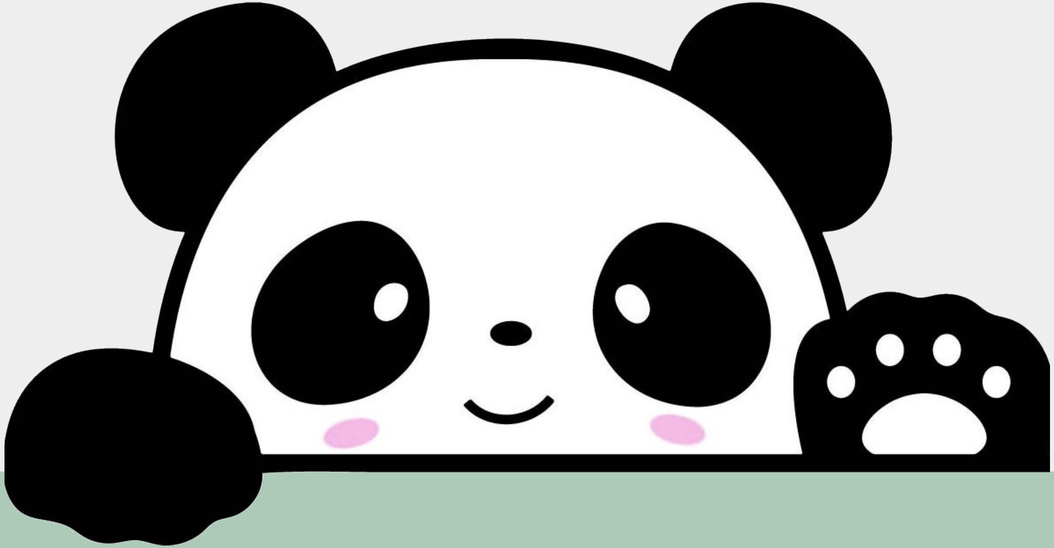
```
grouped1['orbital_radius_AU'].mean()
```

```
planet_type
Gas Giant        21.515449
Neptune-like      0.224902
Super Earth       0.109952
Terrestrial       0.062381
Unknown          16.650000
Name: orbital_radius_AU, dtype: float64
```

```
grouped1.get_group('Unknown').iloc[:,6:8]
```

#name	orbital_radius_AU	orbital_period_yr
KIC 10001893 b	NaN	0.000548
KIC 10001893 c	NaN	0.000821
KIC 10001893 d	NaN	0.002190
LkCa 15 b	14.7	0.999316
LkCa 15 c	18.6	0.999316

Hmm… 2 planets at different orbital radii but with the same period. The same period as Earth, no less. Take with a solar mass of salt. These 2 were refuted in 2019.



Advanced Operations

Applying complex &/or user-defined functions

`.map()`

Apply 1 function that accepts & returns a scalar element-wise

`.agg()`

Apply ≥1 reducing (aggregating) functions

`.apply()`

Apply ≥1 functions column- or row-wise to ≥1 Series

`.transform()`

Apply ≥1 broadcasting functions

Most User-defined functions incorporate 1 or more of these

Yes, there is substantial overlap between these methods for regular Series & DataFrames, but `.agg()` & `.transform()` can also be applied to GroupBy objects

Element-wise functions with .map()

Series/DataFrame method `.map(func)` takes a scalar function & broadcasts it to every element of the data structure

- Function `func` may be passed by name or lambda function, but both input & output must be scalars (no arrays)
- **Note**: it's usually faster to apply vectorized functions if possible (e.g. `df**0.5` is faster than `df.map(np.sqrt)`)
- Not for GroupBy objects

```
def my_func(T):
    if T<=0 or np.isnan(T) is True:
        pass
    elif T<300:
        return 0.2*(T**0.5)*np.exp(-616/T)
    elif T>=300:
        return 0.9*np.exp(-616/T)

junk = pd.DataFrame(np.random.randint(173,high=675,size=(4,3)),
                    columns = ['A', 'B', 'C'])
print(junk,'\n')
print(junk.map(my_func))
```

```
     A    B    C
0  231  426  572
1  497  628  410
2  375  600  577
3  408  206  616

          A         B         C
0  0.211211  0.211957  0.306578
1  0.260593  0.337479  0.200328
2  0.174117  0.322379  0.309452
3  0.198858  0.144310  0.331091
```

Aggregating (reducing) functions with .agg()

`.agg()` *only* accepts functions that take all values along given axis (column/ row) as input & output 1 scalar

- E.g. max(), np.std(), …
- Can pass >1 function with list of function names or dict of row/ column names with functions to apply to them as values
- Use on DataFrames, Series, **& GroupBy objects**

```
grouped2=df.groupby(['detection_method'])
grouped2[['mass_ME','radius_RE', 'orbital_radius_AU', 'orbital_period_yr']].agg('mean')
```

	mass_ME	radius_RE	orbital_radius_AU	orbital_period_yr
detection_method				
Astrometry	4890.840000	12.600000	0.499825	0.726626
Direct Imaging	7929.949333	15.835680	514.123769	40445.285440
Disk Kinematics	795.000000	13.216000	130.000000	957.300000
Eclipse Timing Variations	2154.773529	12.880000	3.962357	9.628240
Gravitational Microlensing	746.775584	10.241521	2.541477	7.065273
Orbital Brightness Modulation	350.513333	9.623000	0.013667	0.003164
Pulsar Timing	205.652857	5.395333	4.897800	17.617327
Pulsation Timing Variations	2385.000000	12.712000	1.700000	2.750000
Radial Velocity	1041.315930	10.031391	2.112706	5.167191
Transit	172.593293	4.111279	0.128524	0.069854
Transit Timing Variations	461.589167	5.698096	0.501715	0.532655

Broadcasting functions with .transform()

`.transform()` broadcasts functions to every cell of data structure that calls it; aggregating functions (e.g. mean, std, sum, etc.) not allowed

- Can pass >1 function with list of names or dict of row/ column names with functions to apply to them as values (like `.agg()`)
    - Can pass lambda functions in dict but not list
- Transforming DataFrame of *n* columns by list of *m* functions yields *hierarchical* DataFrame with *n*×*m* columns (*not* like `.agg()`)
- Use on DataFrames, Series, **& GroupBy objects*;** but **don't modify data in-place!**

```
print(df1)

   a   b   c
0  0   1   2
1  3   4   5
2  6   7   8
3  9  10  11

def funcA(x):
    return x**2+2*x+1
def funcB(x):
    return x**0.5-1
df2 = df1.transform([funcA,funcB])
print(df2)
print(df2.columns)

      a                b                c          
  funcA     funcB  funcA     funcB  funcA     funcB
0     1 -1.000000      4  0.000000      9  0.414214
1    16  0.732051     25  1.000000     36  1.236068
2    49  1.449490     64  1.645751     81  1.828427
3   100  2.000000    121  2.162278    144  2.316625
MultiIndex([('a', 'funcA'),
            ('a', 'funcB'),
            ('b', 'funcA'),
            ('b', 'funcB'),
            ('c', 'funcA'),
            ('c', 'funcB')],
           )
```

If all else fails, there's .apply()

- Slower than `.agg()` or `.transform()`, but more flexible—can handle aggregating, broadcasting, & expanding* functions (*list-like output for each input cell)
- Accepts GroupBy objects, but can err in preserving structure (either groups or columns) because it has to infer function type
  - Error messages may be misleading; e.g. if input or output is not the expected shape, it may raise `TypeError: Unexpected keyword argument` that singles out a legit kwarg of `.apply()`, not an extra kwarg to be passed to the input function
- `.apply()` may be better (& more intuitive) if your function varies by group: `.transform()` receives GroupBy objects in 2 parts—the original columns split into Series, & then the groups as DataFrames—while `.apply()` only receives the groups (like `.agg()`)

The .filter() method (a 1-trick pony)

Conditional selection is trickier for GroupBy object than DataFrames or Series. `.filter()` method helps fill gap.

- Syntax: `GB_obj.filter(func)`
- Input function `func` is evaluated for entire group
- Only groups that collectively return `True` are included in the output
- Group structure not preserved in output

```
temp=df.groupby(['planet_type']).filter(lambda x: len(x) > 5)
print(temp['planet_type'].unique())
print(temp)
```

```
['Gas Giant' 'Super Earth' 'Neptune-like' 'Terrestrial']
                      distance  star_mag  planet_type  discovery_yr
mass_ME  \
#name
11 Comae Berenices b     304.0   4.72307    Gas Giant          2007
6169.20
11 Ursae Minoris b       409.0   5.01300    Gas Giant          2009
4687.32
14 Andromedae b          246.0   5.23133    Gas Giant          2008
1526.40
14 Herculis b             58.0   6.61935    Gas Giant          2002
2588.14
16 Cygni B b              69.0   6.21500    Gas Giant          1996
566.04
...                        ...       ...          ...           ...
...
```

Typical use: filtering groups with too small sample sizes

Windowing Operations

4 methods for evaluating other methods over moving/expanding windows, with similar API to GroupBy objects (most allow similar aggregating methods):

Method	Windowing type	Allows time-based windows?	Allows 2D windows?	Can apply to GroupBy Objects?
`.rolling()`	rolling (aka sliding)	Yes	Yes	Yes
`.rolling(win_type='response_func')`	rolling, weighted by `SciPy.signal` functions	No	No	No
`.expanding()`	expanding (cumulative)	No	Yes	Yes
`.emw()`	exponentially-weighted moving	only with `halflife`	No	Yes

- All methods evaluate from current position back/upward to window size or start of Series.
- All have kwarg `min_periods` to specify minimum number of valid data points in a window.
- All but `expanding()` let you call method on GroupBy objects (to be applied per group).

Iteration

Iteration is **S L O W** ! Use vectorized methods if possible! If not, there are 3 methods:

- `.items()`: gives (index, value) pairs for Series & (column, Series) pairs for DataFrames
- `.iterrows()`: generates pairs of (row_index/label, row contents) where row contents are returned as Series (some dtypes not preserved); orthogonal to `.items()` for DataFrames
- `.itertuples()`: generates iterator of rows packed in `namedtuple()` objects where 1st field is Index & remaining field names are column labels; preserves dtypes & is faster than `.iterrows()`
    - `namedtuple()`: factory function from `collections` module for making tuples with named fields; somewhere between dict & class, but lighter weight than dict

If you want to iterate over groups, the syntax is `for name, group in df.groupby(['col1', 'col2', …]): …` or `for name, group in df.groups: …` but if you group by multiple categories at a time, the `name` iterable will be a tuple of the categories instead of a string.

```python
junk = pd.DataFrame([['eggs',3,'whites',147.8],
                     ['spam',4, 'oz',113.4],
                     ['toast',2,'slices',76]],
                     columns=['food','qtty','unit','g'])
print(junk)
```

```
    food  qtty    unit      g
0   eggs     3  whites  147.8
1   spam     4      oz  113.4
2  toast     2  slices   76.0
```

```python
for itp in junk.itertuples():
    print(itp)
```

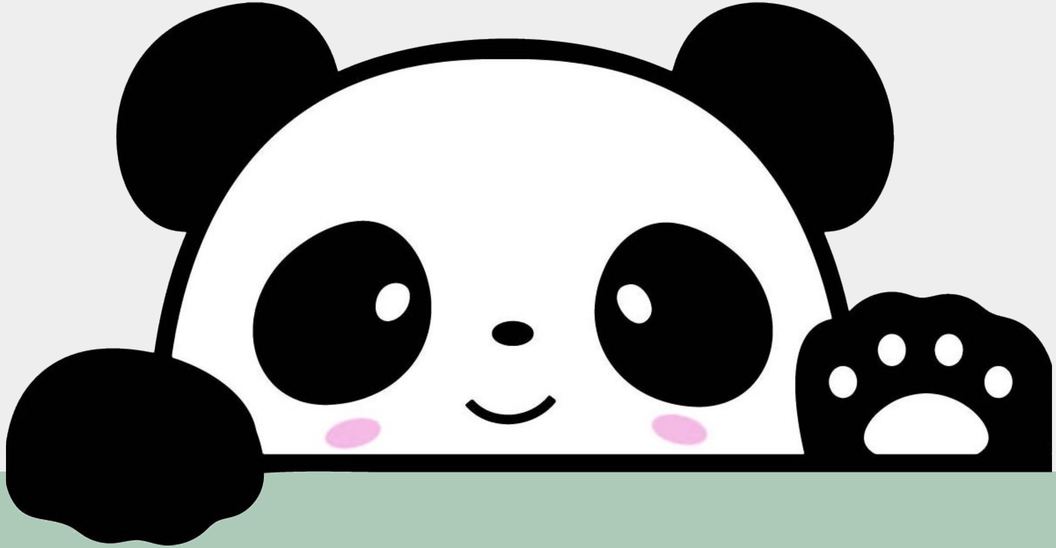
```
Pandas(Index=0, food='eggs', qtty=3, unit='whites', g=147.8)
Pandas(Index=1, food='spam', qtty=4, unit='oz', g=113.4)
Pandas(Index=2, food='toast', qtty=2, unit='slices', g=76.0)
```

```python
for i,c in junk.items():
    print(i,c.values)
```

```
food ['eggs' 'spam' 'toast']
qtty [3 4 2]
unit ['whites' 'oz' 'slices']
g [147.8 113.4  76. ]
```

```python
for i,r in junk.iterrows():
    print(i,r.values)
```

```
0 ['eggs' 3 'whites' 147.8]
1 ['spam' 4 'oz' 113.4]
2 ['toast' 2 'slices' 76.0]
```



Built-in Plotting Methods

The .plot() wrapper method

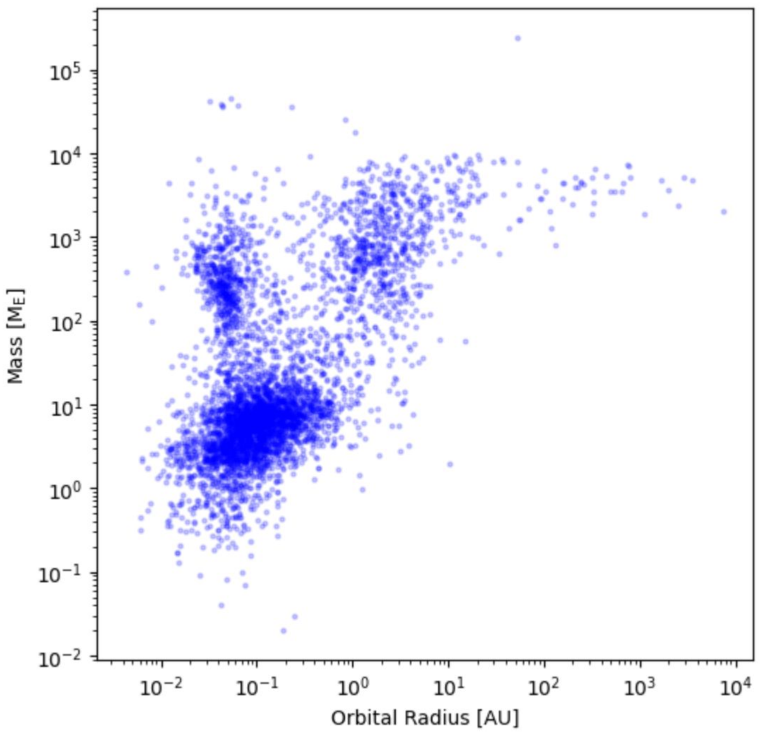
`.plot(kind='line')` or .plot.<kind>() method lets you visualize Series, DataFrames, or Groups (with `.get_group()`) without converting to NumPy

Default plot `kind` is `'line'`. Others available:

- `'bar'` | `'barh'`: bar plots
- `'hist'`: histogram
- `'box'`: boxplot
- `'area'`: area plot (lines filled underneath)
- `'kde'` | `'density'` : Kernel Density Estimation plot `'pie'`: pie plot (don't use this, though)
- `'scatter'`: scatter plot **(not for Series)**
- `'hexbin'`: hexbin plot **(not for Series)**

```
df.plot(kind='scatter', y='mass_ME', x='orbital_radius_AU', loglog=True,
        ylabel='Mass [M$_\mathrm{E}$]', xlabel='Orbital Radius [AU]',
        marker='.', color='b', alpha=0.2,
        figsize=[6,6])
```

```
<Axes: xlabel='Orbital Radius [AU]', ylabel='Mass [M$_\\mathrm{E}$]'>
```



The `.plot()` method continued

- Other kwargs let you control most Matplotlib `figure` & `axis` configurations, including subplots & axis titles
- Most kwargs passable to implemented plot types can be passed as kwargs of `.plot()`, rather than as a dictionary.
- Series can be plotted as lines against their indexes with no args

Limitations:

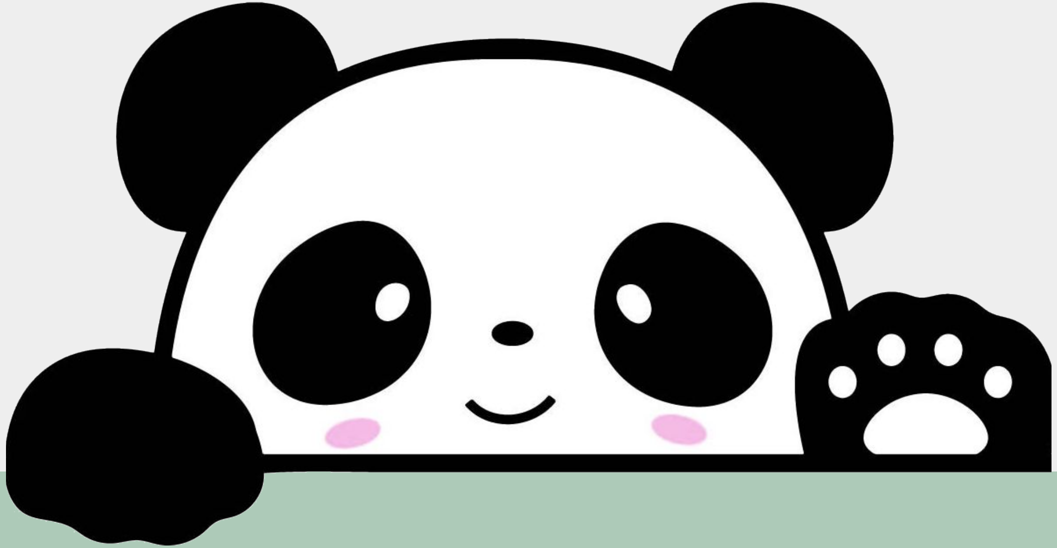
- Limited customization of axis tick labels & scales, & legend location
- Log bin scaling fails for `'hexbin'`
- Only 1 plot style for all subplots per use of `.plot()`
- Passing 2-tuples of columns as `subplots` disables use of `'scatter'` & `'hexbin'`

Other plotting options

Pandas has a `.plotting` submodule with more niche plot types implemented as functions:

- `scatter_matrix()`, `andrews_curves()`, & `parallel_coordinates()`: see if variables are correlated
- `autocorrelation_plot()` & `lag_plot()`: assess randomness of (time) series
- `bootstrap_plot()`: visualize uncertainty of mean, media, & midrange stats
- `radviz()`: like a hybrid of web diagram & k-means clustering scatter plot

Most of what you can do with the `.plot()` wrapper, & much more, can be done better using Seaborn.



Time Series

Time series data types

Pandas incorporates NumPy `datetime64` & `timedelta64` data types, plus object classes from `datetime` & its `dateutil` extension, to define 3 pandas data types & 1 scalar class:

1. **`datetime64[ns(,tz)]`**: data type of a Series of **`Timestamp`**-type scalars (dates only or dates with times), where the default units are ns & a timezone (tz) can optionally be specified; coerced to **`DatetimeIndex`** if used as an Index for a Series or DataFrame

2. **`timedelta64[ns]`**: data type of a Series of **`Timedelta`**-type scalars (associated with a unit that defaults to ns) representing absolute time increments from a start time; coerced to **`TimedeltaIndex`** if used as an Index for a Series or DataFrame

3. **`period[freq]`**: like `datetime64` but typically treated **like bins**, not points; specified by start date & recurrence rate; coerced to **`PeriodIndex`** if set as Index of Series or DataFrame

4. **`DateOffset`**: not its own Pandas data type, but imported implicitly from `dateutil` to combine `timedelta`-like functionality with calendar rules (e.g. to handle leap years or DST)

Scalar Class	Array Class	Pandas Data Type	Pandas Creation Method
`Timestamp` (date only or date & time)	`DatetimeIndex`	`datetime64[ns(, tz)]` (may or may not include time zone)	`.to_datetime(`*dates*`)` or `.date_range(`*start*`, `*end*`=None, `*periods*`=None, `*freq*`=None)` (must specify 2 of 3 kwargs)
`Timedelta`	`TimedeltaIndex`	`timedelta64[ns]`	`.to_timedelta(`*tdelts*`)` or `.timedelta_range(`*start*`=None, `*end*`=None, `*periods*`=None, `*freq*`=None)` (must specify 3 of 4)
`Period`	`PeriodIndex`	`period[freq]`	`.Period(`*t_init*`, `*freq*`=None)` or `.period_range(`*start*`=None, `*end*`=None, `*periods*`=None)` (need 2 of 3)
`DateOffset`	-	-	`.tseries.offsets.DateOffset(`*unit*` = n_units)` (*unit* can be day, month, …)

Parsing dates, timestamps, timedeltas, etc.

Usually you will parse timestamps, timedeltas, etc. from existing data. 2 functions do this:

- `pd.to_datetime(*arg, fmt=None, **kwargs)` accepts str, list, Series, or DataFrame as `arg`, & converts to datetime style in `fmt` (e.g. `fmt='%d/%m/%y %H:%M:%S.%f'`)
  - See `datetime` module docs on `strftime()` and `strptime()` format codes; all units parsed from years* to nanoseconds
  - data with time zones will be converted to UTC with kwarg `utc=True`
- `pd.to_timedelta(*arg, unit=None)` accepts str, list, or Series as `arg`, & tries to parse time increments based on unit kwarg
  - Minimum unit: nanoseconds; maximum unit: weeks (no months or years!)

*Pandas does not accept BCE/BC dates & Julian dates are only partly supported. As a general rule, if your smallest time step is 1+ years, don't bother converting to timestamps

Indexing time series from scratch

3 options if you have to build time series indexes from scratch:

- `pd.date_range(start, end=None, periods=None, freq=None)`: creates `DatetimeIndex` array; exactly 2 of the 3 kwargs after start must be specified.
  - **`start` & `end` must be timestamp strings; float/int are interpreted as Unix times**
  - If *freq* & `periods` are given, then `freq` = time step size & `periods` = number of steps.
  - If *end* & `periods`, then `periods` = number divisions between `start` & `end`.
- `pd.timedelta_range(start=None, end=None, periods=None, freq=None)`: makes `TimedeltaIndex` array; exactly 3 of the 4 kwargs shown must be given (any combo)
  - `start` & `end` must be NumPy timedelta-like (I'll demonstrate; no datetimes!)
- `pd.period_range(start=None, end=None, periods=None, freq=None)`: creates `PeriodIndex` array; must specify exactly 2 of start, end, & periods kwargs.
  - If only `start` & `end` are given, `periods` defaults to days even if `end` is <1 day after `start`

Caveats about precision & date ranges

- Functions like `.date_range()` that take timestamps often also take integers & floats, but assume they are Unix times (time since midnight 01-01-1970), in ns (not Julian dates!)
- All datetimes are 64-bit integer Unix times internally, which limits representable datetimes
- `Timedelta`-creating functions assume time resolution of weeks or better.
- Out-of-range datetimes must be parsed with coarser units with `pd.Timestamp(<datetime>, unit='s')`, or `pd.Timestamp(np.datetime64(<datetime>), unit='s')` for dates >2024 years ago.
- `pd.to_datetime()` can't handle Series of out-of-bounds datetimes, so must use `.apply(lambda x: pd.Timestamp(x))`
- If your time resolution is months or years at best, Pandas time series functions probably aren't worth your time.

```python
print(pd.Timestamp.min, '=', pd.Timestamp.min.value,
      'ns in Unix time, or\n', bin(pd.Timestamp.min.value), 'in binary.\n')
print(pd.Timestamp.max, '=', pd.Timestamp.max.value,
      'ns in Unix time, or\n', bin(pd.Timestamp.max.value), 'in binary.')
```

```
1677-09-21 00:12:43.145224193 = -9223372036854775807 ns in Unix time, or
 -0b111111111111111111111111111111111111111111111111111111111111111 in binary.

2262-04-11 23:47:16.854775807 = 9223372036854775807 ns in Unix time, or
 0b111111111111111111111111111111111111111111111111111111111111111 in binary.
```

Resampling time series

Resampling = interpolating data from one time series to another with different spacing

- **Upsampling** = resampling to more closely spaced time steps
- **Downsampling** = resampling to more widely spaced time steps

Method is **`.resample('<unit>')`** to ***shift* or *downsample*** Series & DataFrames.

`.resample()` is a time-based GroupBy, so **most aggregate GroupBy methods (e.g. `sum()`, `mean()`, …) can be called on the result**

Upsampling:

Upsampling usually requires interpolation & does not play well with uneven time steps

- If NaNs at intervening timesteps are OK, use `.resample('<unit>').asfreq()`
- `.resample('<unit>').ffill(limit=None)` fills intervening timesteps (up to limit) with most recent non-NaN value
- `.resample('<unit>').interpolate(method='linear')` can fill intervening time steps with any SciPy interpolation method if output timesteps align with input (see demo); otherwise need to interpolate data separate from times

The .dt accessor

Series objects have `.dt` accessor (like a super-attribute) that can return datetime properties of time Series

- Returns are also Series with same indexes as original Series
- `.dt.<unit>` called on any datetime or timedelta Series returns just the <unit> part of the timestamps
    - must spell out (`<unit>` e.g., `.dt.nanosecond`, not `.dt.ns`)
    - Allows selection & filtering like so, if *values* (not index) are datetimes: `ser[ser.dt.day == 2]`
- `.dt` also has `.round(<unit>)`, `.ceil(<unit>)`, & `.floor(<unit>)` methods to round datetimes to given unit

```
0   2013-01-01 09:10:12
1   2013-01-02 09:10:12
2   2013-01-03 09:10:12
3   2013-01-04 09:10:12
dtype: datetime64[ns]

In [274]: s.dt.hour
Out[274]:
0    9
1    9
2    9
3    9
dtype: int32

In [275]: s.dt.second
Out[275]:
0    12
1    12
2    12
3    12
dtype: int32

In [276]: s.dt.day
Out[276]:
0    1
1    2
2    3
3    4
dtype: int32
```

The .dt accessor continued

- Most attributes/functions callable on time Series with `.dt` can be called directly on `DatetimeIndex`, `TimedeltaIndex`, or `PeriodIndex` without `.dt`
- `Series.dt.to_period(freq)` converts `DatetimeIndex`-type Series or other Series of timestamps to `PeriodIndex`-type, where `freq` is any accepted period alias string
- `Series.dt.to_timestamp(how='s')` converts `PeriodIndex`-type Series to `DatetimeIndex`
    - `how` kwarg specifies use of start (`'s'`, default), or end (`'e'`) of each period as timestamp
- Series with `Timedelta`-type indexes have `.dt.components` attribute that returns DataFrame expansion of units

```
tt = pd.Series(pd.timedelta_range(start='1 day 12:15:00', end='3 day 21:45:00', periods=35))
tt.dt.components.head(2) #also for datetimes since a few months ago
```

	days	hours	minutes	seconds	milliseconds	microseconds	nanoseconds
0	1	12	15	0	0	0	0
1	1	13	56	28	235	294	117

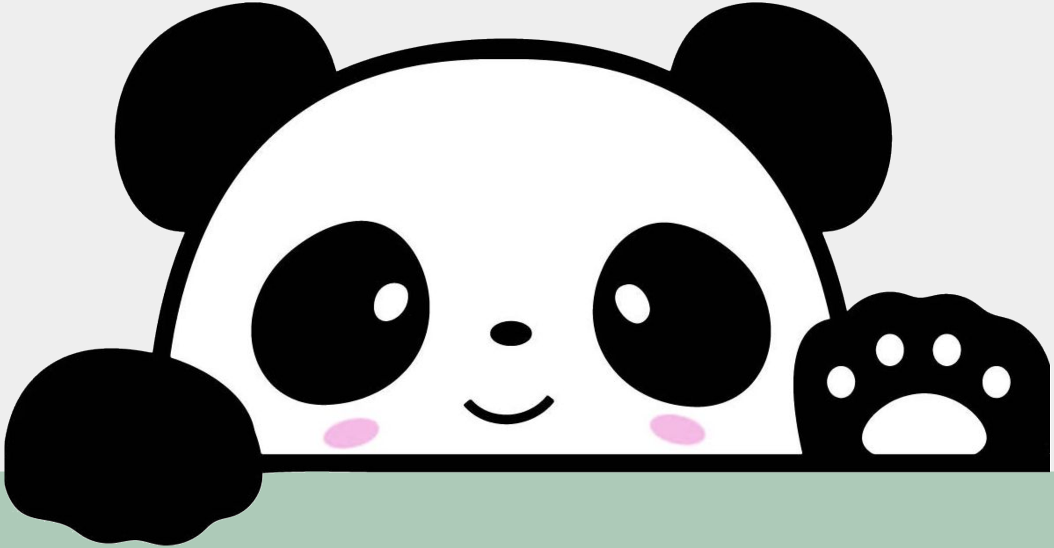
Shifting times & timezones

Need to change `datetime` or `period` indexes? Skip `.reindex()` or manual replacement: just use `.shift(periods=1, freq=None)`

- As usual, `periods=`no. of units to shift by (default 1), & *`freq`*`=`unit (default is smallest unit needed to represent timestamp or period, usually ns)
- Can call directly on Series or DataFrame without extracting Index
- Choosing too small a shift unit can cause loss of first & last data points

Got time zones? Remember these methods:

- **`.tz_localize`**`(<TZ>)`: append to datetime, DatetimeIndex, PeriodIndex, Series, or DataFrame of datetimes to
  a. Assign a time zone, or
  b. if None is passed explicitly, remove all time zone information.
- **`.tz_convert`**`(<TZ>)`: append to same data structures as above to
  a. Convert to specified time zone, or
  b. if None is passed, *convert to UTC & then* remove all time zone information.



(More) Advanced Topics

Preparing data as machine learning input

ML Programs like TensorFlow & PyTorch take Series & DataFrames, but need categorical variables coded as boolean or numeric.

Use `pd.get_dummies(df, dtype=bool, columns=['col1', 'col2', …])`.

For DataFrame `df` (no Series!), given categorical variable V with n unique values c_1, c_2, …, `pd.get_dummies(df[['V', …]])` returns n columns (or n-1 with `drop_first=True`) titled something like `V_c1`, `V_c2`, … with `True` where V=c_n in that row, `False` otherwise.

Memory management tips:

- **Keep `dtype=bool`**. Booleans are 1-bit, `int` & `float` are 8-bit minimum.
- One column of categorical data can use as much memory as several columns of dummy variables
  - Save on memory by using `pd.get_dummies()` & dropping the original column
- Avoid the reverse function, `pd.from_dummies()`

Efficient storage with the `Categorical` type

Have a string- or int-type DataFrame column or Series with few unique values? Convert to `Categorical` type & reduce memory usage by factors of 10 or more!

- `ser = pd.Series(data, dtype='category')`—initialize `Categorical` Series
- `df['cat_col'] = df['cat_col'].astype('category')`—convert to `Categorical`
- `pd.Categorical(data, categories=['your','cats','here'], ordered=False)`—
  Create raw `Categorical` data (e.g., as Index); can assert assigned order with `ordered=True`

What this does: takes list of unique values, maps them to integer codes, & stores codes at column's location in memory with smallest possible bit size, only filling in values in print

- Only string vectorized functions are supported (no numeric functions!)
- Not to be used as input for AI/ML programs

The Categorical type continued

Get attributes of `Categorical` data with `.cat` accessor:

- `.cat.categories`—get Index-type list of categories
- `.cat.codes`—view DataFrame or Series with code numbers in-place

Categories can also be added, removed, rearranged, & renamed as needed.

- Data that do not match any assigned categories are set to NaN

- `.cat.add|remove_categories([cats])`— add to/subtract from existing list
- `.cat.remove_unused_categories()`—automatically remove categories with no data (coded as -1 until matching data are added)
- `.cat.rename_categories([new])`—rename categories by list or dict
- `.cat.as_ordered|as_unordered()`—fix or unfix current category order

Cuts & Intervals

There are also 2 functions to discretize (bin) numerical data (e.g. for age brackets):

- `pd.cut(data, bins, right=True, labels=None)`: provide either *n* equal `bins` or array of variable `bins` & optionally `labels` for each; `right=True` indicates bins are half-open on right (False excludes both bin edges)
- `pd.qcut(data, q, **kwargs)`: same as `cut` but for `q` quantiles (`q` can be int or list of right or left edges)

- In both cases, input data must be Series or array-type (1D), & output is a Series of either `Categorical` (with `labels`) or `Interval`-type
    - `Intervals` are just bin-like Index objects; attributes report edges, midpoint, & edge openness
- Raw `Categorical` & `Interval` objects (analogous to Index) have few methods/attributes, are mainly meant to be args of `df.group_by()`

Sparse Arrays

DataFrames with many mostly-NaN rows or columns can be stored in Sparse form to save memory

- Initialize Series or DataFrames as `SparseDtype` with kwarg **`dtype= SparseDtype`**`(dtype=np.float64, fill_value=None)`
  - Or call method **`.astype( pd.SparseDtype`**`("float", np.nan)`**`)`**
- **`pd.arrays.SparseArray(data, **kwargs)`**—initialize SparseArray with dense array data input (rare)

Sparse arrays have **`.sparse`** accessor (like `.dt`, `.cat`, etc.) with following methods/attributes:

- `df.`**`sparse.density`**—print fraction of data that are non-NaN
- `df.`**`sparse.fill_value`**—print fill value
- `df.`**`sparse.from_spmatrix`**`(data)`—make new SparseDtype DataFrame from SciPy sparse matrix
- `df.`**`sparse.to_coo`**`()`— convert df to sparse SciPy COO type

All NumPy universal functions (`.abs()`, `.std()`, etc) still work on Sparse Arrays

.pivot() & Hierarchical DataFrames

Have lots of categorical variables? Can be efficient to reshape DataFrame with `.pivot(index=indexes, columns=columns)` method on columns &/or rows with repetitive data.

- Result will be **Hierarchical DataFrame** with multi-**level** rows & columns
- Can also use `.pivot_table(*args, aggfunc=functions)` method if df has duplicate rows &/or to apply 1+ aggregating functions to data during pivot

```
more_junk
```

	foo	bar	baz	qux
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	p
4	two	B	5	q
5	two	C	6	r

```
more_junk.pivot(index='foo',
                columns='bar')
```

	baz			qux		
bar	**A**	**B**	**C**	**A**	**B**	**C**
foo						
one	1	2	3	x	y	z
two	4	5	6	p	q	r

MultiIndexing in hierarchical DataFrames

3 ways to select data in hierarchical DataFrames, whose indexes are `MultiIndex`:

1. `hdf.loc[(row_lvl0, row_lvl1, …), (col_lvl0, col_lvl1, …)]` (Can drop () if only 1 level of rows/columns)

2. `hdf.xs(key, level=0, axis=0)` returns "cross-section" at level & axis of `key` (`key` can be 1 label or a tuple)

3. **"Partial" selection:** column-major nested-list-like or ndarray-like syntax (allows only 1 key per level)

- Both rows & columns of hierarchical DataFrame have `.levels` attribute to view what levels exist in what order
- Can use `hdf.reorder_levels([new, indexes], axis=0)` to rearrange levels
- `.sort_index(level=L)` & other Index methods can work with `level` kwarg
- If all levels are named, you can group by one or more of them
- Can flatten with `.melt()` (sort of)

Hierarchical DataFrames continued

- Can create MultiIndex with `pd.MultiIndex.from_<struct>()` where `<struct>` = `tuples`, `arrays`, `frame` (for DataFrames), or `product` (for Cartesian product of exactly 2 lists)
- If all index/column levels are named, you can **group by** them with selection syntax like in `.loc[]` & call aggregate functions (hard to get order right)

```python
print(new_hdf)
new_hdf.groupby(['value', ('qux', 'A')]).sum()
```

```
                baz         qux
bar               A   B   C   A   B   C
color value
R     75         43   9  45 cat  57  40
      150        19   4  39 cat  27  56
G     75          5  48  35 cat  51   9
      150        16  34  17 cat  18  24
B     75         22  55  32 dog  58   9
      150         8  28  47 dog   3  49
```

		baz			qux	
	bar	A	B	C	B	C
value	(qux, A)					
75	cat	48	57	80	108	49
	dog	22	55	32	58	9
150	cat	35	38	56	45	80
	dog	8	28	47	3	49

Notes on parallelization & (new) HPC features

- Built-in functions allow parallelization via `Numba`, with `engine='numba'` & `engine_kwargs={"parallel": True}` in kwargs. Example below.
  - More advanced users can write their own JIT-compiled or Cython functions as detailed in Pandas documentation on [Enhancing Performance]()
- Support for chunking (loading & working on subsets of data) with [Apache Parquet]() input files, JSON input files, & the [PyArrow ChunkedArray]() type
  - Pandas `ArrowExtensionArray` & `ArrowDtype` data types are still experimental

```
import numba
```

```
numba.set_num_threads(4)
stuff =  df.iloc[:,4:9].sample(n=2500000, replace=True, ignore_index=True)
%timeit stuff.rolling(500).mean()
%timeit stuff.rolling(500).mean(engine='numba', engine_kwargs={"parallel": True})
```

```
146 ms ± 524 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
71.1 ms ± 1.53 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

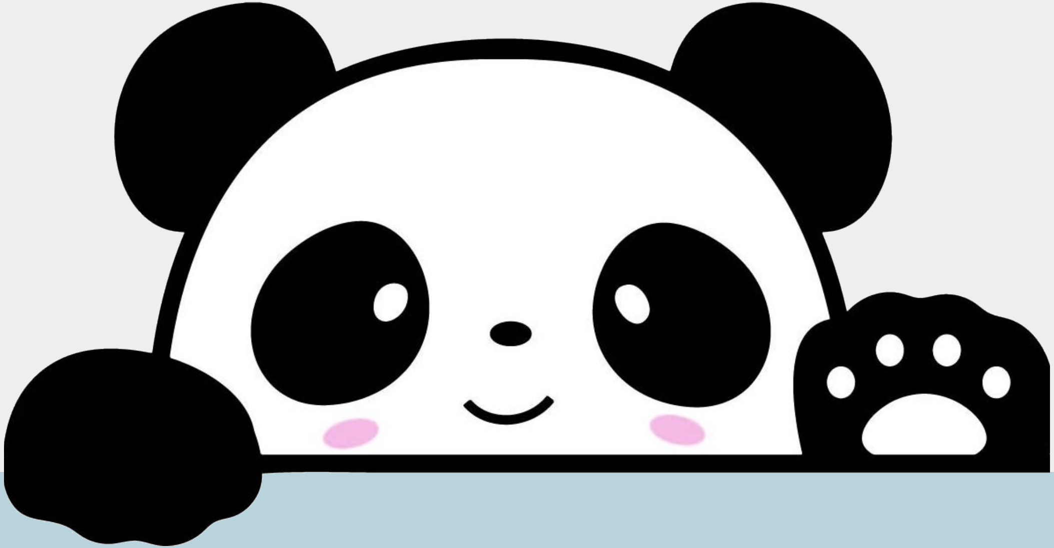
Summary (outline rehash)

Day 1

- What is Pandas? Why use it?
- Object classes & data types
- Basic input/output
- **Inspection & cleaning**: selection & filtering, handling missing data
- **Basic Operations:** stats, binary, vectorized math & string methods
- Sorting, reindexing, & merging

Day 2

- Intro to GroupBy objects
- **More Operations:** comparing data, complex &/or user-defined functions, windowing, & iteration
- Built-in plotting methods
- Time series functionality
- **Advanced topics:** ML prep, memory-saving data types, MultiIndexing & hierarchical DataFrames



Bye-bye!