Lecture 10: Analyzing data I

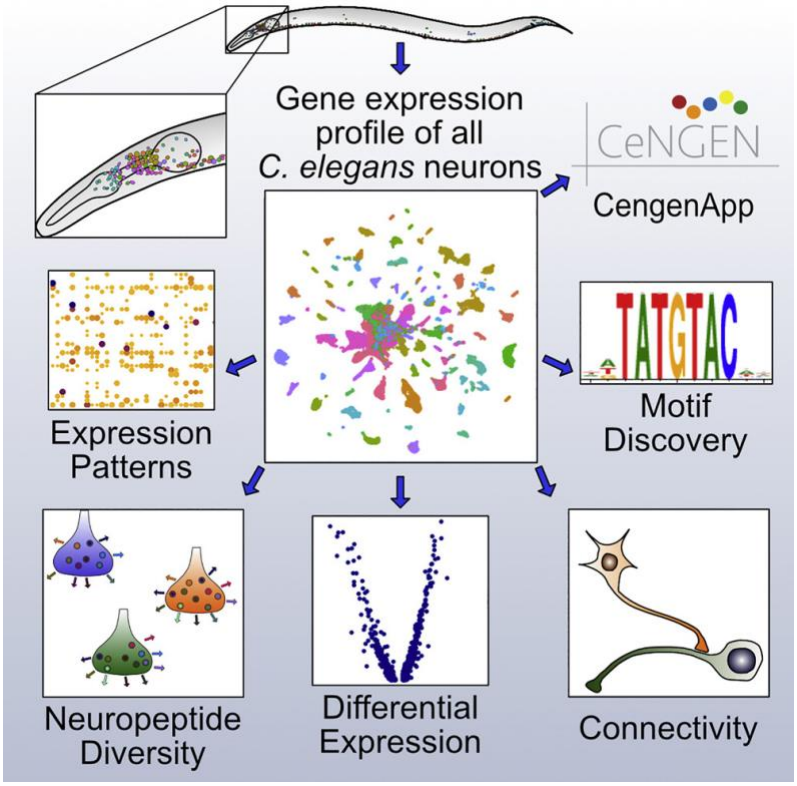
Now that we know how to read in data, we are going to go over some approaches to analyze data. We will some of the features from the pandas module that allows you to rapidly analyze data. Pandas is huge though, so we won't cover all of the things you can do with this module. My main goal is to get you comfortable with using the module. Much of the information in here can be found in some form in the documentation:

http://pandas.pydata.org/pandas-docs/stable/

Topic – Single cell transcriptomics

It is now possible to transcriptionally profile single cells. While our body is made of trillions of cells, most of these cells are functionally distinct from each other. For example, a skin cell is different from a neuron. Further, your brain is filled with neurons of different cell-types. An important reason that cells are different from each other is they each express a different complement, or amount, of each gene. For example, some neurons release different neurotransmitters, like glutamate, acetylcholine, dopamine, or serotonin, because they express enzymes that create those molecules and proteins that package the neurotransmitters into vesicles. Other neurons express protein receptors that sense these different neurotransmitters, which causes them to fire when nearby neurons release the neurotransmitter.

Currently there is a lot of work dedicated towards single cell transcriptomics in all species. An example of this is in *C. elegans*, a small nematode that is a workhorse species of genetics research. Interestingly, every individual worm of this species have exactly 302 neurons that are belong to one of 118 distinct cell types. These neuron classes include many similar to human cell types: excitatory and inhibitory motor neurons, dopaminergic and serotoninergic neurons that are involved in learning, olfactory sensing neurons, and interneurons. A recent paper reported the gene expression of all 118 neuronal cell types. ~70,000 neurons were profiled and classified into these classes using machine learning approaches:



This data is available for analysis at the following website. For this class we will analyze a csv file containing information for each gene and it's expression in the 118 neuron classes. It is available in the ExampleData folder. First let's open it with pandas.

Topic – Basics of a dataframe object

```
>>> import pandas as pd
>>> dt = pd.read_csv('021821_medium_threshold2.csv', index_col = 0)
```

Here, we have read in a csv and created a dataframe object, which we can see as so:

```
>>> type(dt)
<class 'pandas.core.frame.DataFrame'>
```

If we want to see what this object is like, we can just type the name:

```
>>> dt
      gene_name     Wormbase_ID         ADA         ADE         ADF         ADL  ...          VB        VB01        VB02          VC       VC_4_5       VD_DD
1         nduo-6  WBGene00010957  7531.007875  12141.685889  3255.836119  3509.352698  ...  12174.467291  8882.122440  6999.569000  4403.522328  3954.239257  9000.479633
2         ndfl-4  WBGene00010958  1075.540509   1355.504184   593.604224   534.883641  ...    810.037857   539.706719   490.172543   463.079238   177.566634  1134.297704
3         MTCE.7  WBGene00014454  1175.681680   1574.857532   159.518954   234.534995  ...   1539.757971  1110.195374   866.906513   593.188912   290.830469   986.987877
4         nduo-1  WBGene00010959  1732.528365    676.950710   640.879206  1252.851856  ...    990.757384   615.048301   687.607927   864.504816   851.927266  1261.443259
5          atp-6  WBGene00010960  4470.880547   5169.989040  2648.946349  2080.622062  ...   3986.484236  2546.816138  2556.891073  2395.979005  2473.374134  6113.173536
...          ...             ...          ...          ...          ...          ...  ...  ...          ...          ...          ...          ...          ...          ...
13665    C50D2.3  WBGene00016807     0.000000     0.000000     0.000000     0.000000  ...     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000
13666     gst-35  WBGene00001783     0.000000     0.000000     0.000000     0.000000  ...     0.000000     0.000000     0.000000     0.000000     0.000000     0.000000
13667  C28C12.11  WBGene00016181     0.000000     0.000000     0.000000     0.000000  ...     0.000000     0.000000     0.000000     0.000000   151.823000     0.000000
13668   T12F5.2  WBGene00020467     0.000000     0.000000     0.000000     0.000000  ...     0.000000     0.000000     0.000000     0.000000   335.116500     0.000000
13669      ref-1  WBGene00004334     0.000000     0.000000     0.000000     0.000000  ...     0.000000     0.000000     0.000000     0.000000   151.379400     0.000000

[13669 rows x 130 columns]
```

There are 13,669 rows and 130 columns in this dataframe. **IMPORTANT**: it's important to remember that there are two fundamental ways to access data, using the column name and using the row name.

Each column has a unique name to it (called the column name) and each row has a unique name to it (called the index). You can see the names using the following two attributes:

```
>>> dt.columns
Index(['gene_name', 'Wormbase_ID', 'ADA', 'ADE', 'ADF', 'ADL', 'AFD', 'AIA',
       'AIB', 'AIM',
       ...
       'URX', 'URY', 'VA', 'VA12', 'VB', 'VB01', 'VB02', 'VC', 'VC_4_5',
       'VD_DD'],
      dtype='object', length=130)
>>> dt.index
Int64Index([    1,     2,     3,     4,     5,     6,     7,     8,     9,
                10,
            ...
            13660, 13661, 13662, 13663, 13664, 13665, 13666, 13667, 13668,
            13669],
           dtype='int64', length=13669)
```

As you can see, there are 130 unique columns and 13,669 unique rows of data.

Topic – Filtering data by row and column names

If you would like you can return data for a specific column using a slicing operator with the name of the column:

```
>>> dt['ADA']
1        7531.007875
2        1075.540509
3        1175.681680
4        1732.528365
5        4470.880547
            ...
13665       0.000000
13666       0.000000
13667       0.000000
13668       0.000000
13669       0.000000
Name: ADA, Length: 13669, dtype: float64
```

If you want multiple columns, you can put a list of column names in the slicing operator:

```
>>> dt[['ADA','ADE']]
             ADA          ADE
```

```
1      7531.007875  12141.685889
2      1075.540509   1355.504184
3      1175.681680   1574.857532
4      1732.528365    676.950710
5      4470.880547   5169.989040
...            ...           ...
13665     0.000000      0.000000
13666     0.000000      0.000000
13667     0.000000      0.000000
13668     0.000000      0.000000
13669     0.000000      0.000000
```

Notice in both cases, the index of the data remains as part of the object returned.

If you want to remove specific columns, use the drop method:

```
>>> dt.drop(columns = 'ADA')
>>> dt.drop(columns = ['ADA','ADE'])
```

If you want to return specific rows, use loc attribute. Loc is a little funny. It is kind of like a combination of a slicing operator and a method. One thing that is funny about it is that you also **have** to provide a list of index names you would like, even if you only want one row.

```
>>> dt.loc[1]
KeyError: 1
>>> dt.loc[[1]]
  gene_name     Wormbase_ID         ADA         ADE         ADF         ADL  ...          VB       VB01      VB02          VC
VC_4_5       VD_DD
1    nduo-6  WBGene00010957  7531.007875  12141.685889  3255.836119  3509.352698  ...  12174.467291
8882.12244  6999.569  4403.522328  3954.239257  9000.479633


[1 rows x 130 columns]
>>> dt.loc[[1,2]]
  gene_name     Wormbase_ID         ADA         ADE         ADF         ADL  ...          VB       VB01      VB02          VC
VC_4_5       VD_DD
1    nduo-6  WBGene00010957  7531.007875  12141.685889  3255.836119  3509.352698  ...  12174.467291
8882.122440  6999.569000  4403.522328  3954.239257  9000.479633
2    ndfl-4  WBGene00010958  1075.540509   1355.504184   593.604224   534.883641  ...    810.037857   539.706719
490.172543   463.079238   177.566634  1134.297704
```

```
[2 rows x 130 columns]
```

Notice you are selecting on the index name. If you use a 0 (which doesn't exist in the index, you get an error.

```
>>> dt.loc[[0]]
KeyError: "None of [Int64Index([0], dtype='int64')] are in the [index]"
```

You can also combine row and column selectors:

```
>>> dt.loc[[1,2]][['ADA','ADF','VB']]
         ADA         ADF           VB
1  7531.007875  3255.836119  12174.467291
2  1075.540509   593.604224    810.037857
```

To access the value of an individual cell, you can use loc like so:

```
>>> dt.loc[1,'ADA']
7531.00787529841
```

One thing that is rather confusing is that some similar methods do not return the value of the cell. Note that these two approaches do not return floats, but rather DataSeries objects:

```
>>> dt.loc[[1],'ADA']
1    7531.007875
Name: ADA, dtype: float64
>>> dt.loc[[1]]['ADA']
1    7531.007875
Name: ADA, dtype: float64
```

Topic – Filtering data by values

You can also sort out rows from the table if there value for a certain column satisfies some criteria. In this case you use the slicing operator, but within it you should include the criteria. For example, if you wanted to identify all the genes with no expression in ADA you would use the dot operator with the column name like so.

```
>>> dt[dt.ADA == 0]
      gene_name     Wormbase_ID  ADA       ADE       ADF       ADL      AFD  ...       VA12       VB       VB01       VB02
VC      VC_4_5      VD_DD
4949      aass-1  WBGene00019819  0.0   3.704172   0.000000   0.000000  2.981855  ...    0.000000   0.000000
0.000000   0.000000   0.000000    0.000000   0.000000
```

```
4950      fln-1  WBGene00022048  0.0  25.998111   5.094926   5.523742  3.921382  ...   38.476396  24.649429
21.064056  15.495714  12.913250    0.000000  30.202766
4951      spl-1  WBGene00004981  0.0   7.198891  20.673656  10.177956  9.531302  ...    0.000000   5.138134
0.000000   3.655022  14.060404   11.374391  14.453758
4952    pqbp-1.2  WBGene00020295  0.0   6.692457   0.871237   0.169525  0.000000  ...   73.697992   3.116835
7.156536   4.603081   0.000000    0.000000   0.000000
4953    Y66H1A.4  WBGene00022046  0.0  17.746668  12.647100   7.135530  5.939107  ...   30.839500  23.400891
36.208975  19.758165  11.124660    9.546957  15.931612
...         ...           ... ...       ...       ...       ...       ... ...       ...       ...       ...       ...       ...       ...       ...
13665    C50D2.3  WBGene00016807  0.0   0.000000   0.000000   0.000000  0.000000  ...   86.312370   0.000000
0.000000   0.000000   0.000000    0.000000   0.000000
13666     gst-35  WBGene00001783  0.0   0.000000   0.000000   0.000000  0.000000  ...  555.358600   0.000000
0.000000   0.000000   0.000000    0.000000   0.000000
13667  C28C12.11  WBGene00016181  0.0   0.000000   0.000000   0.000000  0.000000  ...    0.000000   0.000000
0.000000   0.000000   0.000000  151.823000   0.000000
13668    T12F5.2  WBGene00020467  0.0   0.000000   0.000000   0.000000  0.000000  ...    0.000000   0.000000
0.000000   0.000000   0.000000  335.116500   0.000000
13669      ref-1  WBGene00004334  0.0   0.000000   0.000000   0.000000  0.000000  ...    0.000000   0.000000   0.000000
0.000000   0.000000  151.379400   0.000000


[8721 rows x 130 columns]
```

You can also access a column using the slicing operator instead of the dot operator:

```
>>> dt[dt['ADA'] == 0]
```

Some other valid comparisons are:

```
>>> dt[dt.ADA != 0]
>>> dt[dt.ADA > 0]
>>> dt[dt.gene_name == 'ndfl-4']
>>> dt[dt.gene_name.isin(['glb-5', 'glb-6'])]
```

You can also filter on multiple conditions if you wrap each one in a () and use & or | for and and or comparisons. E.g.:

```
>>> dt[(dt.ADA == 0) & (dt.URX == 0)]
[6039 rows x 130 columns]
```

How does filtering work? If you look at what is returned by the conditional statement:

```
>>> dt.ADA > 0
1        True
2        True
3        True
4        True
5        True
        ...
13665    False
13666    False
13667    False
13668    False
13669    False
Name: ADA, Length: 13669, dtype: bool
```

For each row, a DataSeries of Booleans is returned. If you send that to the dataframe with a slicing operator, it keeps or removes the row based upon the Boolean.

You can also use apply, a method of a data series, in order to create this DataSeries of Booleans. This takes a lambda function that allows you to perform custom functions. So if you want to search for genes that have the letters ‘dop’ in the, you can do something like this:

```
>>> dt['gene_name'].apply(lambda x: 'dop' in x)
1        False
2        False
3        False
4        False
5        False
        ...
13665    False
13666    False
13667    False
13668    False
13669    False
Name: gene_name, Length: 13669, dtype: bool
```

Then if you want to filter the dataframe to only see those rows:

```
>>> dt[dt['gene_name'].apply(lambda x: 'dop' in x)]
     gene_name     Wormbase_ID        ADA        ADE        ADF        ADL        AFD  ...       VA12        VB       VB01
VB02        VC      VC_4_5      VD_DD
```

```
1926     dop-6  WBGene00016037   14.174411   0.000000   74.116565   0.000000   6.049744  ...    0.000000   11.983566
10.122069   12.915835  27.597373  119.609458   36.303283
1930     dop-1  WBGene00001052  161.630962  26.838409    0.000000   0.000000   0.000000  ...  722.480537
182.782705  272.475075  178.838877   0.000000    0.000000    0.000000
3830     dop-5  WBGene00011382  306.411418   0.000000  120.502557  23.047089  38.872716  ...   55.983189
99.794319    0.000000   75.881567   0.000000    0.000000    0.000000
6164     dop-2  WBGene00001053    0.000000  84.847420   50.329887   0.000000   0.000000  ...  226.834111
100.708352  167.419689   55.657727   0.000000    0.000000  111.340177
7279     dop-3  WBGene00020506    0.000000   0.000000  173.080600   0.000000   0.000000  ...  100.443500
146.965100   60.120770   95.091670   0.000000    0.000000  137.818800
10573    dop-4  WBGene00016872    0.000000   0.000000    0.000000   0.000000   0.000000  ...    0.000000    0.000000
0.000000    0.000000   0.000000    0.000000    0.000000

[6 rows x 130 columns]
```

Topic – Iterate through the dataframe

The easiest way to iterate through the dataframe is to use the iterrows() method:

```
>>> for i,rec in dt.iterrows():
...   break
...
>>> i
1
>>> rec
gene_name            nduo-6
Wormbase_ID    WBGene00010957
ADA              7531.007875
ADE             12141.685889
ADF              3255.836119
                    ...
VB01              8882.12244
VB02               6999.569
VC               4403.522328
VC_4_5            3954.239257
VD_DD             9000.479633
Name: 1, Length: 130, dtype: object
```

Notice that this returns a tuple of length 2. The first element is the index and the second element is a

dataseries where the index is the column names and the data are the values. To access the data:

```
>>> rec.gene_name
'nduo-6'
>>> rec.ADE
12141.6858888786
```