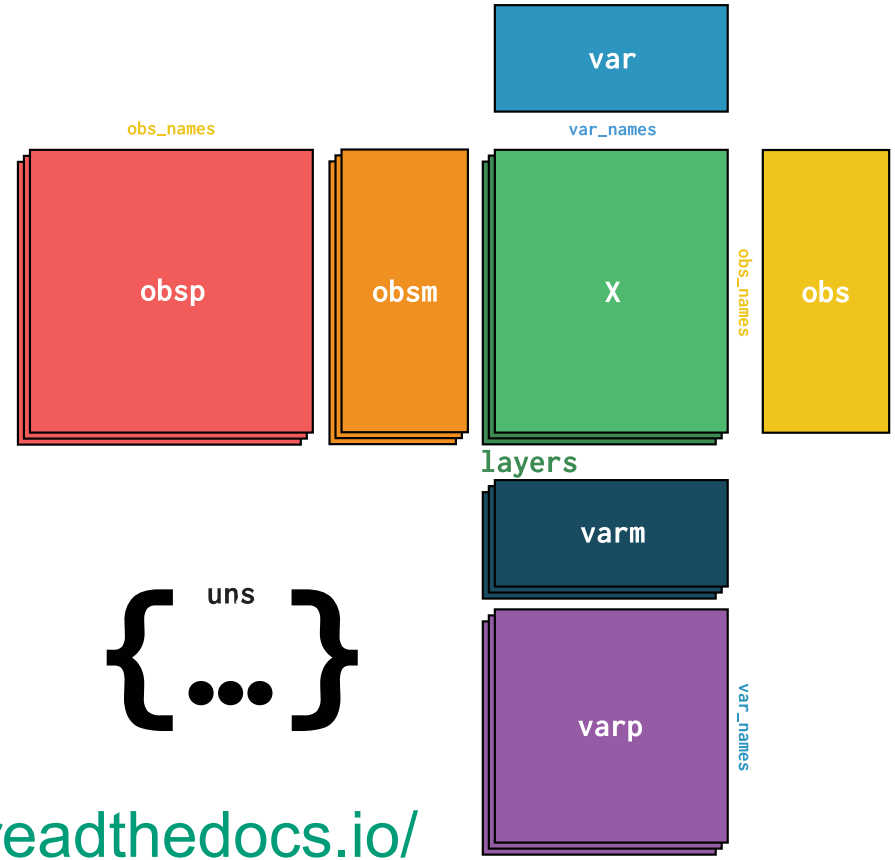
JOHNS HOPKINS
SCHOOL of MEDICINE

Module 2.1 - Essential Python Data Structures and Packages for Genomics: numpy & pandas

ME.440.825.FA23 9/11/23

The importance of standardized data structures in genomics

- Allow for consistent organization and development of ‘common’ frameworks
- Help to optimize/implement standard operations
- Promotes standard data transfer and storage formats
- Genomics datasets can be very large in size
  - Require specific storage and algorithmic considerations
- Organization of multidimensional data (e.g. DNA, RNA, protein, etc)
  - Integrating data across experiments



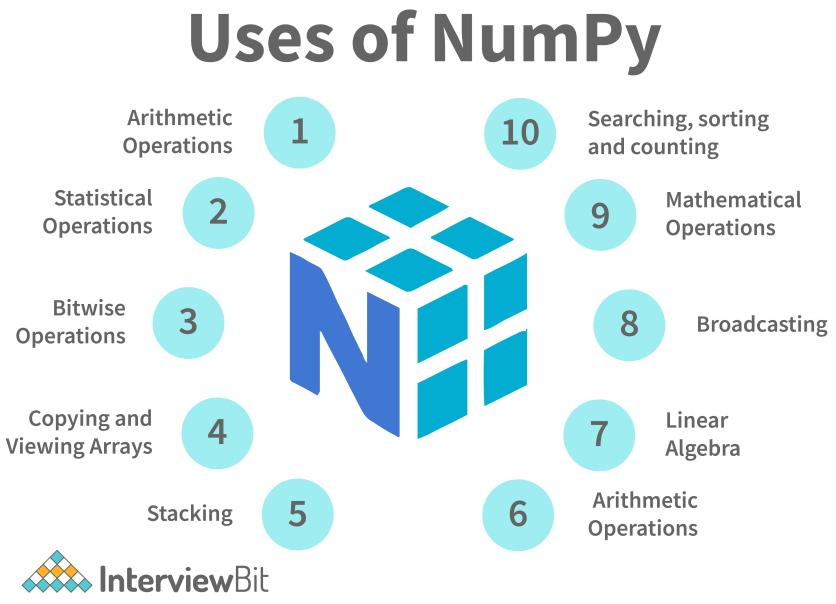
https://anndata.readthedocs.io/

Numerical python

NUMPY

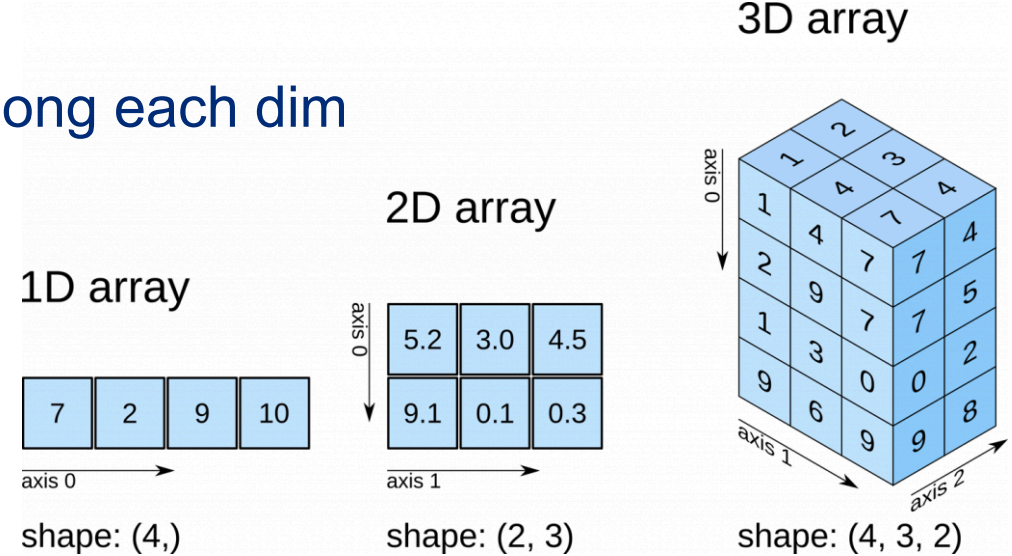
NUMerical PYthon

- What is `numpy`?
  - A fundamental package for scientific computing in python
  - Creates powerful tools for working with multi-dimensional arrays and matrices
  - Defines functions for mathematical and statistical operations on arrays
  - Efficient array operations - faster than python lists due to optimized storage and vectorized operations
- `numpy` for genomics
  - Data are often high-dimensional
    - e.g. Multi-sample, multi-gene, multi-channel
  - Efficient storage and manipulation are crucial
  - Basis framework for other foundational libraries (e.g. pandas, scikit-learn, AnnData)



Numpy arrays (ndarray)

- A grid of values, all of the same data type (dtype)
- Indexed by a tuple of non-negative integers
- *rank* = number of dimensions
- s*hape* = describes the size along each dim
- Examples?



Creating numpy arrays

```
numpy.array(object, dtype=None, *, copy=True, ...)
```

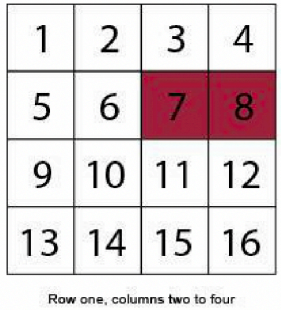
- Can be instantiated using python lists
  - `numpy.array([1,2,3]) #1D`
- Nested lists define multi-dimensional arrays
  - `numpy.array([[1,2,3],[4,5,6]) #2D`
- Special functions to create ‘placeholder’ arrays
  - `np.zeros(), np.ones()`
  - `np.arange() # similar to python ‘range()’`
  - `np.linspace() #`
-

Array attributes

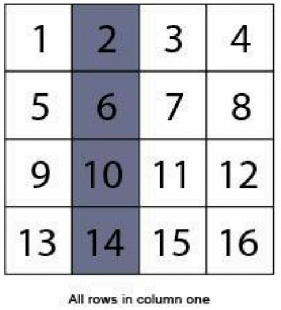
```
import numpy as np
arr = np.array([[4,18,7,12,5],[2,10,9,6,15]],dtype=np.float32)

arr.ndim # number of dimensions
  2
arr.shape # tuple of dimension lengths
  (2, 5)
arr.size # total number of elements in array
  10
arr.dtype # data type of the array
  dtype('float32')
```

Indexing & Slicing arrays

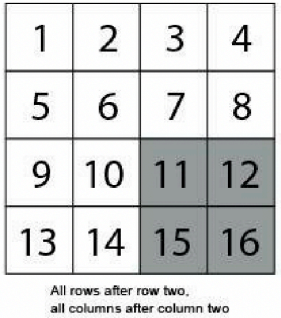
- Access individual elements with [] notation:
  - `arr[2] # for 1D array`
  - `arr[1,2] # for 2D array`
- Slice arrays using start:stop:step notation similar to lists
  - `arr[1:3]`
- Boolean indexing
  - Conditional selection using boolean operations, ideal for filtering data based on conditions.
  - `arr[arr > 5]`



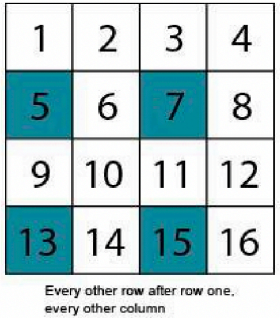

```
>>> arr[1, 2:4]
array([7, 8])
```




```
>>> arr[:, 1]
array([2,  6, 10, 14])
```




```
>>> arr[2:, 2:]
array([[11, 12],
       [15, 16]])
```




```
>>> arr[1::2, ::2]
array([[5, 7],
       [13, 15]])
```

Basic Arithmetic operations

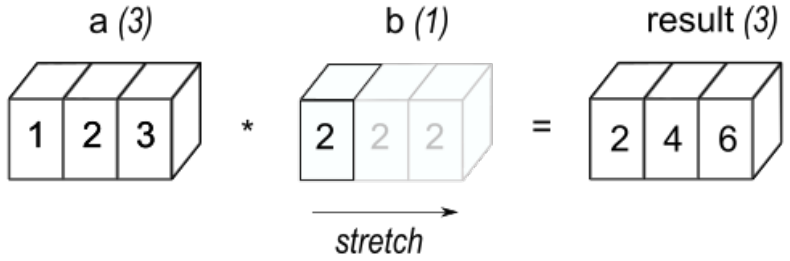
- np.add(a,b)
- np.subtract(a,b)
- np.multiply(a,b)
- np.divide(a,b)

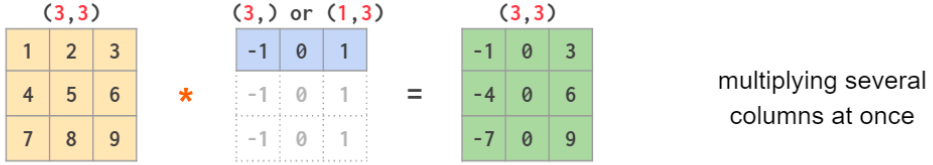
- Examples:

```
import numpy as np

a = np.array([1,2,3])
b = np.array([4,5,6])
a + b
   array([5, 7, 9])
a * b
   array([ 4, 10, 18])
a / b
   array([0.25, 0.4 , 0.5 ])
```

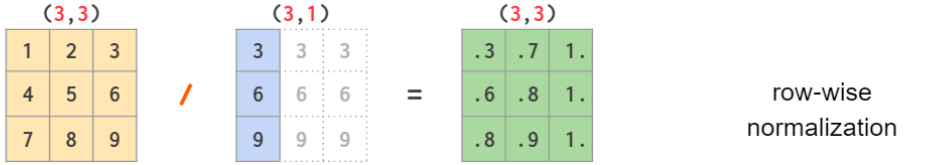
Broadcasting

- Provides a means of vectorizing array operations
- Two arrays do not need to have same shape (with some conditions)
- Simplest broadcasting example occurs when an array and a scalar value are combined (ie. `a * 2`)
- Two dimensions (between arrays) are 'broadcast' compatible when:
  - they are equal, ***or***
  - one of them is 1

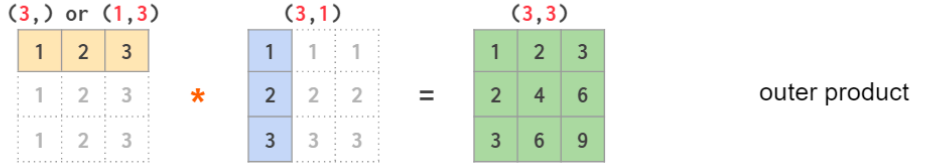




multiplying several columns at once



row-wise normalization



outer product

Mathematical Functions

- Common math functions available

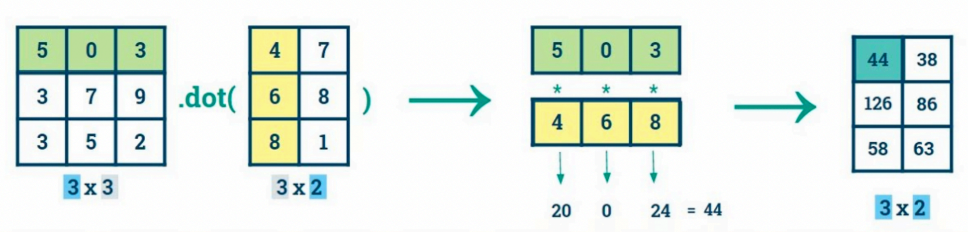
```
arr = np.array([[1,2,3,4],[5,6,7,8]])
arr
   array([[1, 2, 3, 4],
          [5, 6, 7, 8]])

arr.sum() # or np.sum(arr)
    36

arr.mean() # or np.mean(arr)
    4.5

arr.std() # or np.std(arr)
    2.29128...
```

- The `axis` argument allows for vectorized math across dimensions

```
arr.mean(axis=0) # operate across rows
    array([3., 4., 5., 6.])

arr.mean(axis=1) # operate across columns
    array([2.5, 6.5])

arr.sum(axis=0)
    array([ 6,  8, 10, 12])
```

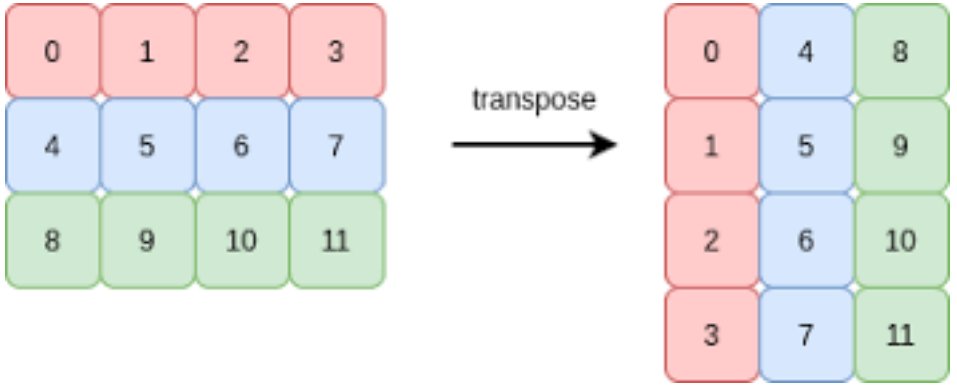
Matrix operations

`np.dot(a,b)`



`np.cross(a,b)`

`np.transpose(a)`



Numpy use in genomics

- Primarily behind the scenes, providing a numerical framework for many common data structures and operations in genomics
- Knowing `numpy` means you will feel more comfortable navigating and using more complex data structures in genomics
- Often, `ndarray` operations are used in analysis workflows to normalize or scale gene expression matrices
- Mathematical functions help in summarizing gene datasets, understanding variability, or identifying outliers.
- ***Matrix operations play a crucial role in certain algorithms, like Principal Component Analysis (PCA), often used in genomics.***

Resources

- numpy reference
  - https://numpy.org/doc/stable/reference/
- numpy cheat sheet
  - https://assets.datacamp.com/blog_assets/Numpy_Python_Cheat_Sheet.pdf
- numpy welcome tutorial
  - https://numpy.org/doc/stable/user/absolute_beginners.html

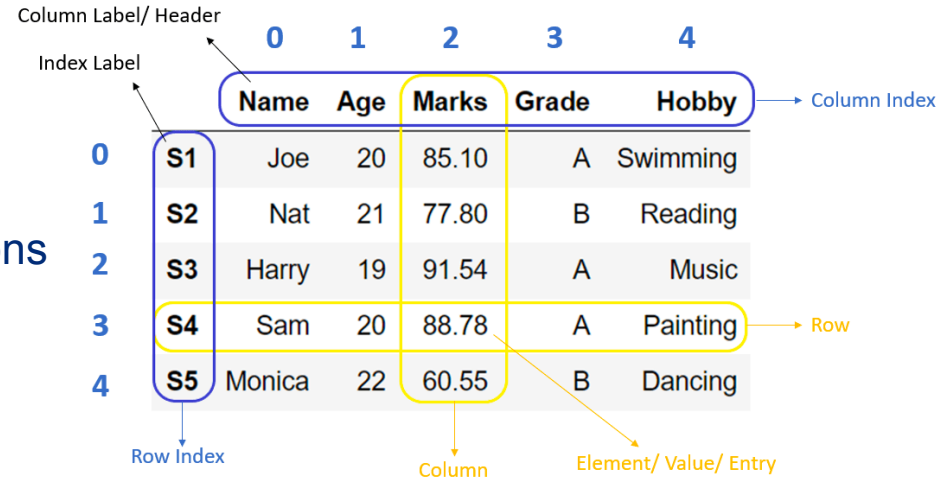
JOHNS HOPKINS
SCHOOL of MEDICINE

Python Data Analysis Library

PANDAS

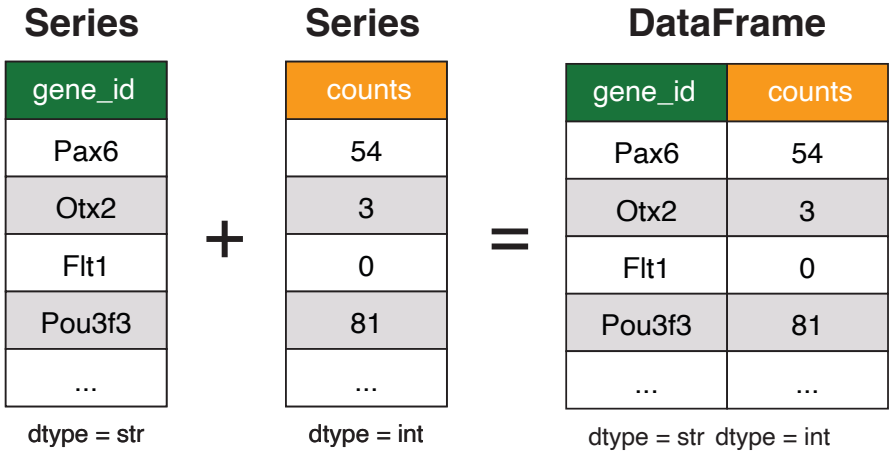
What is pandas?

- Open-source Python library providing high-performance, easy-to-use data structures and analysis tools
- Ideal for ‘structured data’ operations and manipulations
- Built on top of `numpy`
- Built-in plotting functions for quick visualizations
- Easily handles large datasets
  - Genomic datasets can be massive, and `pandas` allows for efficient querying and subsetting.
- Integrated handling of missing data
  - Genomic data often have gaps or missing values.



Pandas Data Structures

- Series (1D)
  - A vector of data points of the same dtype
  - Think ‘column’
- DataFrame (2D)
  - a multi-dimensional table made up of a collection of Series.
  - Columns (Series) can be different dtypes



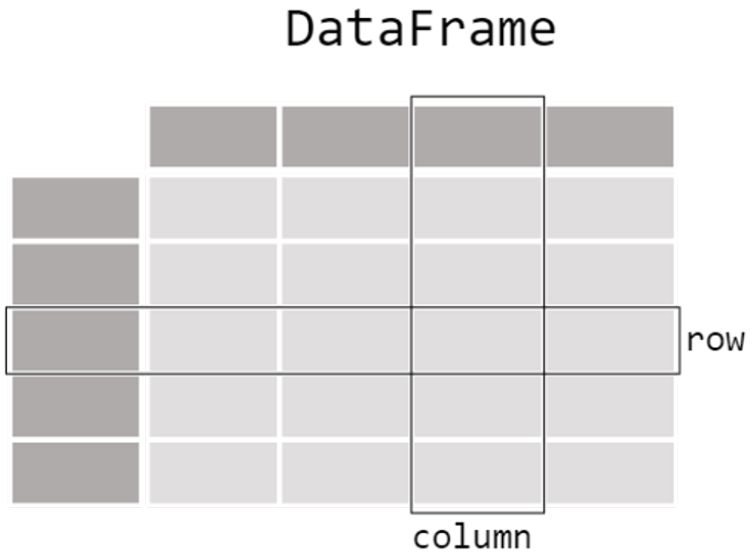
pandas.DataFrame

`pandas.DataFrame(data, index , columns , dtype , copy )`

- **data:** data takes various forms like *ndarray*, *series*, *map*, *lists*, *dict*, constants and also another *DataFrame*.
- **index:** For the **row labels**, that are to be used for the resulting frame,
- **columns:** For **column labels**, the optional default syntax is - *np.arrange(n)*. This is only true if no index is passed.
- **dtype:** Data type of each column.
- **copy:** If *True*, then a copy of the data is created instead of a reference. The default is *False*.

A pandas.DataFrame can be created using various inputs like:

- Lists, Dicts, Series, ndarrays, DataFrame



Creating a DataFrame from mixed data types

- Using a python `dict` as input

```
import pandas as pd

df = pd.DataFrame({'gene_name': ['gene1', 'gene2', 'gene3',
'gene4'],
                   'gene_id': ['id1', 'id2', 'id3', 'id4'],
                   'chromosome': ['chr1', 'chr2', 'chr3', 'chr4'],
                   'start': [100, 200, 300, 400],
                   'end': [200, 300, 400, 500]})
```

	gene_name	gene_id	chromosome	start	end
0	gene1	id1	chr1	100	200
1	gene2	id2	chr2	200	300
2	gene3	id3	chr3	300	400
3	gene4	id4	chr4	400	500

Reading data from a structured file into pandas

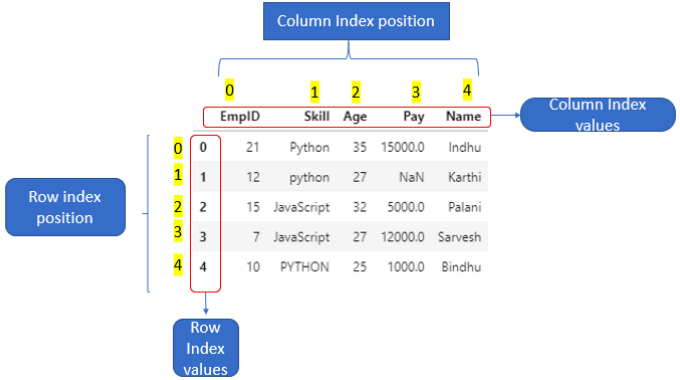
```
pd.read_csv(filepath_or_buffer, sep=' ,' , header='infer',  index_col=None,
usecols=None, skiprows=None)
```

- `filepath_or_buffer`: Location of the structured file. It accepts any string path or URL of the file.
- `sep`: It stands for separator, default is ‘, ‘.
- `header`: It accepts `int`, a `list` of `int`, row numbers to use as the column names, and the start of the data.
    - If no names are passed, i.e., `header=None`, then, it will display the first column as 0, the second as 1, and so on.
- `usecols`: Retrieves only selected columns from the CSV file.
- `index_col`: If None, there are no index numbers displayed along with records.
- `skiprows`: Skips passed rows in the new data frame.

- `pd.read_csv()` can be used to directly read a structured data file (or file handle) into a`pandas DataFrame` object
- When importing data, you can choose a row index column
- The `sep` argument can be used to define a custom separator. e.g. ‘tab-separated’ (‘\t’)
    - `df = pd.read_csv('data_file.tsv',sep='\t')`
- Other relevant import functions include:
    - pd.read_excel()
    - pd.read_hdf()
    - pd.read_json()
    - pd.read_html() # HTML tables

DataFrame Indexing

- Pandas uses *indexing* to optimize the speed and efficiency of operations on `DataFrames`
  - including accessing data, performing arithmetic operations, and merging data.
- Each row and column has a unique ‘index’
  - Default: range starting from 0
- You can choose a custom index on DataFrame creation or using the `.set_index()` method
  - Returns a ***new*** `DataFrame` by default (does not operate ***inplace***)
  - Use `inplace=True` to modify df directly or re-assign `df`
  - Reset to default using `.reset_index()`



	gene_name	gene_id	chromosome	start	end
0	gene1	id1	chr1	100	200
1	gene2	id2	chr2	200	300
2	gene3	id3	chr3	300	400
3	gene4	id4	chr4	400	500

```
df.set_index('gene_name')
```

gene_name	gene_id	chromosome	start	end
gene1	id1	chr1	100	200
gene2	id2	chr2	200	300
gene3	id3	chr3	300	400
gene4	id4	chr4	400	500

DataFrame Column Indexing

gene_name	gene_id	chromosome	start	end
gene1	id1	chr1	100	200
gene2	id2	chr2	200	300
gene3	id3	chr3	300	400
gene4	id4	chr4	400	500

- Columns are referenced using their key values in a syntax similar to `dict`
  - `df['gene_id']`
  - Selecting a single column returns a `Series` object
  - Can also use 'dot' notation for columns
    - `df.gene_id`

```
gene_name
gene1    id1
gene2    id2
gene3    id3
gene4    id4
Name: gene_id, dtype: object
```

- Passing a `list` of column names returns only the subset of columns selected
  - `df[['gene_id','chromosome']]`

gene_name	gene_id	chromosome
gene1	id1	chr1
gene2	id2	chr2
gene3	id3	chr3
gene4	id4	chr4

DataFrame Row selection

- You can access rows of a DataFrame in three different ways:
  - Index label-based (`.loc[]`):
    - `df.loc['gene3']`
  - Position-based (`.iloc[]`):
    - `df.iloc[1]`
    - Can use standard `[start:stop:skip]` slicing here
      - `df.iloc[::2]`
  - Conditional indexing:
    - `df[df['gene_id'] == 'id3']`

gene_name	gene_id	chromosome	start	end
gene1	id1	chr1	100	200
gene2	id2	chr2	200	300
gene3	id3	chr3	300	400
gene4	id4	chr4	400	500

```
gene_id         id3
chromosome     chr3
start           300
end             400
Name: gene3, dtype: object
```

gene_name	gene_id	chromosome	start	end
gene3	id3	chr3	300	400

Viewing & Summarizing

- `.head()` and `.tail()` can be used to preview a DataFrame
- .shape() works similarly as in numpy
- `.info()` provides information about the structure of the DataFrame
- `.describe()` provides summary statistics for numerical columns

```
df.info()
    <class 'pandas.core.frame.DataFrame'>
    Index: 4 entries, gene1 to gene4
    Data columns (total 4 columns):
    #   Column      Non-Null Count  Dtype
    ---  ------      --------------  -----
    0   gene_id     4 non-null      object
    1   chromosome  4 non-null      object
    2   start       4 non-null      int64
    3   end         4 non-null      int64
    dtypes: int64(2), object(2)
    memory usage: 320.0+ bytes
```

	start	end
count	4.000000	4.000000
mean	250.000000	350.000000
std	129.099445	129.099445
min	100.000000	200.000000
25%	175.000000	275.000000
50%	250.000000	350.000000
75%	325.000000	425.000000
max	400.000000	500.000000

Descriptive & Summary Statistics

```
df['start'].max()
    400

df['start'].cumsum()
    gene_name
    gene1     100
    gene2     300
    gene3     600
    gene4    1000
    Name: start, dtype: int64

df['start'].argmax()
    3

df['start'].corr(df['end'])
    1.0
```

Method	Description
count	Number of non-NA values
describe	Compute set of summary statistics for Series or each DataFrame column
min, max	Compute minimum and maximum values
argmin, argmax	Compute index locations (integers) at which minimum or maximum value obtained, respectively
idxmin, idxmax	Compute index values at which minimum or maximum value obtained, respectively
quantile	Compute sample quantile ranging from 0 to 1
sum	Sum of values
mean	Mean of values
median	Arithmetic median (50% quantile) of values
mad	Mean absolute deviation from mean value
var	Sample variance of values
std	Sample standard deviation of values
skew	Sample skewness (3rd moment) of values
kurt	Sample kurtosis (4th moment) of values
cumsum	Cumulative sum of values
cummin, cummax	Cumulative minimum or maximum of values, respectively
cumprod	Cumulative product of values
diff	Compute 1st arithmetic difference (useful for time series)
pct_change	Compute percent changes

Pandas ‘category’ type

```
df['chromosome'] = df['chromosome'].astype('category')
```

```
df.dtypes
```

```
gene_id          object
chromosome     category
start             int64
end               int64
dtype: object
```

- Equivalent to ‘factors’ in R.
- Used to improve efficiency and workflow for categorical values in `DataFrames`
- Useful to keep in mind for some downstream applications including aggregation and visualization

Data Wrangling with pandas

```
exp = pd.DataFrame(
    {'gene': ['gene1', 'gene2', 'gene3']*3,
    'celltype': ['neuron']*3 + ['astrocyte']*3 + ['oligodendrocyte']*3,
    'counts': np.random.poisson(lam=5,size=9)}
)
```

	gene	celltype	counts
0	gene1	neuron	2
1	gene2	neuron	3
2	gene3	neuron	3
3	gene1	astrocyte	2
4	gene2	astrocyte	1
5	gene3	astrocyte	1
6	gene1	oligodendrocyte	1
7	gene2	oligodendrocyte	0
8	gene3	oligodendrocyte	3

- Filtering
  - Using conditional indexing, rows can be filtered based on selected criteria

- Grouping
  - Using the `.groupby()` method, you can perform operations and summarize groups of records that match certain criteria (ie. Group by chromosome and count number of genes per chromosome)

```
exp.groupby('gene')['counts'].mean()
```

gene	counts
gene1	1.666667
gene2	1.333333
gene3	2.333333

```
exp.groupby('celltype')['counts'].mean()
```

celltype	counts
astrocyte	1.333333
neuron	2.666667
oligodendrocyte	1.333333

‘Long’ vs ‘wide’ data formats

Long

	gene	celltype	counts
0	gene1	neuron	2
1	gene2	neuron	3
2	gene3	neuron	3
3	gene1	astrocyte	2
4	gene2	astrocyte	1
5	gene3	astrocyte	1
6	gene1	oligodendrocyte	1
7	gene2	oligodendrocyte	0
8	gene3	oligodendrocyte	3

- Considered ‘tidy’ representation of data
- Required layout for many visualization tasks (e.g. `plotnine` and GoG)
- Columns are variables/features
- Each row is a unique observation

Wide

celltype	astrocyte	neuron	oligodendrocyte
gene			
gene1	2	2	1
gene2	1	3	0
gene3	1	3	3

- ‘Condensed’ representation for larger datasets
- Easier to visualize relationships across conditions
- Common format for genomics datasets (features X samples)
-

Transforming data in pandas

- Need for reshaping
  - Make data more readable or understandable
  - Prepare data for specific types of analysis
  - Aggregating and summarizing complex datasets

- ***Pivot*** functions convert *long -> wide* format
  - `.pivot()`
    - 3 main arguments: `index`, `columns`, and `values`
    - Ideal for simple rearrangement w/o aggregation
    - Throws an error with duplicate entries
  - `.pivot_table()`
    - More flexible than `.pivot()`
    - Adds an `aggfun` argument that takes a function (e.g. `sum`, `mean`, `sd`, etc) to aggregate data

- ***Melt*** functions reshape *wide -> long* format
  - `.melt()`
    - id_vars argument specifies 'non-value' columns (e.g. labels)
    - `ignore_index=False` to keep index values in melted df

```
exp.pivot(index='gene', columns='celltype', values='counts')
```

	gene	celltype	counts
0	gene1	neuron	5
1	gene2	neuron	0
2	gene3	neuron	7
3	gene1	astrocyte	5
4	gene2	astrocyte	1
5	gene3	astrocyte	8
6	gene1	oligodendrocyte	9
7	gene2	oligodendrocyte	3
8	gene3	oligodendrocyte	6

celltype	astrocyte	neuron	oligodendrocyte
gene			
gene1	5	5	9
gene2	1	0	3
gene3	8	7	6

```
exp_wide.melt(ignore_index=False)
```

celltype	astrocyte	neuron	oligodendrocyte
gene			
gene1	5	5	9
gene2	1	0	3
gene3	8	7	6

	gene	celltype	counts
0	gene1	neuron	5
1	gene2	neuron	0
2	gene3	neuron	7
3	gene1	astrocyte	5
4	gene2	astrocyte	1
5	gene3	astrocyte	8
6	gene1	oligodendrocyte	9
7	gene2	oligodendrocyte	3
8	gene3	oligodendrocyte	6

Resources

- For a great, fast introduction to pandas try the official tutorial ’10 minutes to pandas’
  - https://pandas.pydata.org/docs/user_guide/10min.html
- pandas documentation
  - https://pandas.pydata.org/pandas-docs/stable/index.html
- pandas: Input/output
  - https://pandas.pydata.org/pandas-docs/stable/reference/io.html
- pandas: DataFrame
  - https://pandas.pydata.org/pandas-docs/stable/reference/frame.html

Side Note

- Pandas is not the ***only*** python library for data analysis, but it is one of the most popular
- Recent packages (including one from the creator of pandas, polars) are attempting to fix issues related to scalability with pandas
- For now, pandas remains a useful framework to learn DataFrames and how to operate/index/search/manipulate them.