Part 2. Performing analysis and advanced visualizing & analyzing methods

Using Python for single cell RNA seq data analysis

Seoul National University, Department of Biological Science

Laboratory of Development and Disease Modeling

Jong Hwi Kim

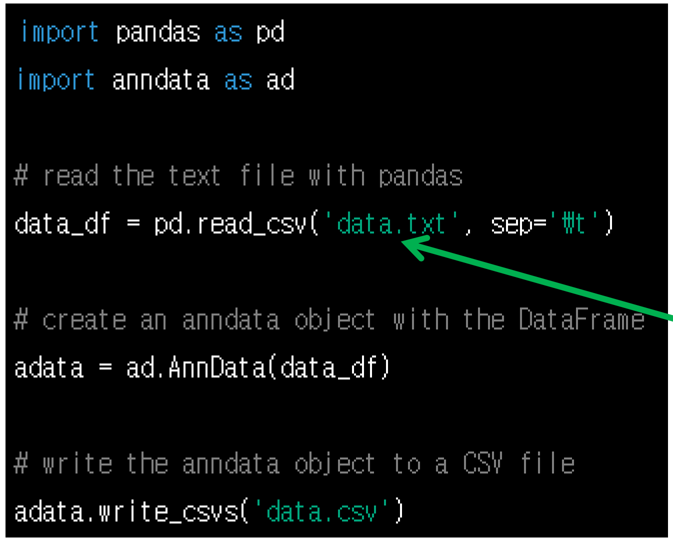
Step 3. Understanding the work flow of pre-processing

Parsing the data, Reading and trimming the dataset, Cell cycle regression, Doublet removal, Removing mitochondrial and ribosomal gene, Basic PCA and neighbor analysis

3. Pre-processing

3.1. Parsing the dataframe

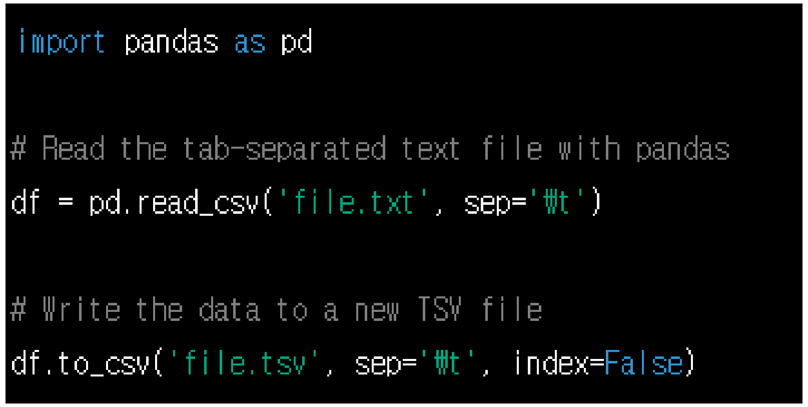
- Use either 1) !wget command for importing data directly from URL or 2) adata.read function to import data from your local directory
- Copying and pasting the path of the file is necessary
- So, make sure to 1) Not contain non-English words for the directory path 2) Not use ‘U’ for the first letter and 3) store files under numerous folders in order to keep your directory path compact
- Also, make sure to unzip all the materials before parsing it into python environment
- You can import .txt files into .tsv or .csv by adding ‘sep=\t’ before closing the parameter line



```
import pandas as pd
import anndata as ad

# read the text file with pandas
data_df = pd.read_csv('data.txt', sep='₩t')

# create an anndata object with the DataFrame
adata = ad.AnnData(data_df)

# write the anndata object to a CSV file
adata.write_csvs('data.csv')
```

<- Script example for converting .txt file into .csv file

Script example for converting .txt file into tsv file ->

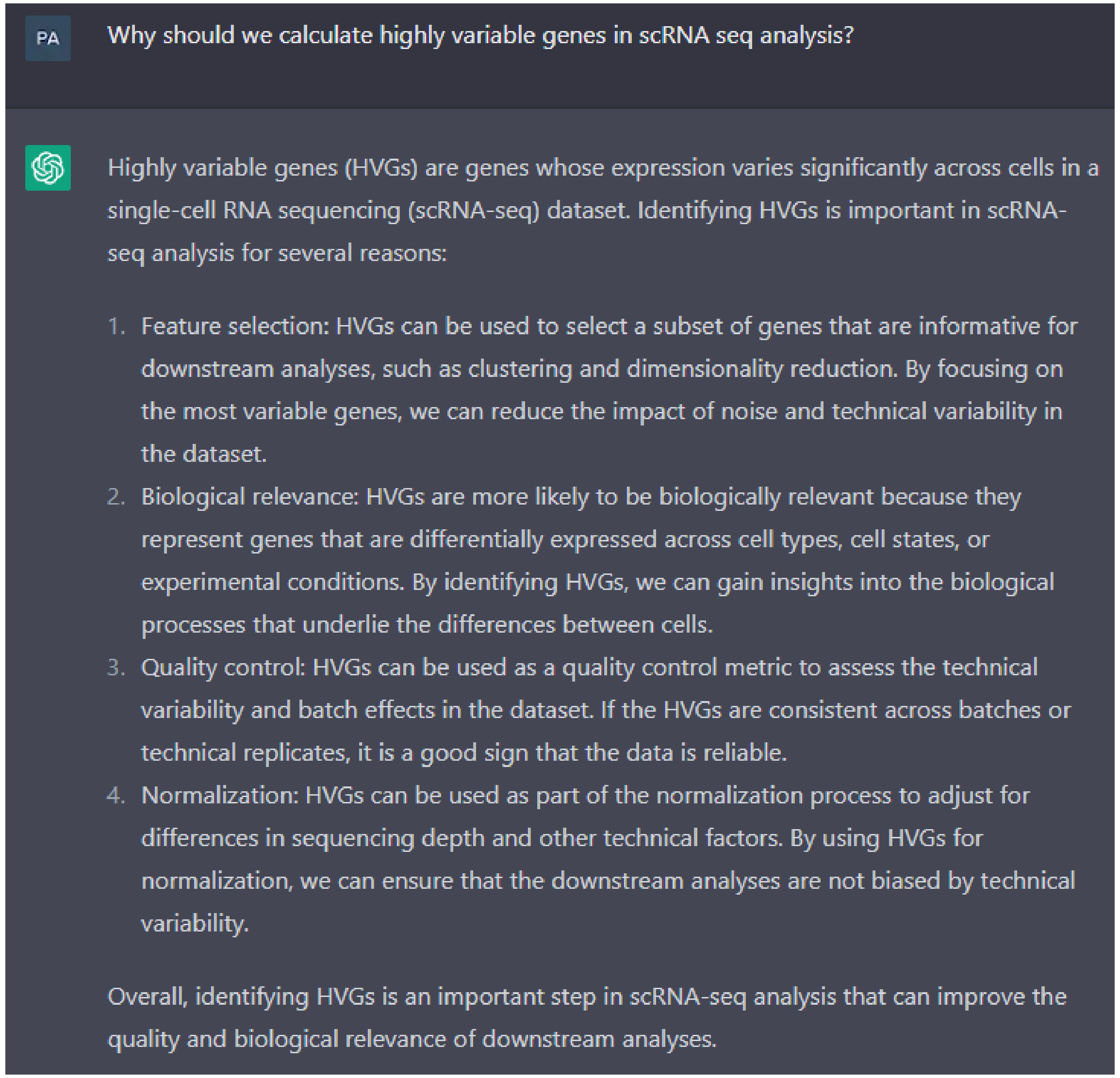


```
import pandas as pd

# Read the tab-separated text file with pandas
df = pd.read_csv('file.txt', sep='₩t')

# Write the data to a new TSV file
df.to_csv('file.tsv', sep='₩t', index=False)
```

Copy and paste the path for your file

3. Pre-processing

3.2. Calculating highly variable genes



PA Why should we calculate highly variable genes in scRNA seq analysis?

Highly variable genes (HVGs) are genes whose expression varies significantly across cells in a single-cell RNA sequencing (scRNA-seq) dataset. Identifying HVGs is important in scRNA-seq analysis for several reasons:

1. Feature selection: HVGs can be used to select a subset of genes that are informative for downstream analyses, such as clustering and dimensionality reduction. By focusing on the most variable genes, we can reduce the impact of noise and technical variability in the dataset.
2. Biological relevance: HVGs are more likely to be biologically relevant because they represent genes that are differentially expressed across cell types, cell states, or experimental conditions. By identifying HVGs, we can gain insights into the biological processes that underlie the differences between cells.
3. Quality control: HVGs can be used as a quality control metric to assess the technical variability and batch effects in the dataset. If the HVGs are consistent across batches or technical replicates, it is a good sign that the data is reliable.
4. Normalization: HVGs can be used as part of the normalization process to adjust for differences in sequencing depth and other technical factors. By using HVGs for normalization, we can ensure that the downstream analyses are not biased by technical variability.

Overall, identifying HVGs is an important step in scRNA-seq analysis that can improve the quality and biological relevance of downstream analyses.

```
sc.pp.highly_variable_genes(adata, n_top_genes = 2000, subset = True, flavor = 'seurat_v3')
```

```
sc.pl.highly_variable_genes(adata)
```

- Calculating HVG is very important before you go into deeper analysis
- If this step is neglected, you will not be able to calculate PCA, neighbors, cluster etc.
- The ‘flavor’ parameter could be altered according to your taste, but using ‘Seurat_v3’ will be most quintessential
- After calculating hvg, add Python cell to see if ‘hvg’ was added to your adata.uns

```
In [12]:  adata

Out[12]: AnnData object with n_obs × n_vars = 6099 × 2000
             var: 'n_cells', 'highly_variable', 'highly_variable_rank', 'means', 'variances', 'variances_norm'
             uns: 'hvg'
```

3. Pre-processing — 3.3. Cell cycle regression

- This step is to remove unwanted gene expression level variation caused by cell cycle difference
- This could be done during PCA analysis or linear regression too, but adding this step is highly recommended
- Simplest way to do this: Use “sc.pp.regress_out(adata, ['S_score', 'G2M_score’])”

```
cell_cycle_genes = [x.strip() for x in open('./regev_lab_cell_cycle_genes_10X.txt')]
s_genes = cell_cycle_genes[:43]
g2m_genes = cell_cycle_genes[43:]
cell_cycle_genes = [x for x in cell_cycle_genes if x in adata_all.var_names]

sc.tl.score_genes_cell_cycle(adata_all, s_genes=s_genes, g2m_genes=g2m_genes)
```

Download txt file from URL below & paste path of the file
https://www.science.org/doi/full/10.1126/science.aad0501

```
adata_hvg.obs['phase'] = pd.Categorical(adata_all.obs['phase'])
ref_cluster = pd.Categorical(adata_hvg.obs['phase'],categories=['S','G2M','G1','Cycling','Non-Cycling'])
x = adata_hvg.obs['phase'] =='S'
ref_cluster[x] = 'Cycling'
x = adata_hvg.obs['phase'] =='G2M'
ref_cluster[x]='Cycling'
x = adata_hvg.obs['phase'] =='G1'
ref_cluster[x]='Non-Cycling'
adata_hvg.obs['proliferation'] = ref_cluster
adata_hvg.obs['proliferation'].cat.remove_unused_categories(inplace=True)
```

```
adata_hvg.obs['S_score']=adata_all.obs['S_score']
adata_hvg.obs['G2M_score']=adata_all.obs['G2M_score']
adata_hvg.uns['proliferation_colors']=['#377eb8','#bdbdbd']
```

Convert this into the annotation that you provided for imported data

Standard procedure for cell cycle regression

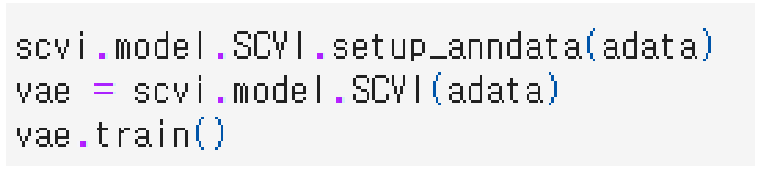
**For applying linear regression

```
sc.pp.regress_out(adata, ['total_counts'])
```

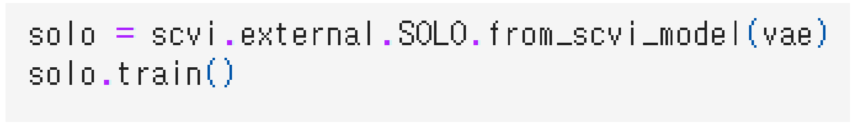
3. Pre-processing

3.4. Doublet removal

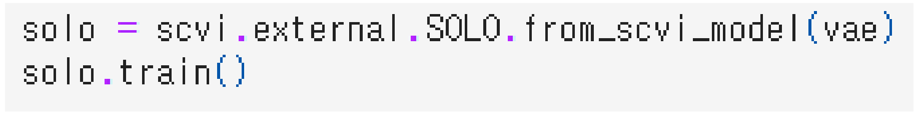
- Doublet removal starts with calculating doublets
- Import scvi-tools from your library, and run the scripts written below



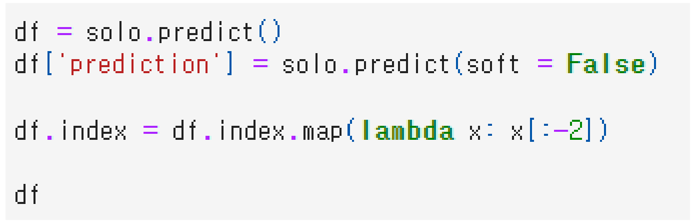
```
scvi.model.SCVI.setup_anndata(adata)
vae = scvi.model.SCVI(adata)
vae.train()
```



```
solo = scvi.external.SOLO.from_scvi_model(vae)
solo.train()
```

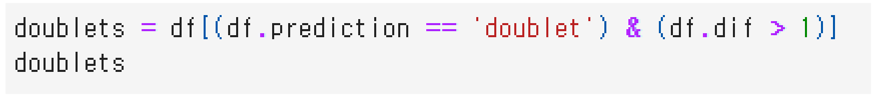


```
solo = scvi.external.SOLO.from_scvi_model(vae)
solo.train()
```

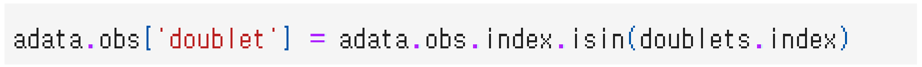


```
df = solo.predict()
df['prediction'] = solo.predict(soft = False)

df.index = df.index.map(lambda x: x[:-2])

df
```

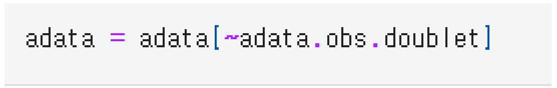
**Leave the () unfilled. It does not have to be filled



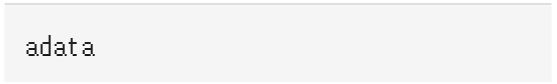
```
doublets = df[(df.prediction == 'doublet') & (df.dif > 1)]
doublets
```



```
adata.obs['doublet'] = adata.obs.index.isin(doublets.index)
```



```
adata = adata[~adata.obs.doublet]
```



```
adata
```

**Check that 1) the cell number is reduced and 2) adata.obs contain 'doublet' before proceeding to the next step

3.5. Removing mitochondrial and ribosomal gene

- scRNA seq data matrix contains gene counts for non-translating RNA, mitochondrial gene and ribosomal gene. These can be contained in the HVG counts or marker genes and make results confusing.
- Removing the mitochondrial gene is done by labeling gene ID starting with MT- or mt- or Mt- into mitochondrial gene
- Removing the ribosomal gene is done by importing table of ribosomal genes
- Follow the codes below and always check the adata.var for confirming successful labeling

```
adata.var['mt'] = adata.var.index.str.startswith('MT-')
```

```
ribo_url = "http://software.broadinstitute.org/gsea/msigdb/download_geneset.jsp?geneSetName=KEGG_RIBOSOME&fileType=txt"
```

```
ribo_genes = pd.read_table(ribo_url, skiprows=2, header = None)
ribo_genes
```

```
adata.var['ribo'] = adata.var_names.isin(ribo_genes[0].values)
```

```
sc.pp.calculate_qc_metrics(adata, qc_vars=['mt', 'ribo'], percent_top=None, log1p=False, inplace=True)
```

```
sc.pl.violin(adata, ['n_genes_by_counts', 'total_counts', 'pct_counts_mt', 'pct_counts_ribo'],
             jitter=0.4, multi_panel=True)
```

```
adata = adata[adata.obs.pct_counts_mt < 20]
```

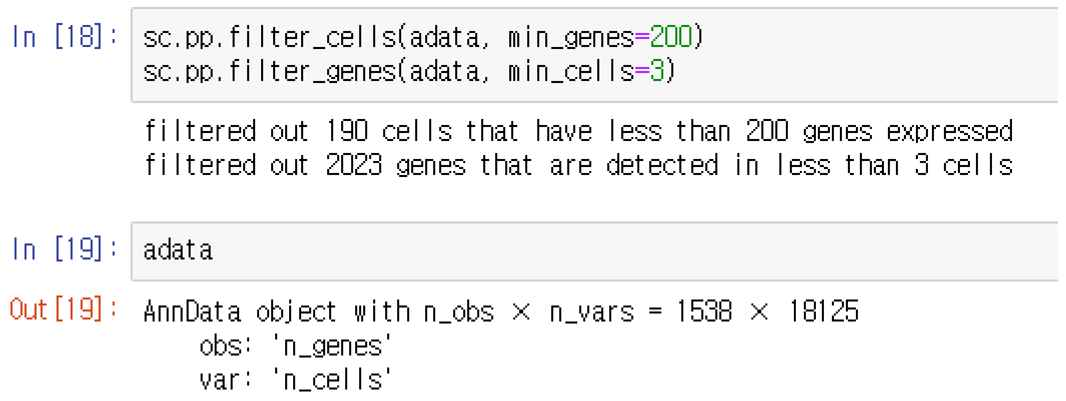
```
adata = adata[adata.obs.pct_counts_ribo < 2]
```

3. Pre-processing

3.6. Cut-off (min_genes, min_cells)/ Normalization

- Cut-off filter should be set according to the characteristic of datasets
- Standard: Minimal expressing gene number = 200, minimal cell number for certain gene expression =3
- Prior to calculating PC axis and further analyzing the data, gene counts should be normalized
- Normalizing the read count for total_sum=1e4 UMI is the most common way
- After normalizing, converting the gene count to logarithmic scale is also available & recommended

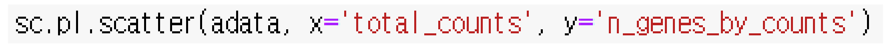
1) Cut-off



```
In [18]:  sc.pp.filter_cells(adata, min_genes=200)
          sc.pp.filter_genes(adata, min_cells=3)

          filtered out 190 cells that have less than 200 genes expressed
          filtered out 2023 genes that are detected in less than 3 cells

In [19]:  adata

Out[19]:  AnnData object with n_obs × n_vars = 1538 × 18125
              obs: 'n_genes'
              var: 'n_cells'
```

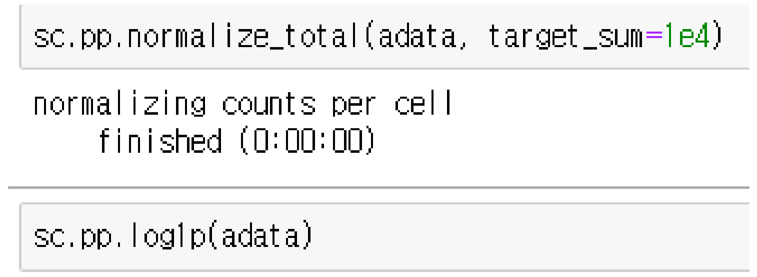
*After filtering, ‘n_genes’ and ‘n_cells’ will be added in the dataframe

2) Normalization



```
sc.pl.scatter(adata, x='total_counts', y='n_genes_by_counts')
```

*Check the total counts of each genes before normalization



```
sc.pp.normalize_total(adata, target_sum=1e4)

normalizing counts per cell
    finished (0:00:00)

sc.pp.log1p(adata)
```

*Scaling the data is not necessary if your data has subtle variance among reads

Step 4. Calculating PCA/t-SNE/UMAP and clustering

Parsing the data, Reading and trimming the dataset, Removing mitochondrial and ribosomal gene, Basic PCA and neighbor analysis

4.1. Calculating PCA and neighboring

- For calculating PCA and neighboring, input the codes written below

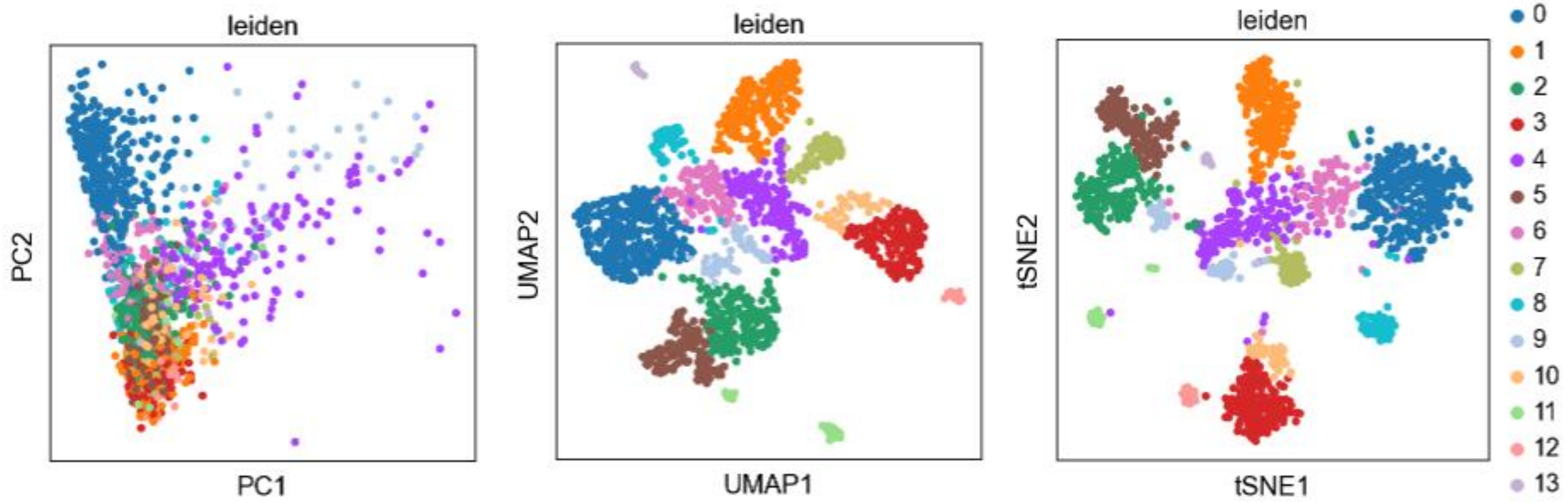
```
sc.tl.pca(adata, svd_solver='arpack')
```

```
sc.pl.pca_variance_ratio(adata, log=True, n_pcs = 50)
```

```
sc.tl.leiden(adata)
```

```
sc.pl.pca(adata, color=['leiden'])
```

```
sc.tl.umap(adata)
sc.pl.umap(adata, color=['leiden'])
```



**Choose the best version of scattered plot that suits your purpose the most
**You should perform PCA and neighboring for calculating UMAP and tSNE

4.2. Clustering

- Clustering is done by exploiting PC axis and neighboring data obtained from previous steps
- Most commonly used clustering algorithm is ‘Louvain’ or ‘Leiden’.
- Two methods are installed in the initial step, so you can use this without adding any packages
- If not, you should install or update clustering method by installing ‘leidenalg’ package

```
adata = adata[:, adata.var.highly_variable]
```

```
sc.pp.regress_out(adata, ['total_counts', 'pct_counts_mt', 'pct_counts_ribo'])
```

```
sc.pp.scale(adata, max_value=10)
```

```
sc.tl.pca(adata, svd_solver='arpack')
```

```
sc.pl.pca_variance_ratio(adata, log=True, n_pcs = 50)
```

```
sc.pp.neighbors(adata, n_pcs = 30)
```

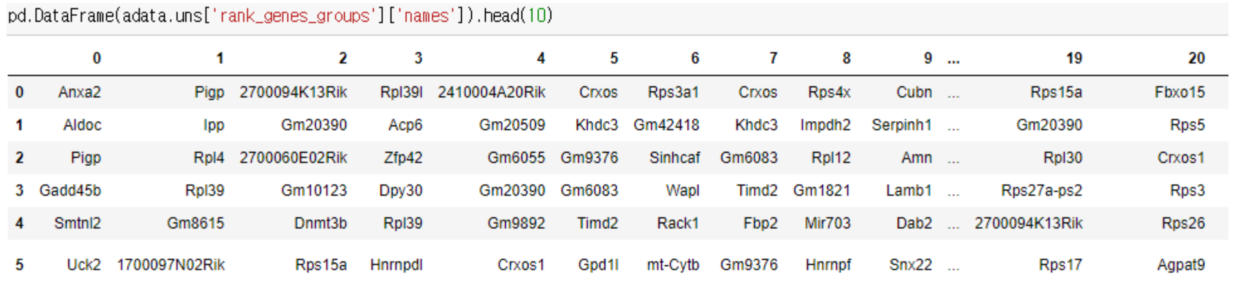
```
#!pip install leidenalg
```

```
sc.tl.leiden(adata, resolution = 0.5)
```

**User’s tip

- After clustering with leiden, call adata.obs and make dataframe with top 10 differentially expressed genes for each cluster
- By searching the reference, annotate them and rename the cluster name into the common name

```
sc.tl.rank_genes_groups(adata, 'leiden', method='t-test')
sc.pl.rank_genes_groups(adata, n_genes=25, sharey=False)
```



```
pd.DataFrame(adata.uns['rank_genes_groups']['names']).head(10)
```

	0	1	2	3	4	5	6	7	8	9	...	19	20
0	Anxa2	Pigp	2700094K13Rik	Rpl39l	2410004A20Rik	Crxos	Rps3a1	Crxos	Rps4x	Cubn	...	Rps15a	Fbxo15
1	Aldoc	Ipp	Gm20390	Acp6	Gm20509	Khdc3	Gm42418	Khdc3	Impdh2	Serpinh1	...	Gm20390	Rps5
2	Pigp	Rpl4	2700060E02Rik	Zfp42	Gm6055	Gm9376	Sinhcaf	Gm6083	Rpl12	Amn	...	Rpl30	Crxos1
3	Gadd45b	Rpl39	Gm10123	Dpy30	Gm20390	Gm6083	Wapl	Timd2	Gm1821	Lamb1	...	Rps27a-ps2	Rps3
4	Smtnl2	Gm8615	Dnmt3b	Rpl39	Gm9892	Timd2	Rack1	Fbp2	Mir703	Dab2	...	2700094K13Rik	Rps26
5	Uck2	1700097N02Rik	Rps15a	Hnrnpdl	Crxos1	Gpd1l	mt-Cytb	Gm9376	Hnrnpf	Snx22	...	Rps17	Agpat9