Introduction to Pandas for Data Science

Rebecca Pitts

Lund University | LTH Faculty of Engineering

Course Outline

Day 1

- What is Pandas & how to load it
- Main object classes
- Basic input/output
- Selection, inspection, & cleaning
- Built-in Operations
- Rearranging data

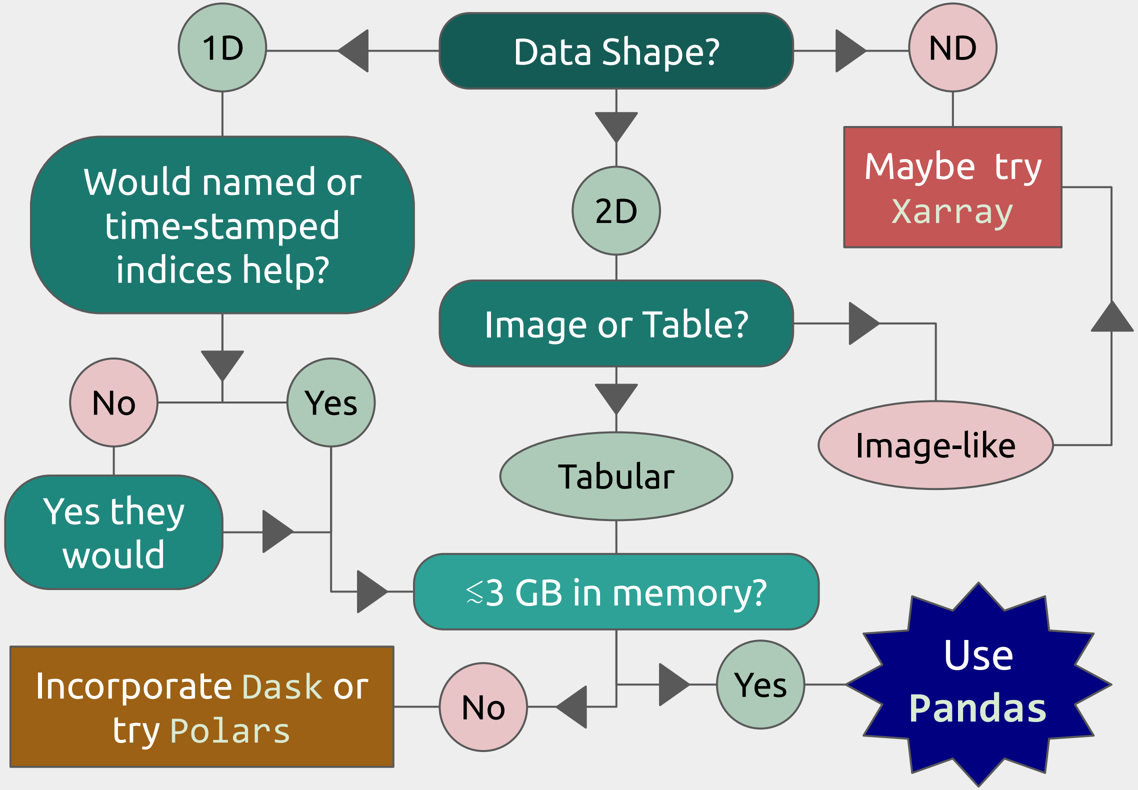
Day 2

- GroupBy objects
- Complex &/or user-defined functions
- Built-in plotting methods
- Time series functionality (if time)
- Advanced topics (e.g. ML prep)

What is Pandas? Is it right for my data?

Pandas = **PAN**el **Data AnalysiS**

Python data library for cleaning, organizing, & statistically analyzing large data sets



Pros and Cons of Pandas

Pros

- Powerful (100s of built-in functions, native multithreading)
- Flexible (dozens of I/O formats)
- Easy to use
- Interfaces with many other packages

Cons

- Sometimes inconsistent syntax
- Hard to handle >2D structures
- Parallelization usually requires other packages

Find, Load, & Import Pandas

Terminal or job script: modules depend on whether you use Anaconda or PyPi

Anaconda: `ml Anaconda3` usually loads everything you need. If not…

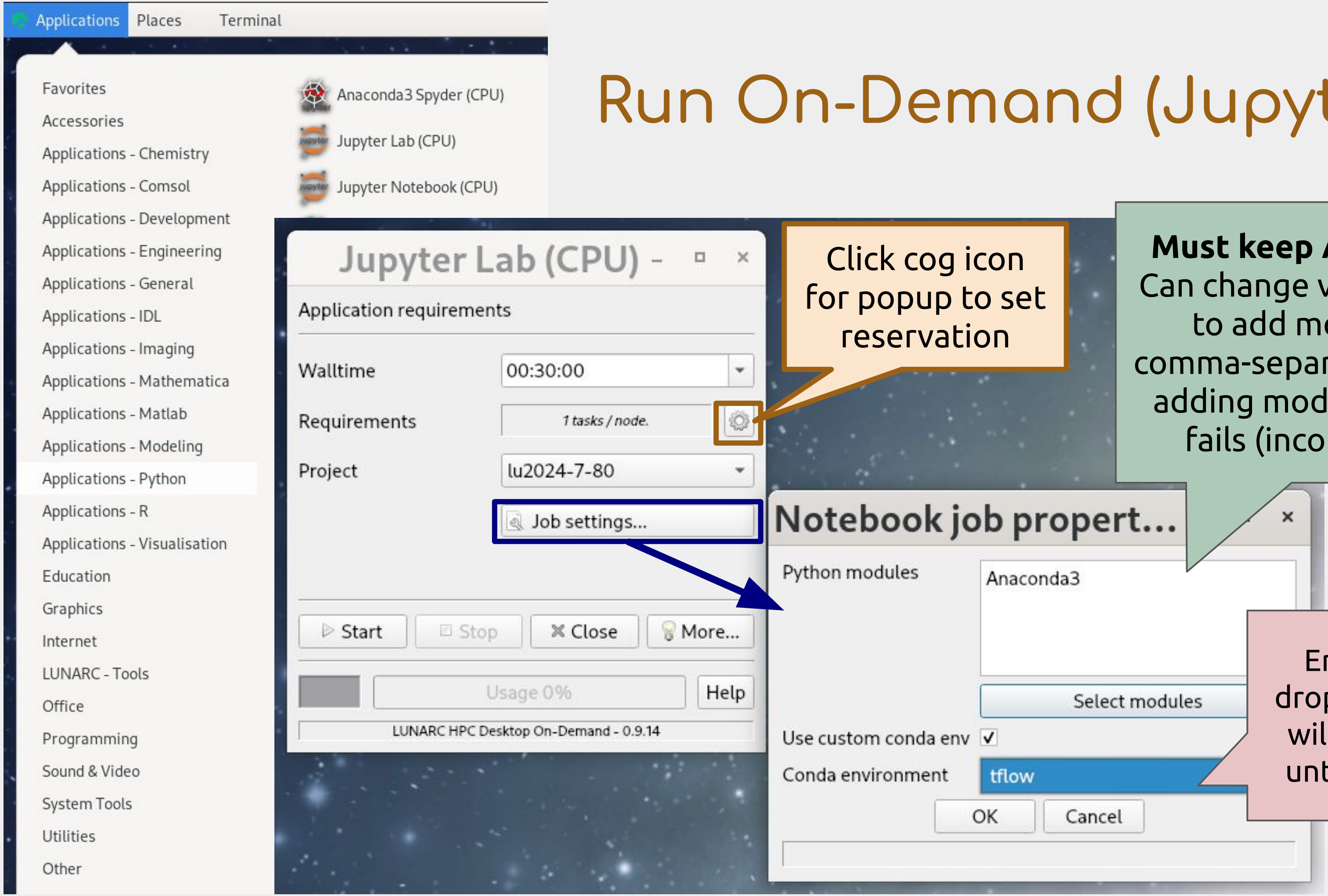
- If the other packages you need are installed, try PyPi versions instead
- Or, create conda environment* & install what you need there

*On-Demand IDEs (Jupyter, Spyder, etc.) use `Anaconda3` – need environments for incompatible or additional packages

PyPi: At most centers, `Pandas` is in `SciPy-bundle` (depends on `GCC` & `MPI`)

- Use `ml spider <dependency>` to check Python or other dependencies for which `GCC` &/or `MPI` you need, then load them
- Use `ml avail SciPy-bundle` to get version number(s), & load/add to script

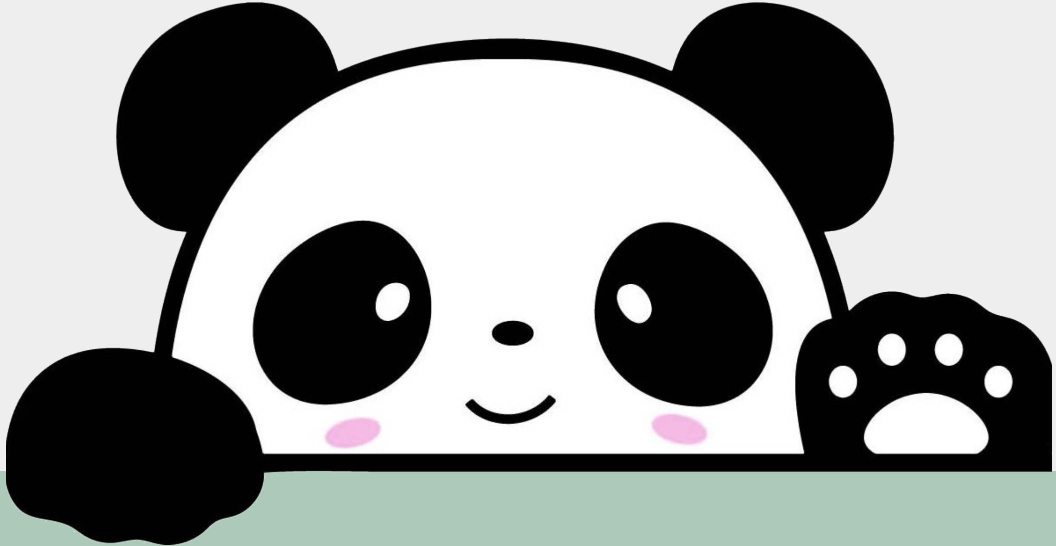
Run On-Demand (Jupyter Lab)



Click cog icon for popup to set reservation

Must keep Anaconda3! Can change version or try to add modules to comma-separated list, but adding modules usually fails (incompatible)

Environment drop-down menu will look empty until you click it



Pandas Object Classes & Data Types

Main Pandas object classes

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

- 1D array with customizable indices

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

- 2D array of series aligned side-by-side

Similar attributes & methods for both classes

- https://pandas.pydata.org/docs/reference/series.html
- https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html

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`

Basic Series & DataFrame Attributes

- `df.index`—list of row labels or numbers
- `df.columns`—list of column labels
- `df.values`—returns `df` as NumPy array (can also call on `.columns` & `.index`)
- Many, MANY more—start at https://pandas.pydata.org/docs/user_guide/10min.html

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

Code-Along: Create a DataFrame

1. Load Pandas & Numpy
2. Make an array with 4 rows & 3 columns, with data values of 1-12
3. Convert it to a DataFrame & label columns [‘a’, ‘b’, ‘c’]

```
import pandas as pd
import numpy as np

a = np.arange(1,13).reshape((4,3))
adf = pd.DataFrame(a, columns=['a', 'b', 'c'])
print(adf)
```

```
    a   b   c
0   1   2   3
1   4   5   6
2   7   8   9
3  10  11  12
```

Index-class objects

`Index`-class objects, (like `df.index` & `df.columns`) are *immutable, hashable* sequences used to align data for easy access. They have…

- Subclasses vary by data-type
- **Many** Series-like attributes (e.g. `.dtype`) & set methods, but Index methods only return copies.

Code-Along

Get indices of `df` in previous example & print the data-type. See https://pandas.pydata.org/docs/reference/api/pandas.Index.html#pandas.Index

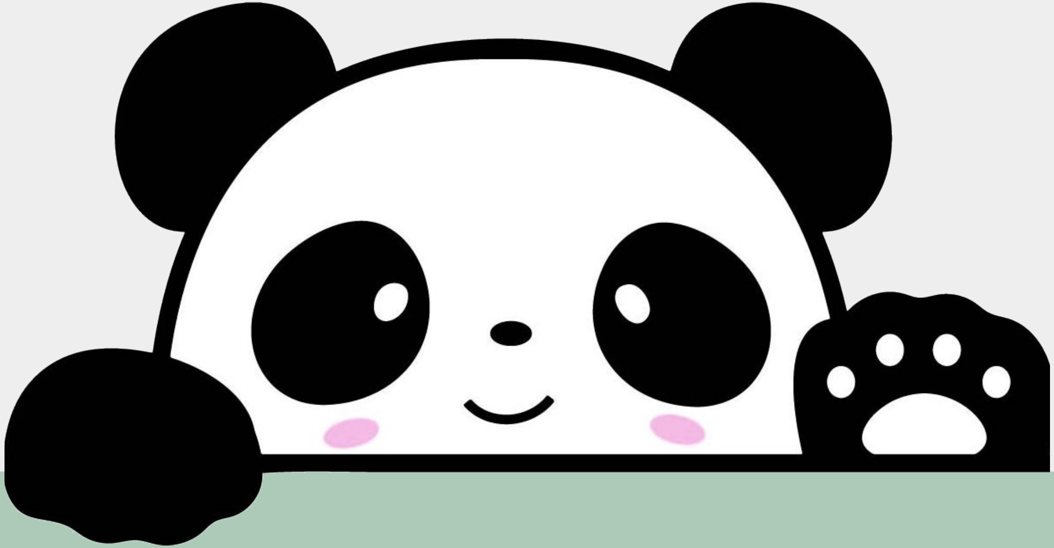
```
print(adf.index)
print(adf.index.dtype)
```

```
RangeIndex(start=0, stop=4, step=1)
int64
```

Quick aside on row & column labels

Pandas documentation has multiple terms for row & column labels:

- "**Indices**" typically only refer to row numbers, but may refer to non-numeric row labels, or column indices if columns are accessed by position
- "**Columns**" may refer to labels & contents of columns collectively, or labels only
- "**Keys**" may refer to column labels, or occasionally both column & row labels, especially in SQL-like commands
- A column label may be called a "**name**", after the optional Series label



Input/Output

Basic I/O

Type	Data Description	Reader	Writer
text	CSV (or any ASCII text with a standard delimiter)	`read_csv(path_or_url, sep=',', **kwargs)`	`to_csv(path, sep=',', **kwargs`
text	Fixed-Width Text File	`read_fwf(path, **kwargs)`	N/A
binary	MS Excel	`read_excel(path_or_url, sheet_name=0, **kwargs)`	`to_excel(path, sheet_name=...)`

& many more. See: https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html

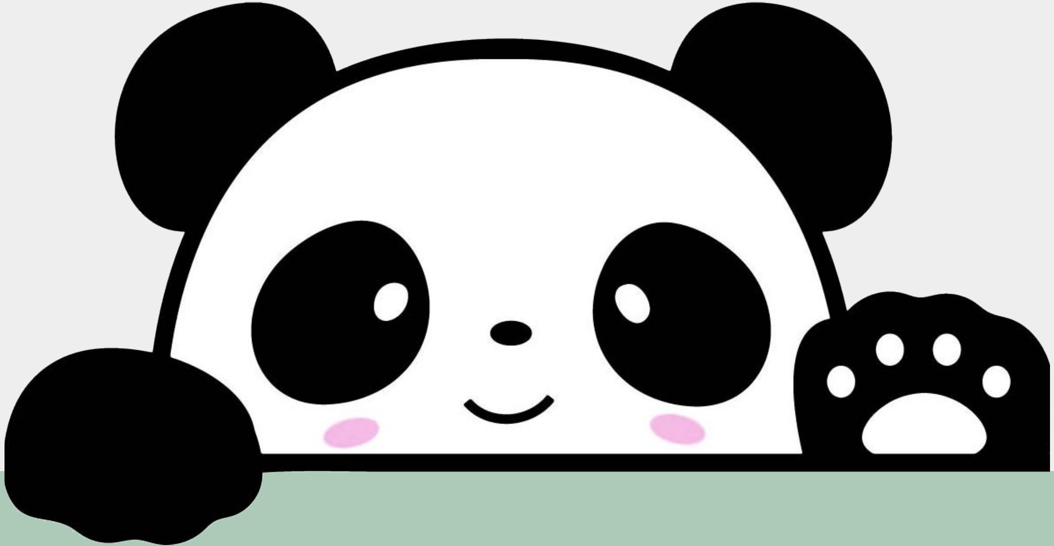
- Most readers assume top row contains column labels, but can override
- 0-based indexes assigned by default, but can set `index_col`
- Can also convert to/from NumPy arrays, structured arrays, or dictionaries.

Code-Along: Basic I/O

1. Load the file ‘exoplanets_5250_EarthUnits.csv’ into a dataframe
   a. Extra: set the index to the leftmost column
2. Write the DataFrame to a tab-separated text file (.txt)

```
exos = pd.read_csv('exoplanets_5250_EarthUnits.csv', index_col=0)
exos.to_csv('exos_5250_EUnits.txt', sep='\t')
```

If you didn’t figure out how to set the indices to the first column in the loading step, call `.set_index(‘#name’)` on your DataFrame now. We will use this DataFrame in more examples later



Inspection & Cleaning

Data inspection convenience functions

- `df.head()` & `df.tail()` print 5 or n rows from start or end of `df`
- `df.info()` summarizes contents & data types in `df`
- `df.describe()` prints naive statistics for all numeric columns in `df`
- `df.nunique()` prints number of unique values in each column
- `df.value_counts()` prints unique values & number of occurrences per permutation of selected rows/columns (not typically useful for all of `df`)
- And so many more! See [pandas.DataFrame API reference]

Code-Along: Data Inspection

1. Call `.info()` off the exoplanets DataFrame from the previous example. Familiarize yourself with the specs that are included. Notice anything weird?
2. If nothing stuck out in #1, call `.describe()` on that DataFrame.

Solution:

At least 2 columns have labels suggesting their contents should be `float64` type, but instead they are `object` type & are not evaluated by `.describe()`. More on this shortly.

The memory_usage() function

`df.memory_usage()` prints sizes of each column of `df` in memory, in bytes

- Numeric & boolean data are fixed size in bytes—stored within `df` in memory
- Object-type data (strings) are NOT fixed in size—only *pointers* stored with `df`; *values* are elsewhere & much larger in memory
- Must use `memory_usage(deep=True)` to estimate memory used by string values (values will be upper bounds)

Don't rely on `df.info()` to monitor memory use! Size reported by `df.info()` is sum of `memory_usage(deep=False)`

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 cell (scalar output)	`df.at['row','col']` or `df.iat[i,j]`
column(s) by name	`df['col']` or `df[['col0', 'col1', ...]]`
row(s) by index	`df.iloc[i]` or `df.iloc[i:j]`
rows & columns by name	`df.loc[['rowA','rowB', ...], ['col0', 'col1', ...]]`
rows & columns by index	`df.iloc[i:j, m:n]`
columns by name & rows by index	**You can mix .loc[] & .iloc[] for selection, but NOT for assignment!**

Use "`:`" to select all rows or all columns in `.loc[]` or `.iloc[]`

Conditional Selection

Binary comparison operators (>, <, ==, =>, =<, !=) & most logical operators can be used in `[]` of `df[...]`, `df.loc[...]`, or `df.iloc[...]` provided:

- Bitwise logical operators (`&`, `|`, `^`, `~`) must be used instead of plain-English versions (`and`, `or`, `xor`, `not`)
- When 2+ conditions are specified, each condition must be enclosed by `()`
- Use `.isna()` or `.notna()` to check for invalid data, & `.isin()`, `.notin()`, or `.str.contains()` to look for substrings.

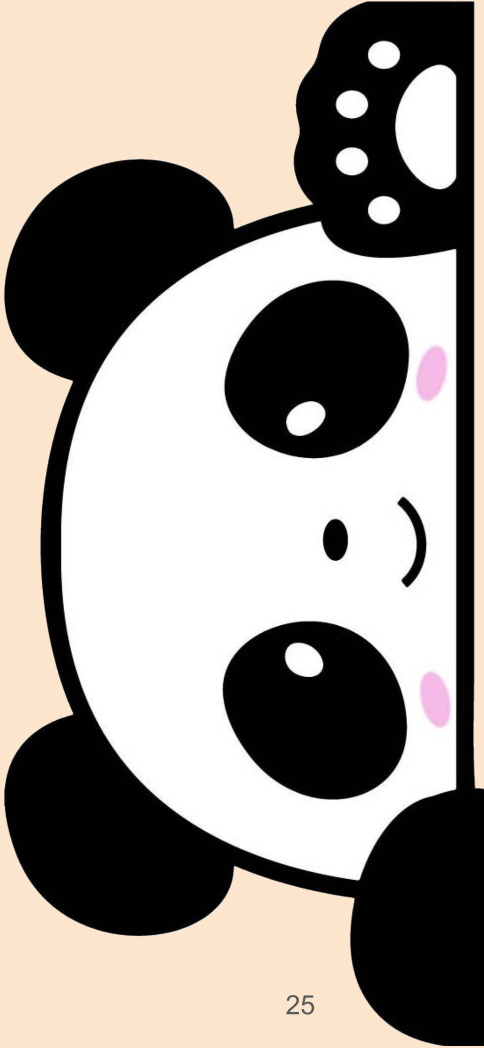
https://pandas.pydata.org/docs/user_guide/10min.html#boolean-indexing describes syntax

Code-Along: Selection Syntax

1. Assign rows 25 through 35 of the exoplanets DataFrame to another DataFrame called `df2`.
2. Print both the “`planet_type`” & “`mass_ME`” columns of `df2`.
3. Go back to the full exoplanets DataFrame, select rows where the “discovery_yr” is before 2007 and “planet_type” is not “Gas Giant”, & print only the “planet_type” column of that selection.

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

Exercise time!



Finding & handling missing/invalid data

- Check for missing data with `.isna()` & `.notna()` methods
  - Pandas assumes all whitespace is intentional; **numeric columns with white spaces are `object` type & `.isna()` ignores them**
  - Must convert to NumPy to find +/-`inf`
- Use `.dropna(axis=axis)` to remove whole rows/columns containing invalid entries (usually not necessary)
- Use `.fillna()` to replace NaNs with fixed or interpolated values

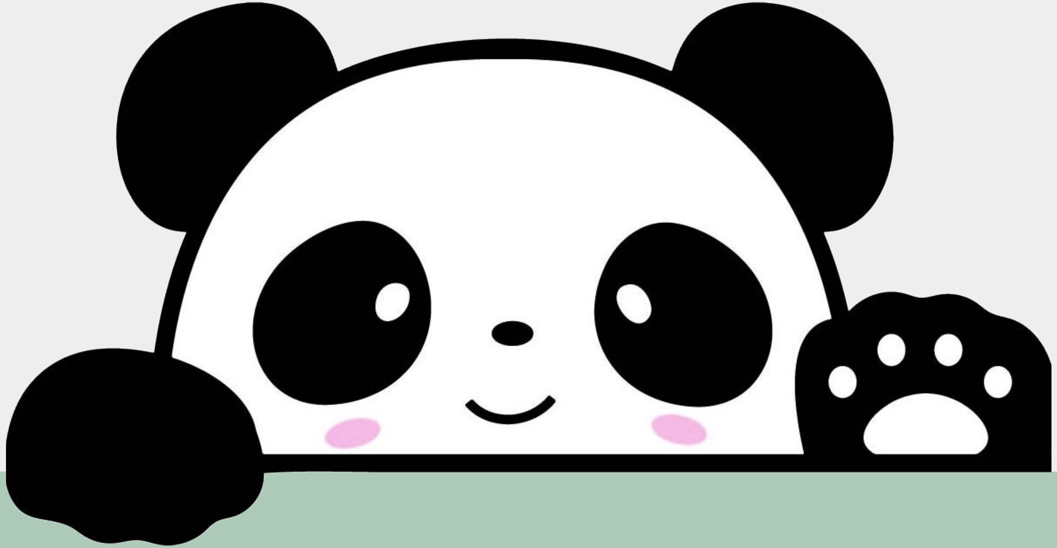
Cleaning anomalous or duplicate data

- `df.drop_duplicates()` finds identical rows & removes all but 1
  - Use `subset` kwarg to remove duplicates by specific columns (more aggressive).
- `df.drop(data, axis=axis)` removes unneeded columns by name or index
- `df.mask(condition, other=None)`, masks bad *numeric* data where `other` (default NaN) can pass replacement values/Series/DataFrames
- `df.replace(to_replace=old, value=new)` can replace almost anything (see docs for ways `old` & `new` can be used)

Most DataFrame/Series methods have **`inplace`** kwarg (default `False`) to toggle copy or overwrite

Code-Along: Bad Data Handling

1. The exoplanets DataFrame has space characters (‘ ’) for missing values in the ‘`mass_ME`’ & ‘`radius_RE`’ columns. Use the `.replace()` method to replace ‘ ’ with `np.NaN` in-place.
   a. Extra: use `.astype()` to convert the data types of these columns to ‘`float64`’ in-place (you’ll have to reassign those columns). Check your results with `.info()`
2. Use the `.mask()` method to mask the ‘`eccentricity`’ column wherever the data are exactly 0 (these are assumed values). The "Non-Null Count" for that column in `.info()` should be smaller.



Basic Operations

Vectorized String Methods

Most built-in string methods can be applied column-wise to Pandas data structures using .str.<method>(). Scroll through methods of `pandas.Series.str` in left menu panel at https://pandas.pydata.org/docs/reference/series.html#accessors

- `.str.replace()` does not accept `dict` input where keys are existing substrings & values are replacements; use the general `.replace()` (without `.str`) instead
- `.str.split()/rsplit()` can make 1 column of lists or >1 column of substrings

`.str[...]` accessor also allows indexing of lists or strings in DataFrame cells

Math & Statistics

Series & DataFrame objects have most NumPy ufunc (universal function) methods, a few SciPy methods, & cumulative-sum & -product methods. See https://pandas.pydata.org/docs/user_guide/basics.html#descriptive-statistics

- Methods require no args/kwargs for Series, but **for DataFrame, you must set `numeric_only=True` & double-check `axis`**
  - `axis=0` or `'index'`: columns preserved, indices collapse (default)
  - `axis=1` or `'columns'`: indices preserved, columns collapse
- `.describe()` computes many stats for all numeric columns automatically, but treats integers as floats

Broadcasting basic arithmetic

- Can do vectorized arithmetic with normal operators (`+`, `-`, `*`, `/`, `**`, and `%`) between a Series/DataFrame & a scalar, or 2 Series/ DataFrames of the same shape (e.g.: `df/100.`, `df**-1.5`, `dfA+dfB`, …)
- To broadcast arithmetic between Series & DataFrame, use functions in https://pandas.pydata.org/docs/user_guide/basics.html#matching-broadcasting-behavior
- **Performance tip:** install & import `numexpr`, & wrap expression `expr` with `pd.eval(expr, engine='numexpr')` to automatically multi-thread
  - Only for scalar/element-wise functions on pure `float64`/`int64` Series/DataFrames

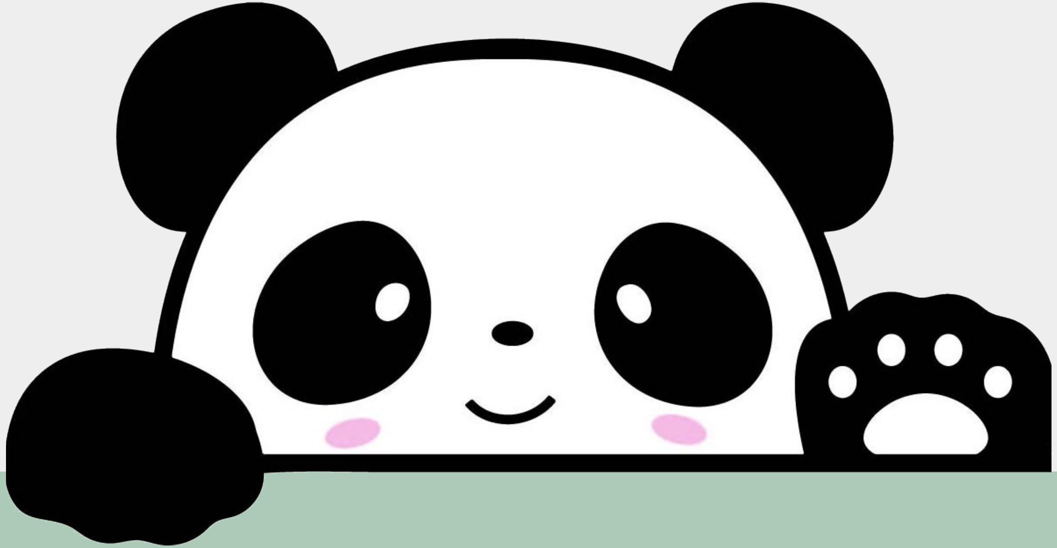
Comparative methods

- Regular operators work to compare Series/DataFrames to scalars, but elementwise comparison requires special operators. See https://pandas.pydata.org/docs/user_guide/basics.html#flexible-comparisons
    - Also recommend reading https://pandas.pydata.org/docs/user_guide/basics.html#boolean-reductions
- Use `df1.compare(df2)` to find & print differences between 2 **identically indexed** Series or DataFrames (same row/column labels in same order)
    - `.compare()` does not check type; use `pd.testing.assert_frame_equal(df1, df2)` or `pd.testing.assert_series_equal(df1, df2)` & see if it raises `AssertionError` for differently-typed identical values

Code Along: Basic Operations

1. Calculate the median of all numeric columns in the exoplanets DataFrame
2. Divide the “`mass_ME`” column by 317.8 & assign the result to “`mass_MJ`” (planet mass in Jupiter masses)

You’ll use the string & comparative methods in the exercises.



(Re)organizing & Merging Data

Sorting & Reindexing

- Can sort Series & DataFrames by 1 or more indices, values, or both. See https://pandas.pydata.org/docs/user_guide/basics.html#sorting
- If indices or columns are missing, `.reindex()` can add & sort them simultaneously, or change indices by assignment (not in-place)
  - Can also select data where labels may not all exist, without raising errors
  - https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.reindex.html#pandas.DataFrame.reindex
- `df1.reindex_like(df2)` makes empty DataFrame with same row & column labels as `df2` & inserts values from `df1` where labels are shared with `df2`

Combining data structures

Pandas functions:

1. `.concat()`: concatenate ≥2 Series or DataFrames on shared axis
2. `.merge(left, right, how='inner')`: join 2 DataFrames or Series on columns with SQL logic

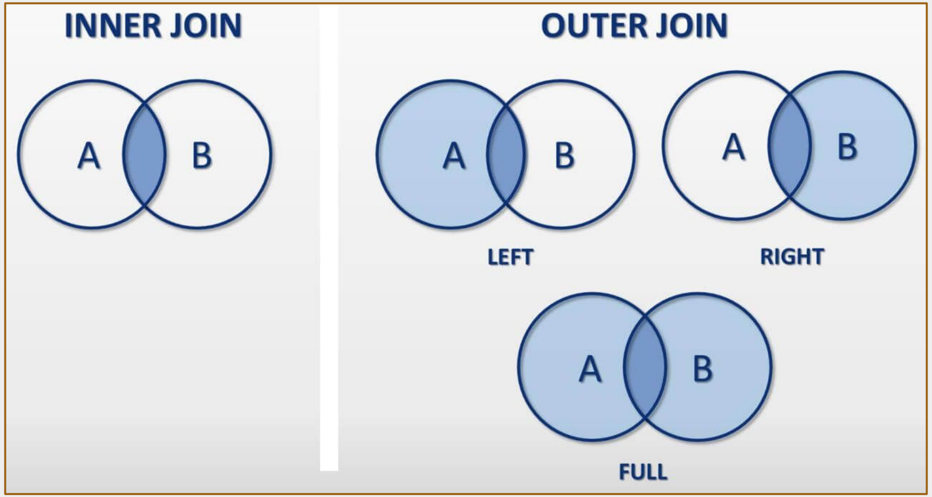
More options here: https://pandas.pydata.org/docs/user_guide/merging.html

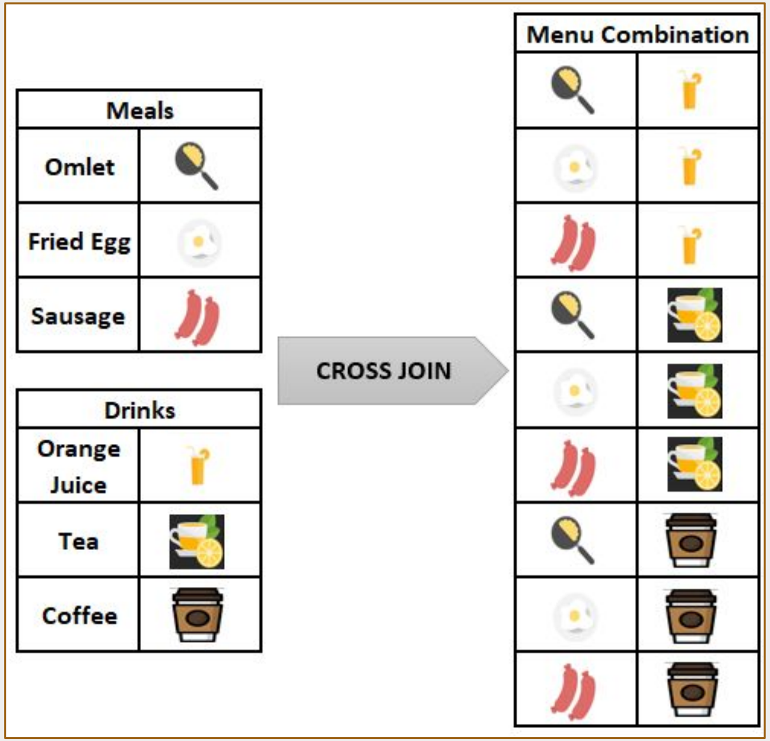
DataFrame (& Series) methods:

1. `df1.combine_first(df2)`: insert values from `df2` where `df1` is missing data
2. `df1.combine(df2, func)`: merge 2 DataFrames column-wise based on function `func`
3. `df1.join(df2)` wrapper for column-wise `.merge()`

SQL logic of pd.merge() & df1.join(df2)

In left join, `left` is preserved & missing values in intersection are filled from `right`. Obverse for right join. In `full`, intersection is expanded to fit values from `left` & `right`.





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 indices 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 indices 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`.

Code Along: Reorganizing Data

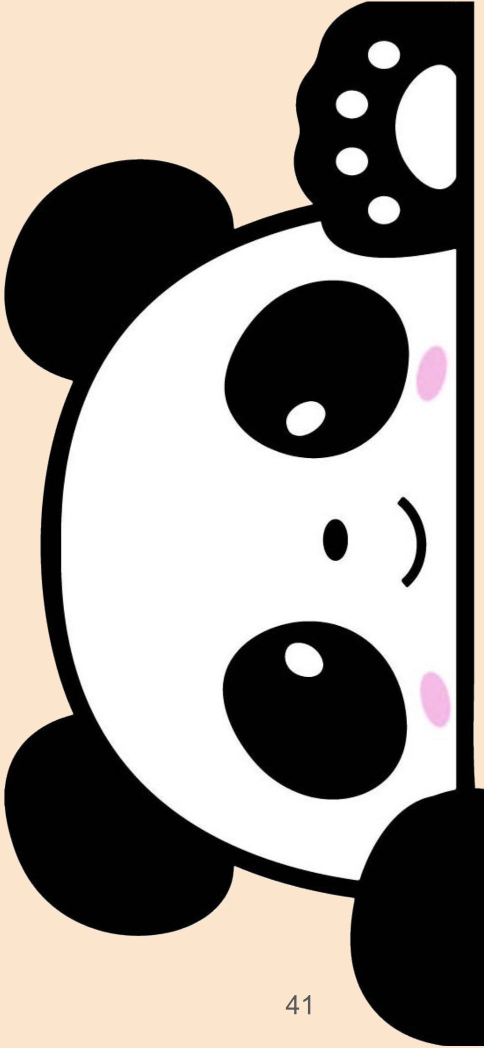
Let `df1 = pd.DataFrame(np.random.randint(0,high=9,size=(4,3)), columns=['B','a','C'], index=['i','j','h','k'])`, and

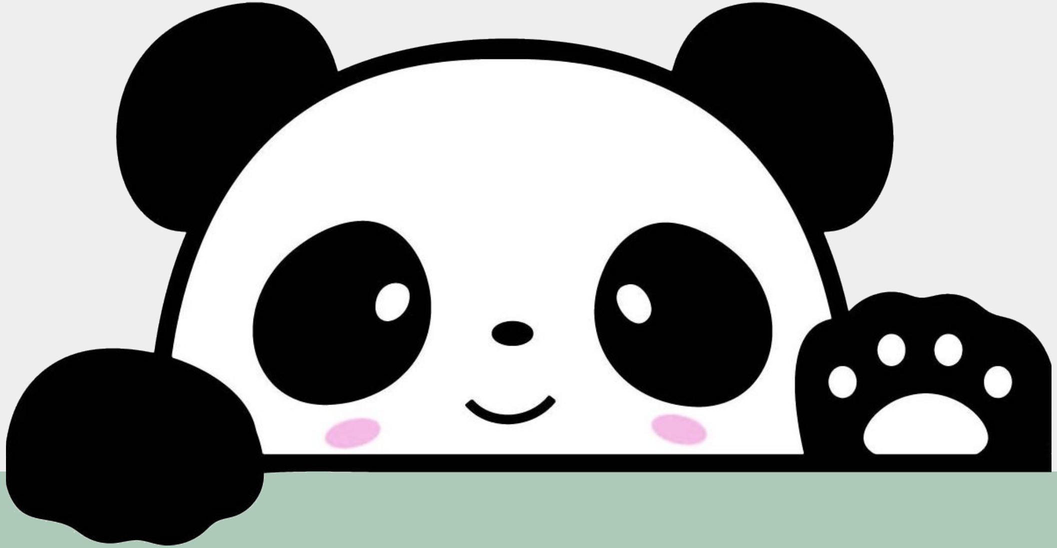
`df2 = pd.DataFrame(np.identity(3), columns=['B','C','D'], index=['k','l','m'])`

1. Sort `df1` by index, by index with `axis=1`, & by the values of column `'C'`.
2. Concatenate `df1` and `df2`. What happens where the indices aren't shared?

Merging is more complicated. You'll get to play with it during the exercises.

Exercise time!





GroupBy objects

Intro to GroupBy Objects

Grouping lets you organize (& sort) data hierarchically & run statistical analyses on different subsets of data simultaneously. Full API documentation at https://pandas.pydata.org/docs/user_guide/groupby.html#groupby

- **Syntax**: `grouped = df.groupby(['col1', 'col2', ...])` or `grouped = df.groupby(by='col')` (Take `df.T` to group by rows)
- Aggregate methods evaluate each group separately (same syntax as DataFrames)
- Can broadcast a different method/function to every group
- Groups are selected by category name with `.get_group(('cat',))` or `.get_group(('cat1', 'cat2', ...))` (single group selection shown in docs is deprecated: inner parentheses & comma required)

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

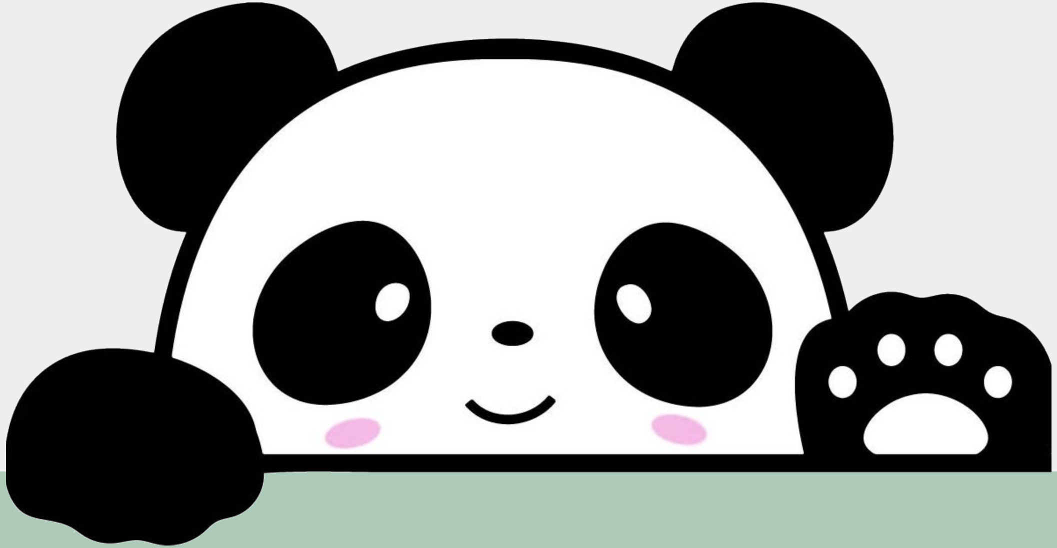
The .filter() method

Conditional selection is hard for GroupBy objects. `.filter()` method helps. See https://pandas.pydata.org/docs/reference/api/pandas.core.groupby.DataFrameGroupBy.filter.html#pandas.core.groupby.DataFrameGroupBy.filter

- Syntax: `GroupBy_obj.filter(func)`
- Only groups that collectively return `True` are included in output
- Typical use case: removing groups with too few members
- Group structure not preserved in output (so you can add results back to original DataFrame)

Code Along: GroupBy Objects

1. Group the exoplanets DataFrame by “`planet_type`” & assign the result to `grouped_pt`.
2. Print the mean “`mass_ME`” for each group in `grouped_pt`.
3. **Challenge***: call the filter method on `grouped_pt` to remove the `planet_type` group with 5 or fewer members. (*I don’t expect most people to get this in the time available)



Advanced Operations

Applying complex &/or user-defined functions

`.map()`

Apply 1 function that accepts & returns scalars element-wise

`.agg()`*

Apply ≥1 reducing (aggregating) functions (e.g `median()`)

`.apply()`

Apply ≥1 functions column- or row-wise to ≥1 Series in `df`

`.transform()`*

Apply ≥1 broadcasting functions

***Usable on GroupBy objects**

Need to chain functions that take & return whole Series/DataFrames/GroupBy objects? Use **`.pipe()`**. See also https://pandas.pydata.org/docs/user_guide/basics.html#function-application

Element-wise functions with .map()

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

- Function `func` may be passed by name or lambda function, but **both input & output must be scalars** (no arrays)
- **Note**: usually faster to apply vectorized functions if possible

```
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* takes functions with array input & scalar output (e.g. mean)
- Can pass >1 function by list of names, or dict with row/column names as keys & functions to apply as values

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

detection_method	mass_ME	radius_RE	orbital_radius_AU	orbital_period_yr
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 Series/DataFrame

- Passing >1 function works like in `.agg()`
- Transforming DataFrame of *x* columns by list of *y* functions yields *hierarchical* DataFrame with *x*×*y* columns
- **Do NOT 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')],
           )
```

When all else fails, use `.apply()`

- `.apply()` allows aggregating, broadcasting, & expanding* functions (*list-like output for every cell), but is **slower** than `.agg()` or `.transform()`
- Accepts GroupBy objects, but may fail to preserve structure &/or raise misleading error messages
- `.apply()` may be better (more intuitive) if you have a broadcasting function that varies by group
  - `.transform()` receives GroupBy objects in 2 parts—columns split into Series, & then groups as DataFrames—but `.apply()` only receives groups (like `.agg()`)

Windowing Operations

4 methods for evaluating other methods over moving/expanding windows, with similar API to GroupBy objects (most allow similar aggregating methods): https://pandas.pydata.org/docs/user_guide/window.html

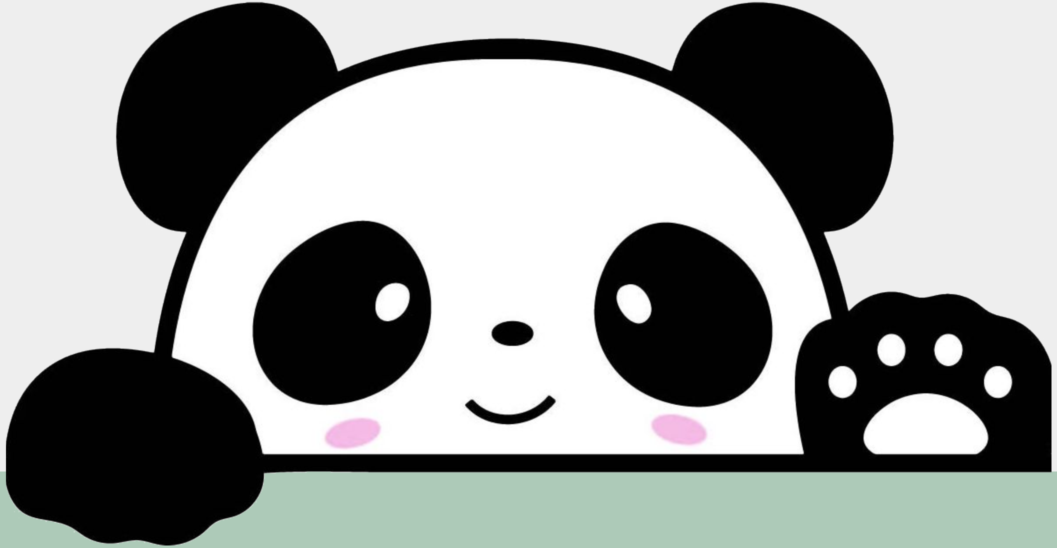
- All evaluate from current position back/up to window size or start of Series.
- All have `min_periods` kwarg to set minimum number of valid data points in window for consideration.
- All but `expanding()` can be used on GroupBy objects (applied per group).

Code-Along: Use the `.agg()` function

1. Call the `.agg()` function on `grouped_pt` (the exoplanets DataFrame grouped by `planet_type`) to calculate
    - the *mean* of `mass_ME`,
    - the *median* of `radius_RE`, &
    - the *0.5 quantile* (50th percentile) of `orbital_period_yr`

    in 1 line of code. Consult

https://pandas.pydata.org/docs/user_guide/basics.html#aggregating-with-multiple-functions (Hint: `.agg()` doesn't care if the input DataFrame is grouped)



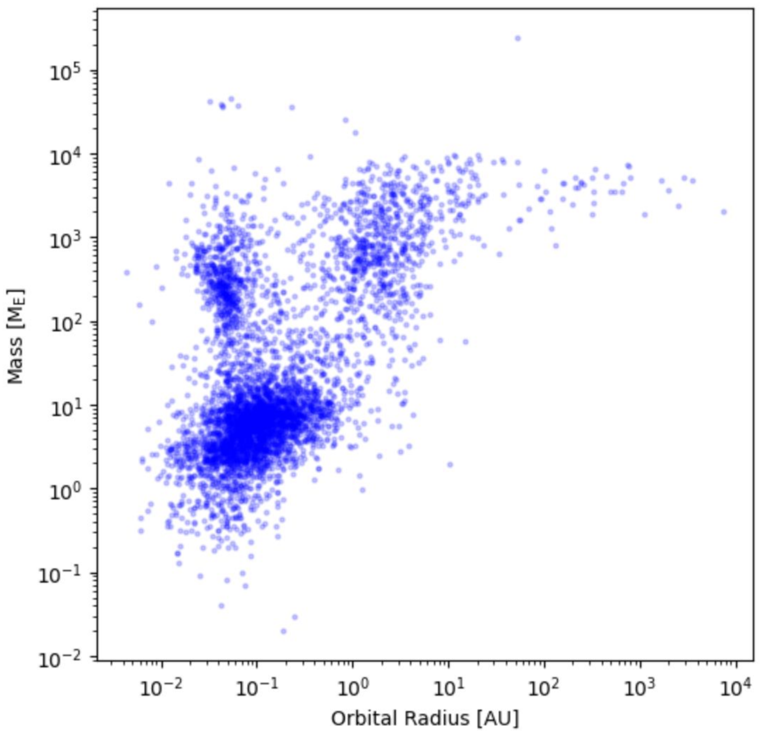
Built-in Plotting Methods

The .plot() wrapper method

- `.plot(kind='<kind>')` or `.plot.<kind>()` method lets you visualize Series/DataFrames/Groups without importing Matplotlib or converting to NumPy
- Default plot `kind` is `'line'`. Most other pairwise functions are also available. See https://pandas.pydata.org/docs/user_guide/visualization.html

```python
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

Limitations:

- Number of kwargs can quickly make code illegible
- 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'`

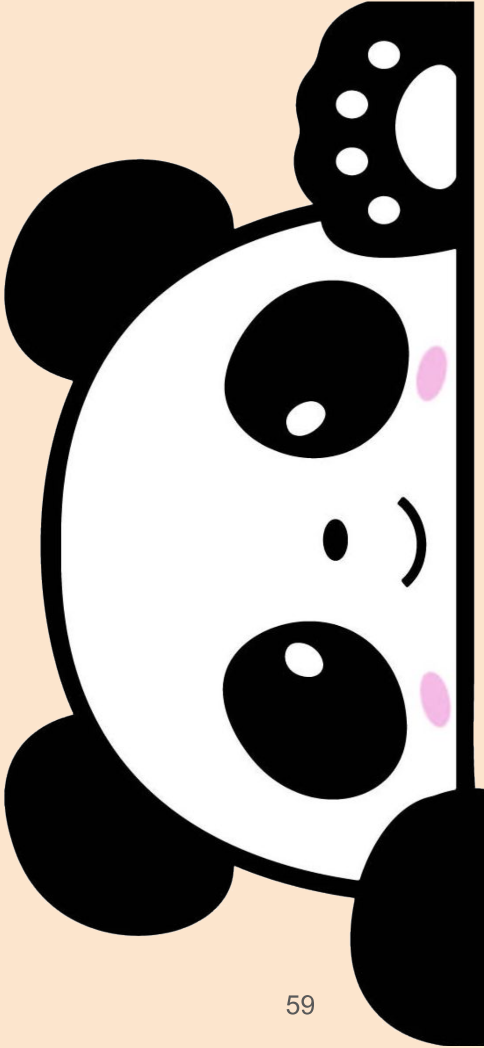
Most of what you can do with the `.plot()` wrapper, & much more, can be done better using Seaborn.

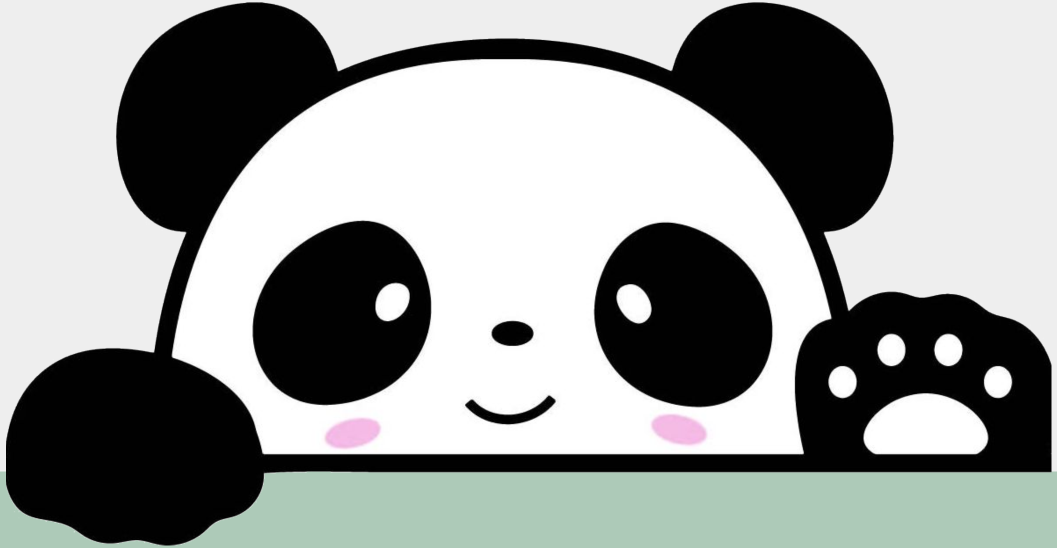
Code-Along: Plotting

1. Make a KDE plot of the `radius_RE` column of the exoplanets DataFrame. Set the x-axis limits to 0 and 30.

Consult https://pandas.pydata.org/docs/user_guide/visualization.html for how to choose a plot kind & pass in typical Matplotlib kwargs.

Exercise time!





Time Series

Scalar Class	Array Class	Pandas Data Type	Pandas Conversion/Creation Methods (results are Indices)
`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)` (need 2 of 3 kwargs)
`Timedelta`	`TimedeltaIndex`	`timedelta64[ns]`	**`.to_timedelta(`*`tdelts`*`)`** or **`.timedelta_range`**`(`*`start`*`=None, `*`end`*`=None, `*`periods`*`=None, `*`freq`*`=None)` (need 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)

More info at https://pandas.pydata.org/docs/user_guide/timeseries.html

Caveats about precision & date ranges

- Functions like `.date_range()` that take timestamps assume integer/float inputs are Unix times (time since 00:00:00 01-01-1970), in ns
- `Timedelta`-creating functions assume time resolution of weeks or better.
- Datetimes are natively stored in ns as 64-bit floats, limiting their range
  - Out-of-range datetimes must be converted to coarser units with `pd.Timestamp(<datetime>, unit='s')`, or `pd.Timestamp(np.datetime64(<datetime>), unit='s')` for BCE dates.
- `pd.to_datetime()` can't handle *Series* of out-of-range datetimes, so must use `.apply(lambda x: pd.Timestamp(x))`

```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

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

What's up with upsampling

Upsampling requires interpolation. Data at new timesteps will be NaN everywhere except where they are integer multiples of the old timesteps.

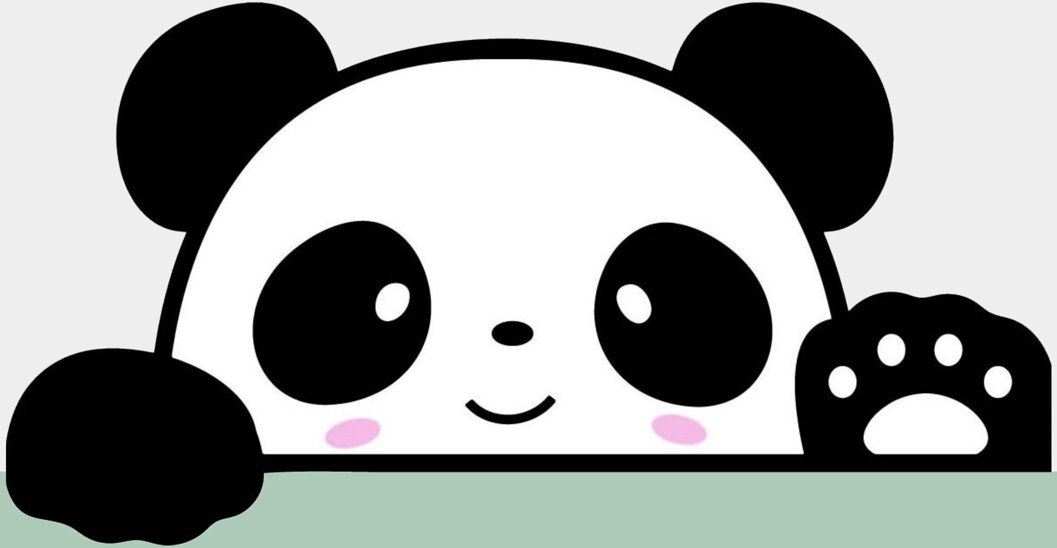
- `.resample('<unit>').asfreq()` leaves NaNs at intervening timesteps
- `.resample('<unit>').ffill(limit=None)` fills intervening timesteps (up to limit) with most recent non-NaN value
- `.resample('<unit>').interpolate(method='linear')` can fill intervening time steps with any SciPy interpolation method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

Code-along: Time Series

1. Initialize a datetime index with 10 timestamps, 1 second apart, ending on midnight 1/1/2025. Use the `pd.date_range()` function.

```
ts = pd.date_range(end='01/01/2025', freq='1s', periods=10)
print(ts)
```

```
DatetimeIndex(['2024-12-31 23:59:51', '2024-12-31 23:59:52',
               '2024-12-31 23:59:53', '2024-12-31 23:59:54',
               '2024-12-31 23:59:55', '2024-12-31 23:59:56',
               '2024-12-31 23:59:57', '2024-12-31 23:59:58',
               '2024-12-31 23:59:59', '2025-01-01 00:00:00'],
              dtype='datetime64[ns]', freq='S')
```



(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 e.g. `V_c1`, `V_c2`, … with `True` where V=c_n in that row, `False` otherwise.

Efficient storage with the Categorical type

`Categorical` type reduces memory usage by mapping unique values to integer codes & storing codes at column's location in memory with smallest bit size

- `ser = pd.Series(data, dtype='category')`—make `Categorical` Series
- `df['cat_col'] = df['cat_col'].astype('category')`—convert column to `Categorical`
- `pd.Categorical(data, categories=['your','cats','here'], ordered=False)` —Create raw `Categorical` data (e.g., as Index)
- Categorical variables only support string vectorized functions

Cuts & Intervals

Two functions to discretize (bin) numerical Series data (e.g. for age brackets, to allow Categorical conversion):

- `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)

More here: https://pandas.pydata.org/docs/user_guide/reshaping.html#cut

Code-Along: the Categorical data type

Print the memory usage of the exoplanets DataFrame with `deep=True`. Then convert the `planet_type` and `detection_method` columns to `'category'` type & print the memory usage again.

Each column should be <2% of its original size in memory.

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))`
- See https://pandas.pydata.org/docs/user_guide/sparse.html

.pivot() & Hierarchical DataFrames

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

- Result will be [Hierarchical DataFrame]
- Can also use `.pivot_table(*args, aggfunc=functions)` method if DataFrame 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

Notes on parallelization & (new) HPC features

- Built-in functions allow parallelization of column*-wise operations via `Numba`, with `engine='numba'` & `engine_kwargs={"parallel": True}` in kwargs.
  - When possible use same number of threads as columns. **Do *not* use more threads than there are columns.**
  - 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, & PyArrow ChunkedArrays

Code-along: Numba demo

Run `numba_demo.py` & compare the outputs of `timeit` for the serial & numba accelerated calculations.

If you are on a HPC cluster:

The submission script is `numba_demo.sh`. Edit the `-A` parameter (& `-p`, if applicable) so that it runs on your cluster. You will probably also need to load a `Numba` module, which may require changing your toolchain.

If you are on a personal computer:

Make sure you have `Numba` installed. Either run the code from the command line or copy it into your notebook to run there. If in Jupyter, use the 2 lines starting with `%timeit` instead of the lines below those.

Summary

Day 1

- What is Pandas? Why use it?
- Main object classes
- Basic input/output
- Inspection & cleaning
- Built-in Operations
- Rearranging data

Day 2

- GroupBy objects
- Complex &/or user-defined functions
- Built-in plotting methods
- Time series functionality
- Advanced topics (e.g. ML prep)

(Final) Exercise time!