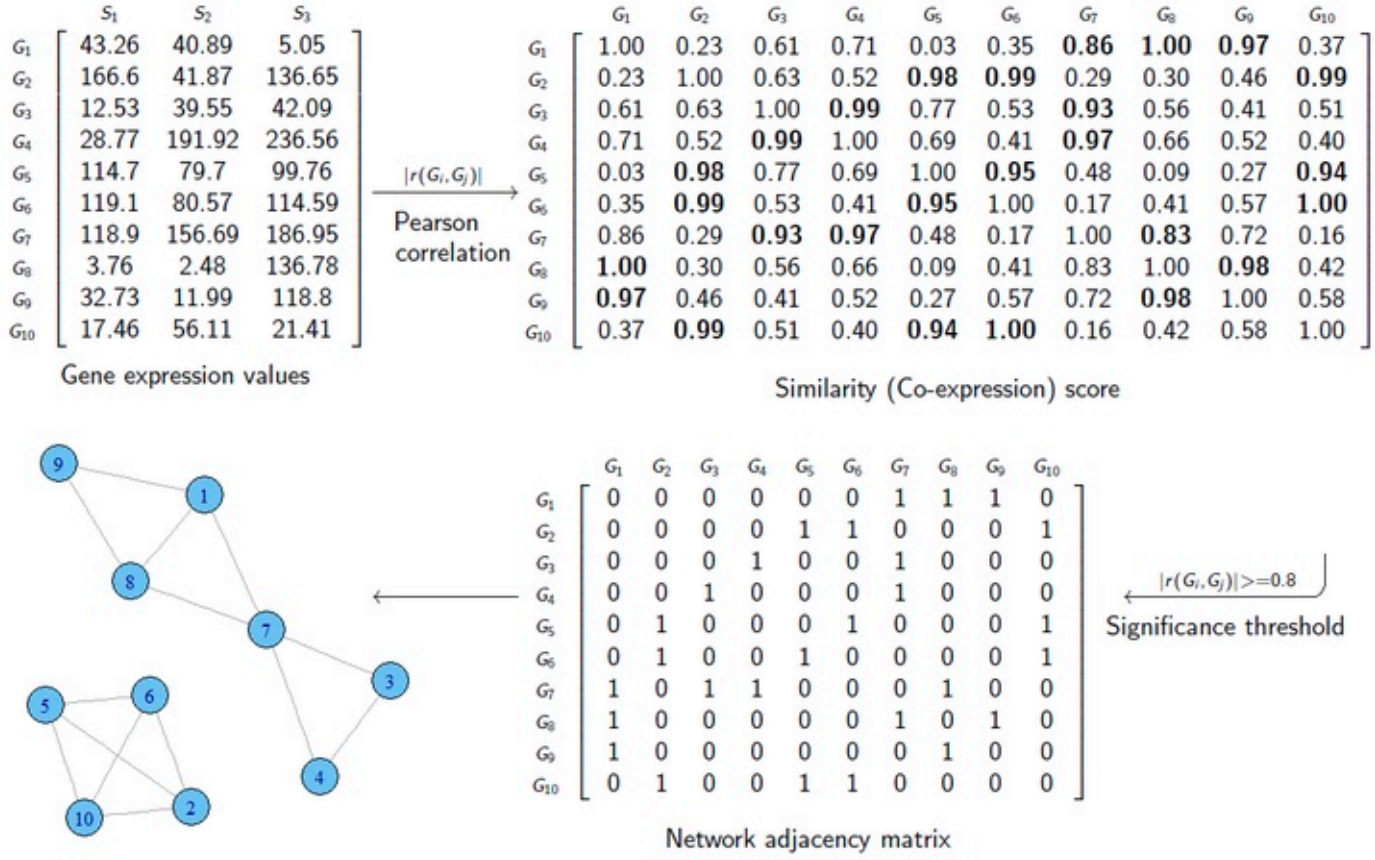
Pandas

What is Pandas?

Python Data Analysis Library

A fully-featured code library for manipulating data arranged in tables (i.e. matricies or data frames). It's arguably the most popular Python library used for data engineering. For those of you who have used the popular statistical language, R, Pandas brings many of that languages capabilities to python.



Why familiarize yourself with Pandas?

So far we've discussed how you build your own multidimensional objects like lists of lists and dictionaries of dictionaries from raw data. However, bioinformatics modules (and many others) will often **return** results in the form of Pandas **data frame** or **a matrix**. Further manipulation of these results (e.g. filtering, statistical analysis, data reorganization) will require some knowledge of Pandas operations.

For example, lets say you want to parse your RNA-seq results to a list of genes within a specific range of p-values and log fold changes, e.g., all p-values < 1e-15 and log fold changes > 1.2. You can apply your knowledge of Python operators such as `and, >, <` to subset a data frame based on the afformentioned parameters.

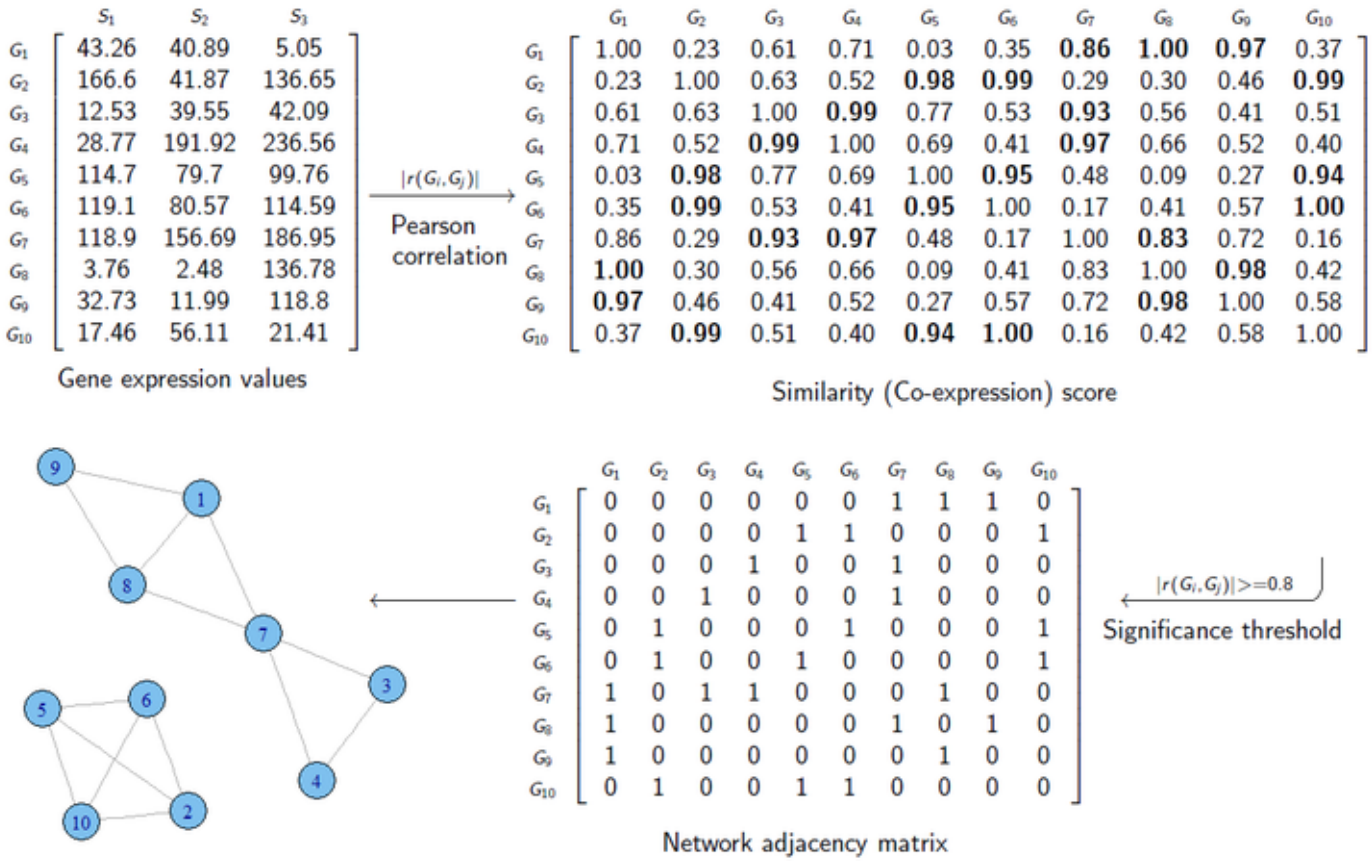
Pandas has the ability to read in various data formats

- Open a local file using Pandas, usually a comma-separated values (CSV) file, but could also be a tab-delimited text file (TSV), Excel, json, etc
- Read a remote file on a website through a URL or read data from a remote database.

Types of data manipulated in Pandas

Matrices

A matrix is an data structure where numbers are arranged into rows and columns. They will typically conatin floats **or** integers, but not both. Matrices are used when you need to perform mathmatical operations between datasets that contain multiple dimensions (i.e. measurements for two or more variables that change at the same time).



Data frames

A data frame is a table-like data structure and can contain different data types (strings, floats, integers, etc.) in different columns. This is the type of data structure you're used seeing in Excel. Each column should only contain one data type.

Genes	Chr	A/B (SN)	C/D (DA)	References	GO term Process
AGTR1	chr3	**0.46**	**0.34**	[13, 29, 30]	signal transduction (GO:0007165)
ALDH1A1	chr9	**0.41**	**0.21**	[12, 13, 29, 30]	cellular aldehyde metabolic process (GO:0006081)
ANK1	chr8	**0.43**	0.71	[13, 19, 29, 30]	cytoskeleton organization (GO:0007010)
ATP5J	chr21	0.93	**0.48**	[16, 19]	mitochondrial proton transport (GO:0042776)
ATP5L	chr11	0.99	**0.59**	[16, 19]	mitochondrial proton transport (GO:0042776)
ATP6V1D	chr14	0,89	**0.57**	[19, 30]	proton transport (GO:0015992)
BEX1	chrX	0.70	**0.41**	[14, 19, 29, 30]	up regulation of transcription factor (GO:0045944)
CBLN1	chr16	**0.52**	0.70	[13, 29, 30]	synaptic transmission (GO:0007268)
COX6C	chr8	1.02	**0.51**	[16, 19]	metabolic energy generation (GO:0006091)
DNM1	chr9	0.73	**0.60**	[16, 19]	endocytosis (GO:0006897)
DYNC1I1	chr7	0.68	**0.53**	[16, 19]	vesicle transport along microtubule (GO:0047496)
FGF13	chrX	**0.40**	0.69	[14, 19, 30]	MAPK cascade (GO:0000165)
GABRB1	chr4	**0.52**	0.72	[16, 29]	signal transduction (GO:0007165)
HSPB1	chr7	**1.63**	2.08	[14, 30]	intracellular signal transduction (GO:0035556)
JMJD6	chr17	**1.63**	1.22	[14, 30]	histone demethylation (GO:0016577)
MKNK2	chr19	**1.5**	1.15	[14, 30]	regulation of translation (GO:0006417)
NDUFB2	chr7	0.88	**0.44**	[16, 19]	complex I (NADH to ubiquinone) (GO:0006120)
NPTX2	chr7	**2.13**	1.42	[15, 29]	synaptic transmission (GO:0007268)
RGS4	chr1	**0.46**	**0.54**	[14, 30]	signal transduction (GO:0007165)
SV2B	chr15	**0.45**	0.82	[14, 16, 30]	neurotransmitter transport (GO:0006836)
SYT1	chr12	0.54	**0.58**	[14, 16, 19, 30]	synaptic transmission (GO:0007268)
TF	chr3	**1.33**	0.80	[14, 15, 30]	iron ion homeostasis (GO:0055072)
TUBD1	chr17	**1.29**	1.45	[15, 16]	microtubule-based process (GO:0007017)
UQCRC2	chr16	0.66	**0.55**	[12, 16, 19]	aerobic respiration (GO:0009060)
ZBTB16	chr11	**1.45**	1.63	[29, 30]	transcription, DNA-templated (GO:0006351)

The known genes confirmed in at least two independent single studies are reported (see references indicated). **Chr**: chromosome; **A/B (SN)** and **C/D (DA)**: expression ratio of value A/value B (SN ONLY) and value C/value D (DA ONLY) resulted from TRAM analysis (see respectively, S2 and S4 Tables). In bold: expression ratio values statistically significative in single gene level TRAM analysis, q value<0.05 (see respectively, S3 and S5 Tables); **GO term Process**: description and accession number of the main biological process associated to the gene according to Gene Ontology Consortium.

doi:10.1371/journal.pone.0161567.t005

A brief word on vectorization

Operations in Pandas, like R, work most efficiently when vectorized

You can think of a vector (also referred to as an array) as a type of list that contains a single data type and optimized for parallel computing. For matricies and data frames in Pandas (also NumPy), vectors are rows and columns.

Rather that looping through individual values (scalars), we apply operations to vectors (rows/columns). That is, the vector is treated as a single object. This topic can get a bit complicated, but it is worth doing your homework if you frequently work with these data types. Here's a few articles to get you started:

- A beginners guide to optimizing pandas code for speed.
- Why is vectorization faster in general than loops?
- Python Lists vs. Numpy Arrays, what's the difference?

Methodology	Average single run time	Marginal performance improvement
Crude looping	645 ms	
Looping with iterrows()	166 ms	3.9x
Looping with apply()	90.6 ms	1.8x
Vectorization with Pandas series	1.62 ms	55.9x
Vectorization with NumPy arrays	0.37 ms	4.4x

Vectorization with Pandas series is **~390x** faster than crude looping

Pandas documentation

Each function (method) in Pandas has many options and might not work the way you expect. It's definitely worth reading the documentation. Functions and options are sometimes updated, so even if you are already familiar with a function, it's a good idea to have a quick look.

Documentation is here https://pandas.pydata.org/docs/

Read getting started first https://pandas.pydata.org/docs/getting_started/index.html#getting-started

(what's possible: data types, summary stats, plots, table layouts, merging)

Pandas user guide (how it works, details on how Pandas thinks about data types) https://pandas.pydata.org/docs/user_guide/index.html#user-guide

Specific information about all the methods and classes https://pandas.pydata.org/docs/reference/index.html#api

But you'll probably want to start with a google search like `pd load dataframe` or `pandas read excel skip rows`

Basic methods for data manipulation

Reading in files

"Slicing" refers to subsetting, or extracting rows and columns from a data frame. Here we'll read in a data frame, look at the contents, and subset it by slicing out arbitrary regions.

```
import pandas as pd

# Setting index_col to 0 tells us that the first column contains the row names
cell_attributes = pd.read_csv("./meta_data.csv", index_col = 0)

type(cell_attributes)
# prints <class 'pandas.core.frame.DataFrame'>
```

Note: We can read/write data in many other formats like tab delimited text `.tsv` and excel spreadsheets `.xlsx`. Please refer to this document for a full description of Pandas I/O tools.

Pandas tries to convert input strings to the appropriate data `dtype` (float, int etc). This is generally very helpful.

```
# We notice rows and columns are truncated with the dimensions printed at the bottom
print(cell_attributes)

# Change the output view options
pd.set_option('display.max_rows', 100)
pd.set_option('display.max_columns', 100)

# Does this function seem familiar?
cell_attributes.head(10)
```

Slicing

Pandas has different methods for subsetting dataframes.
We'll dicuss the most common methods, **loc**, and **iloc**

loc allows us to subset data by row or column label. For example, if I would like to subset the column 'n_counts', I would use the following command:

```
# The comma separates rows and columns, and the colon returns all rows.
cell_attributes.loc[:,'n_counts']
```

Here's the general syntax `df.loc[ ROWS [, COLUMNS] ]` where ROWS and COLUMNS can be START INDEX : END INDEX or NAMES, depending on whether you have given your rows names or are using the default numerical labels

If you just want a whole column, there's a shortcut format

```
cell_attributes['n_counts']
```

iloc allows us to subset rows and colums by index number. This is useful if we want to subset multiple rows or columns without typing index names.

Lets take a look at the column names first and see if we can slice out the ones we'd like to keep.

```
# Return column names
cell_attributes.columns.values
cell_attributes.columns.values[[0,1,3,5,7]]
```

Note `[[ ]]` allows us to mention a list of columns.

Now we can apply the same indexing pattern to our **iloc** method to return only the columns we're interested in.

```
# Return columns 0, 1, 3, 5, and 7
cell_attributes.iloc[:,[0,1,3,5,7]]

# Return rows 1 through 5 and columns 0, 1, 3, 5, and 7
cell_attributes.iloc[:5,[0,1,3,5,7]]
```

Ordering dataframes by column values

Here we'll take look at ordering our data by a particular column value, or multiple column values.

```
# Set ascending=True to reverse the order
cell_attributes.sort_values('n_counts', ascending=False)

# Sort by multiple columns in different directions
cell_attributes.sort_values(by=['tree_ident', 'n_counts'], ascending=[True, False])
```

Subsetting data by condition

Understanding how to subset your data using conditional operations is *very*, *very* useful. You'll often encounter situations where you want to filter your data on a certain set of parameters to reduce it to a more "meaningful" state.

```
# Subsetting on a single condition
cell_attributes.loc[(cell_attributes['tree_ident'] == 1)]
#or in shorter form if you want all the columns and rows that match the condition
cell_attributes[(cell_attributes['tree_ident'] == 1)]
```

In the example below we chain boolean operators together to achieve results that satisfy multiple conditions. You can make these statments complex as you'd like.

Note: Pandas uses the bitwise logical operators (see earlier lecture). A pipe symbol `|` represents `or`, and an ampersand symbol `&` represents `and`. The backslashes in code simply allow us to break up our statement at arbitrary points for readbility.

```
# Subsetting on multiple conditions.
cell_attributes[
    (cell_attributes['tree_ident'] == 1) | \
    (cell_attributes['tree_ident'] == 2) & \
    (cell_attributes['n_genes'] > 1000)]
```

What's actually going on here? The rows in the data frame are actually subsetted on a vector of True/False statements. That is, for every row for which the condition evaluates to True will be returned. If we examine the boolean statements placed within `cell_attributes.loc[]`, you can see why this is occuring.

```
cell_attributes['tree_ident'] == 1 | \
    (cell_attributes['tree_ident'] == 2) & \
    (cell_attributes['n_genes'] > 1000)
```

Performing mathmatical operations on vectors

Lets look at a couple examples where we apply caculations to our data frame. First lets calculate some summary statistics. This can be a useful when viewing our results for the first time to get a handle on how our data is distributed.

```
# Returning summary statistics for all columns
cell_attributes.describe()

# Returning summary statistics for a single column
cell_attributes['n_counts'].describe() # if you are working on a whole column
```

```
# reports the following summary statistics
# count
# mean
# std
# min
# 25%
# 50%
# 75%
# max
```

`n_counts` refers to the number of counts for "unique molecular identifiers", which are barcodes for individual transcripts within in a single cell. Ideally, if the number of `n_counts` is high, then the number of genes per cell should also be high. The number of genes per cell is in the `n_genes` column. Lets see if this observation holds true by calculating the pairwise correlation between these two variables.

```
# Simply add the .corr() method to your dataframe subset
cell_attributes.loc[:,['n_counts','n_genes']].corr()
```

That summarizes our introduction to Pandas. As you can see, Pandas greatly simplifies the process of exploring and making calculations in data frames and matricies. Check out the link below for the offical documentation.

Pandas Documentation