Preprocessing and clustering 3k PBMCs

In May 2017, this started out as a demonstration that Scanpy would allow to reproduce most of Seurat's guided clustering tutorial (Satija et al., 2015).

We gratefully acknowledge Seurat's authors for the tutorial! In the meanwhile, we have added and removed a few pieces.

The data consist of *3k PBMCs from a Healthy Donor* and are freely available from 10x Genomics (here from this webpage). On a unix system, you can uncomment and run the following to download and unpack the data. The last line creates a directory for writing processed data.

[1]:
```
# !mkdir data
# !wget http://cf.10xgenomics.com/samples/cell-exp/1.1.0/pbmc3k/pbmc3k_filtered_gene_bc_matrices.tar.gz -O data/pbmc3k_filtered_gene_bc_matrices.tar.gz
# !cd data; tar -xzf pbmc3k_filtered_gene_bc_matrices.tar.gz
# !mkdir write
```

Note

Download the notebook by clicking on the *Edit on GitHub* button. On GitHub, you can download using the *Raw* button via right-click and *Save Link As*. Alternatively, download the whole scanpy-tutorial repository.

Note

In Jupyter notebooks and lab, you can see the documentation for a python function by hitting `SHIFT + TAB`. Hit it twice to expand the view.

[2]:
```
import numpy as np
import pandas as pd
import scanpy as sc
```

[3]:
```
sc.settings.verbosity = 3             # verbosity: errors (0), warnings (1), info (2), hints (3)
sc.logging.print_header()
sc.settings.set_figure_params(dpi=80, facecolor='white')
```

```
scanpy==1.6.0 anndata==0.7.5.dev7+gefffdfb umap==0.4.2 numpy==1.18.1 scipy==1.4.1
pandas==1.0.3 scikit-learn==0.22.1 statsmodels==0.11.0 python-igraph==0.7.1
leidenalg==0.7.0
```

[4]:
```
results_file = 'write/pbmc3k.h5ad'  # the file that will store the analysis results
```

Read in the count matrix into an AnnData object, which holds many slots for annotations and different representations of the data. It also comes with its own HDF5-based file format: `.h5ad`.

[5]:
```
adata = sc.read_10x_mtx(
    'data/filtered_gene_bc_matrices/hg19/',  # the directory with the `.mtx` file
    var_names='gene_symbols',                # use gene symbols for the variable
names (variables-axis index)
    cache=True)                              # write a cache file for faster
subsequent reading
```

```
... reading from cache file cache/data-filtered_gene_bc_matrices-hg19-matrix.h5ad
```

Note

See anndata-tutorials/getting-started for a more comprehensive introduction to `AnnData`.

[6]:
```
adata.var_names_make_unique()  # this is unnecessary if using `var_names='gene_ids'`
in `sc.read_10x_mtx`
```

[7]:
```
adata
```

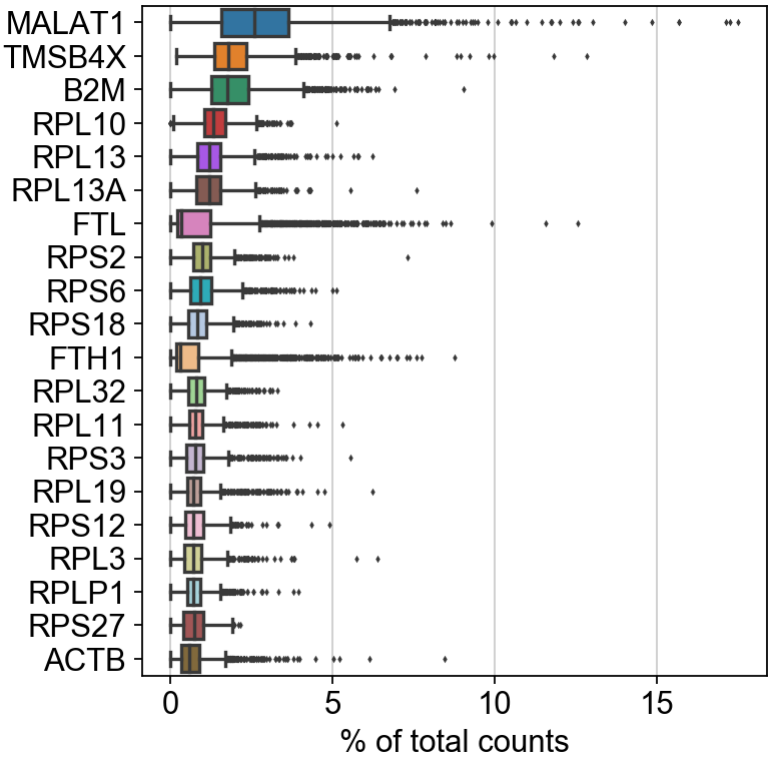
[7]:
```
AnnData object with n_obs × n_vars = 2700 × 32738
    var: 'gene_ids'
```

Preprocessing

Show those genes that yield the highest fraction of counts in each single cell, across all cells.

[8]:
```
sc.pl.highest_expr_genes(adata, n_top=20, )
```

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



Basic filtering:

[9]:
```
sc.pp.filter_cells(adata, min_genes=200)
sc.pp.filter_genes(adata, min_cells=3)
```

```
filtered out 19024 genes that are detected in less than 3 cells
```

Let's assemble some information about mitochondrial genes, which are important for quality control.

Citing from "Simple Single Cell" workflows (Lun, McCarthy & Marioni, 2017):

> High proportions are indicative of poor-quality cells (Islam et al. 2014; Ilicic et al. 2016), possibly because of loss of cytoplasmic RNA from perforated cells. The reasoning is that mitochondria are larger than individual transcript molecules and less likely to escape through tears in the cell membrane.

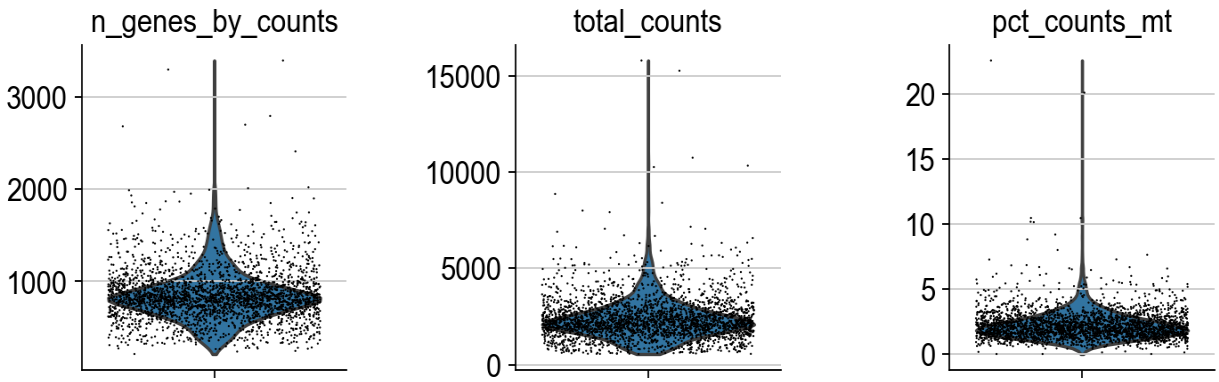
With `pp.calculate_qc_metrics`, we can compute many metrics very efficiently.

[10]:
```
adata.var['mt'] = adata.var_names.str.startswith('MT-')  # annotate the group of mitochondrial genes as 'mt'
sc.pp.calculate_qc_metrics(adata, qc_vars=['mt'], percent_top=None, log1p=False, inplace=True)
```

A violin plot of some of the computed quality measures:

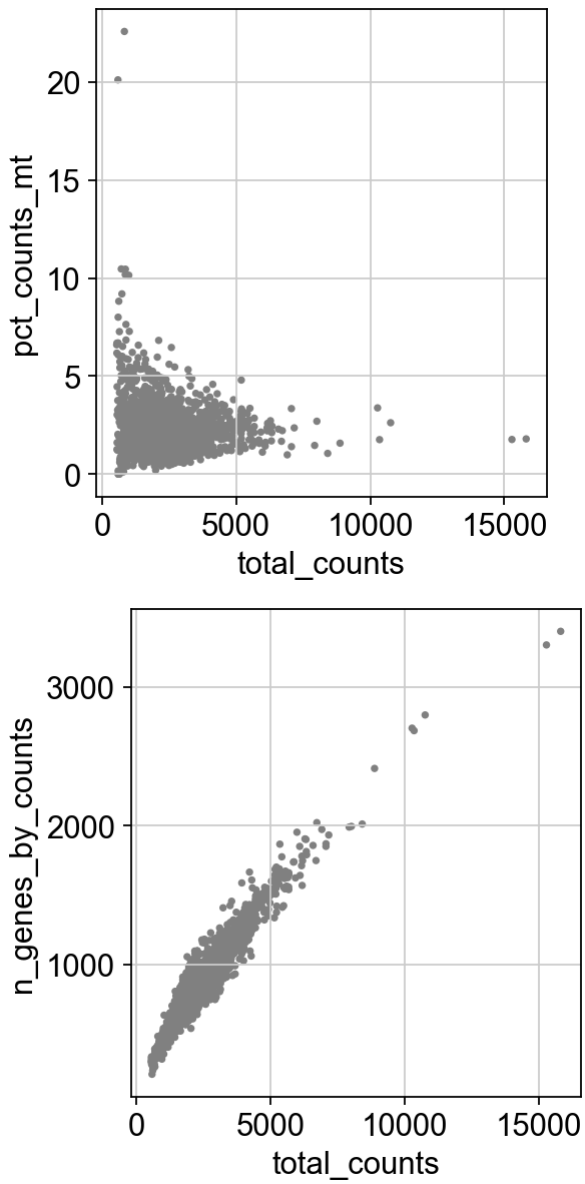
- the number of genes expressed in the count matrix
- the total counts per cell
- the percentage of counts in mitochondrial genes

[11]:
```
sc.pl.violin(adata, ['n_genes_by_counts', 'total_counts', 'pct_counts_mt'],
             jitter=0.4, multi_panel=True)
```



Remove cells that have too many mitochondrial genes expressed or too many total counts:

[12]:
```
sc.pl.scatter(adata, x='total_counts', y='pct_counts_mt')
sc.pl.scatter(adata, x='total_counts', y='n_genes_by_counts')
```



Actually do the filtering by slicing the `AnnData` object.

[13]:
```
adata = adata[adata.obs.n_genes_by_counts < 2500, :]
adata = adata[adata.obs.pct_counts_mt < 5, :]
```

Total-count normalize (library-size correct) the data matrix $\mathbf{X}$ to 10,000 reads per cell, so that counts become comparable among cells.

[14]:
```
sc.pp.normalize_total(adata, target_sum=1e4)
```

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

Logarithmize the data:

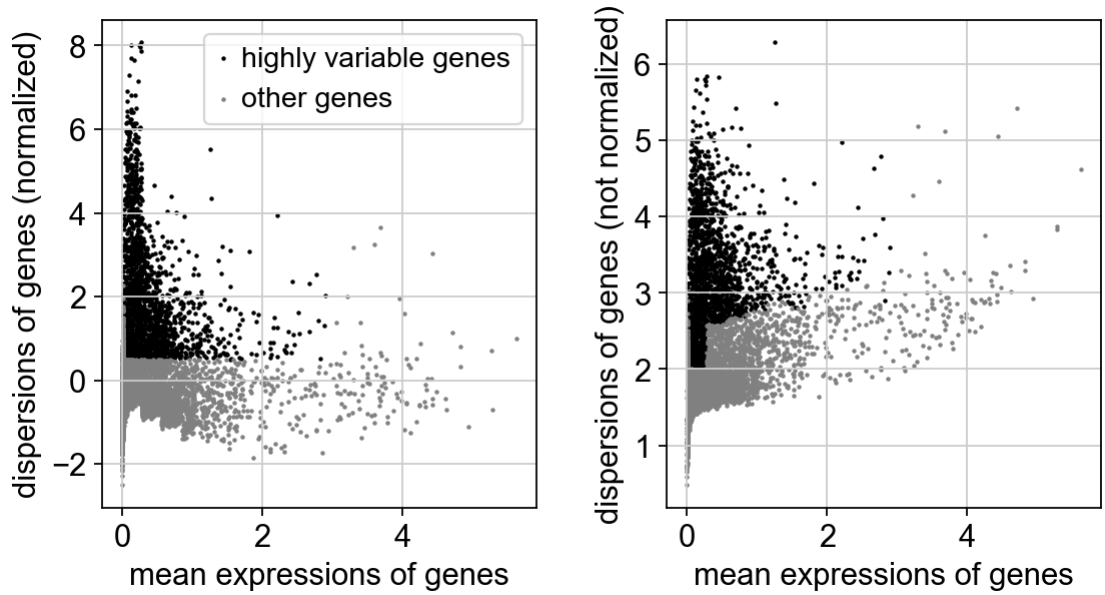
[15]:
```
sc.pp.log1p(adata)
```

Identify highly-variable genes.

[16]:
```
sc.pp.highly_variable_genes(adata, min_mean=0.0125, max_mean=3, min_disp=0.5)
```

```
extracting highly variable genes
    finished (0:00:00)
--> added
    'highly_variable', boolean vector (adata.var)
    'means', float vector (adata.var)
    'dispersions', float vector (adata.var)
    'dispersions_norm', float vector (adata.var)
```

[17]:
```
sc.pl.highly_variable_genes(adata)
```



Set the `.raw` attribute of the AnnData object to the normalized and logarithmized raw gene expression for later use in differential testing and visualizations of gene expression. This simply freezes the state of the AnnData object.

Note

You can get back an `AnnData` of the object in `.raw` by calling `.raw.to_adata()`.

[18]:
```
adata.raw = adata
```

Note

If you don't proceed below with correcting the data with `sc.pp.regress_out` and scaling it via `sc.pp.scale`, you can also get away without using `.raw` at all.

The result of the previous highly-variable-genes detection is stored as an annotation in `.var.highly_variable` and auto-detected by PCA and hence, `sc.pp.neighbors` and subsequent manifold/graph tools. In that case, the step *actually do the filtering* below is unnecessary, too.

Actually do the filtering

[19]:
```
adata = adata[:, adata.var.highly_variable]
```

Regress out effects of total counts per cell and the percentage of mitochondrial genes expressed. Scale the data to unit variance.

[20]:
```
sc.pp.regress_out(adata, ['total_counts', 'pct_counts_mt'])
```

```
regressing out ['total_counts', 'pct_counts_mt']
    sparse input is densified and may lead to high memory use
    finished (0:00:06)
```

Scale each gene to unit variance. Clip values exceeding standard deviation 10.

[21]:
```
sc.pp.scale(adata, max_value=10)
```

Principal component analysis

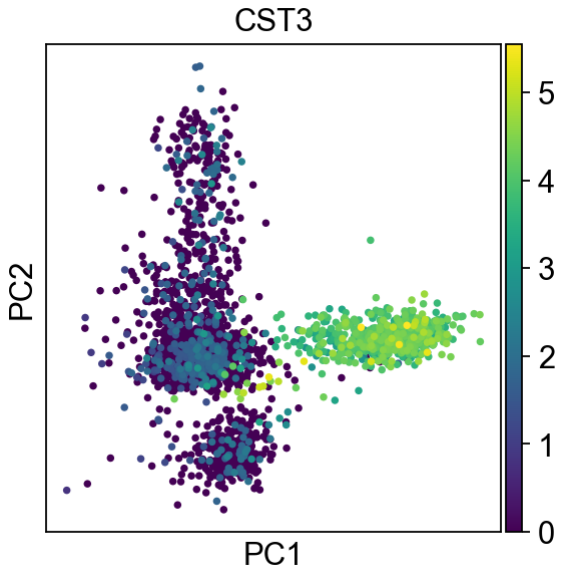
Reduce the dimensionality of the data by running principal component analysis (PCA), which reveals the main axes of variation and denoises the data.

[22]:
```
sc.tl.pca(adata, svd_solver='arpack')
```

```
computing PCA
    on highly variable genes
    with n_comps=50
    finished (0:00:00)
```

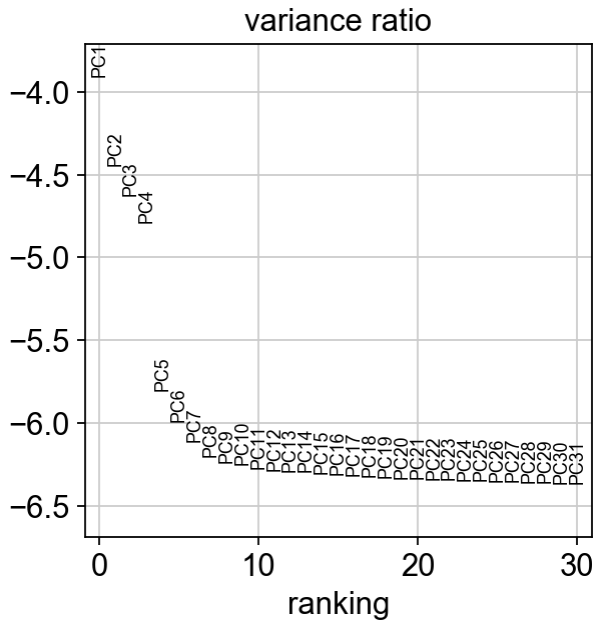
We can make a scatter plot in the PCA coordinates, but we will not use that later on.

[23]:
```
sc.pl.pca(adata, color='CST3')
```



Let us inspect the contribution of single PCs to the total variance in the data. This gives us information about how many PCs we should consider in order to compute the neighborhood relations of cells, e.g. used in the clustering function `sc.tl.louvain()` or tSNE `sc.tl.tsne()`. In our experience, often a rough estimate of the number of PCs does fine.

[24]:
```
sc.pl.pca_variance_ratio(adata, log=True)
```



Save the result.

[25]:
```
adata.write(results_file)
```

[26]:
```
adata
```

[26]:
```
AnnData object with n_obs × n_vars = 2638 × 1838
    obs: 'n_genes', 'n_genes_by_counts', 'total_counts', 'total_counts_mt',
'pct_counts_mt'
    var: 'gene_ids', 'n_cells', 'mt', 'n_cells_by_counts', 'mean_counts',
'pct_dropout_by_counts', 'total_counts', 'highly_variable', 'means', 'dispersions',
'dispersions_norm', 'mean', 'std'
    uns: 'log1p', 'hvg', 'pca'
    obsm: 'X_pca'
    varm: 'PCs'
```

Computing the neighborhood graph

Let us compute the neighborhood graph of cells using the PCA representation of the data matrix. You might simply use default values here. For the sake of reproducing Seurat's results, let's take the following values.

[27]:
```
sc.pp.neighbors(adata, n_neighbors=10, n_pcs=40)
```

```
computing neighbors
    using 'X_pca' with n_pcs = 40
    finished: added to `.uns['neighbors']`
    `.obsp['distances']`, distances for each pair of neighbors
    `.obsp['connectivities']`, weighted adjacency matrix (0:00:01)
```

Embedding the neighborhood graph

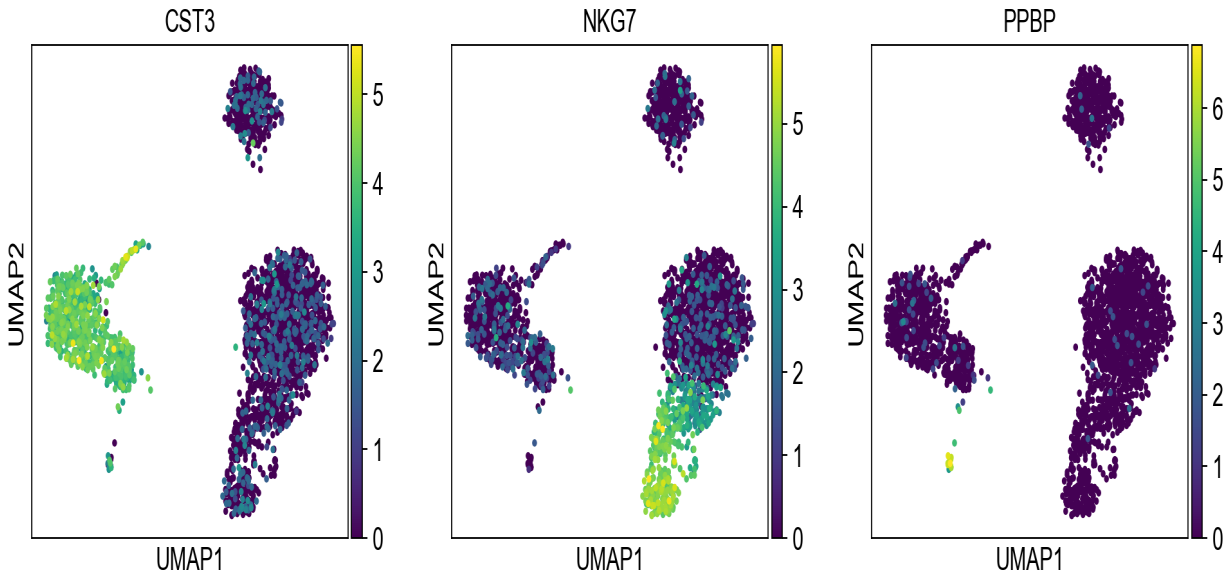
We suggest embedding the graph in two dimensions using UMAP (McInnes et al., 2018), see below. It is potentially more faithful to the global connectivity of the manifold than tSNE, i.e., it better preserves trajectories. In some ocassions, you might still observe disconnected clusters and similar connectivity violations. They can usually be remedied by running:

```
sc.tl.paga(adata)
sc.pl.paga(adata, plot=False)  # remove `plot=False` if you want to see the coarse-grained graph
sc.tl.umap(adata, init_pos='paga')
```

[28]:
```
sc.tl.umap(adata)
```

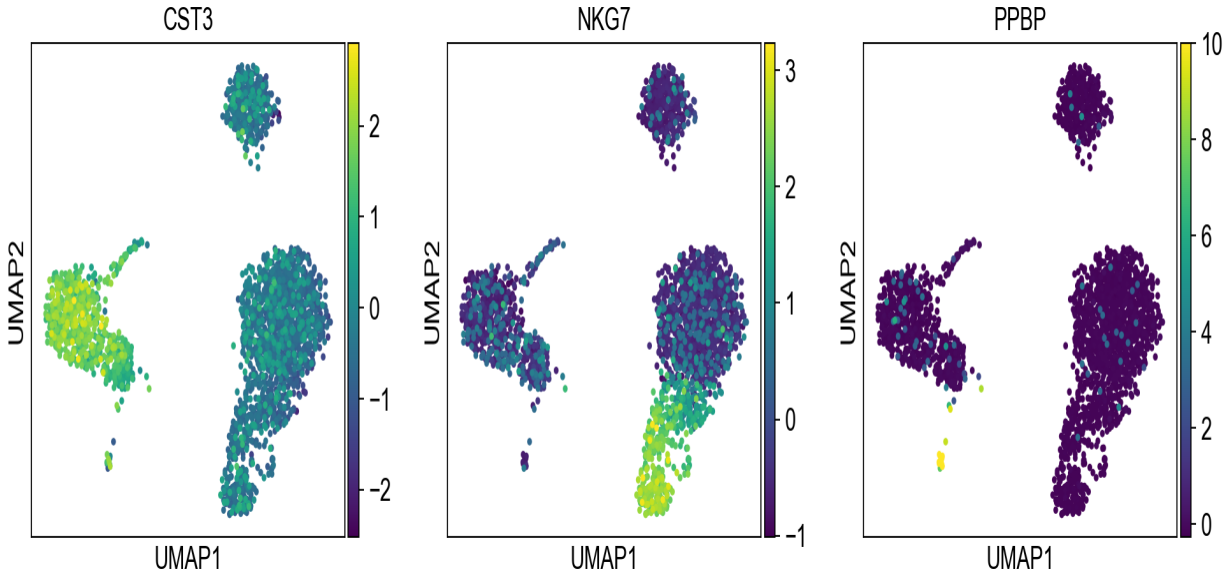
```
computing UMAP
    finished: added
    'X_umap', UMAP coordinates (adata.obsm) (0:00:03)
```

[29]:
```
sc.pl.umap(adata, color=['CST3', 'NKG7', 'PPBP'])
```



As we set the `.raw` attribute of `adata`, the previous plots showed the "raw" (normalized, logarithmized, but uncorrected) gene expression. You can also plot the scaled and corrected gene expression by explicitly stating that you don't want to use `.raw`.

[30]:
```
sc.pl.umap(adata, color=['CST3', 'NKG7', 'PPBP'], use_raw=False)
```



Clustering the neighborhood graph

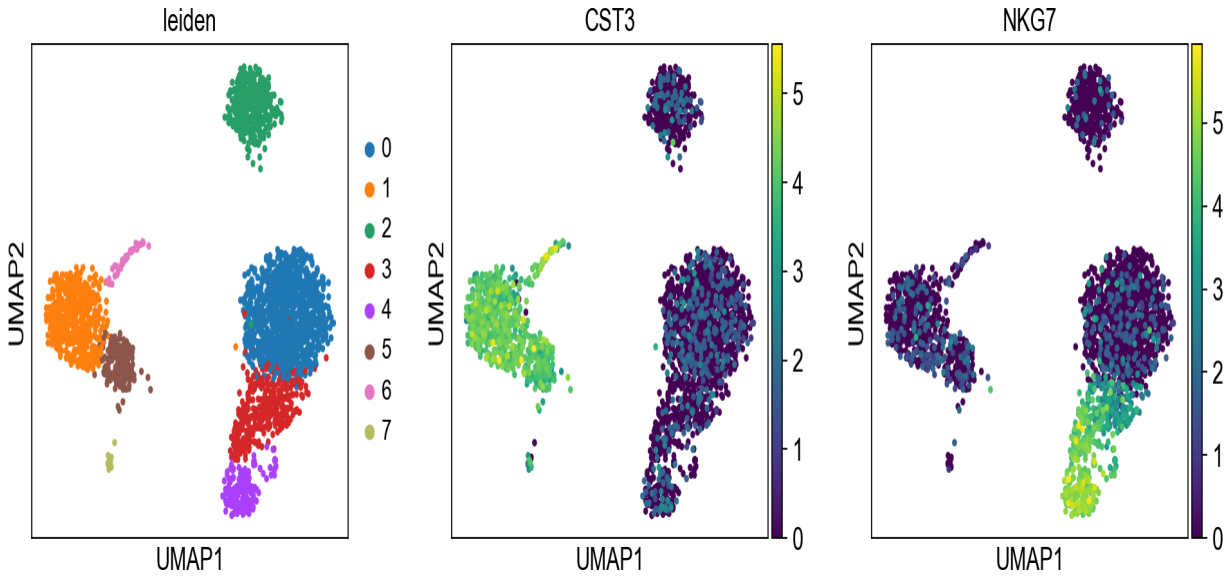
As with Seurat and many other frameworks, we recommend the Leiden graph-clustering method (community detection based on optimizing modularity) by Traag *et al.* (2018). Note that Leiden clustering directly clusters the neighborhood graph of cells, which we already computed in the previous section.

[31]:
```
sc.tl.leiden(adata)
```

```
running Leiden clustering
    finished: found 8 clusters and added
    'leiden', the cluster labels (adata.obs, categorical) (0:00:00)
```

Plot the clusters, which agree quite well with the result of Seurat.

[32]:
```
sc.pl.umap(adata, color=['leiden', 'CST3', 'NKG7'])
```



Save the result.

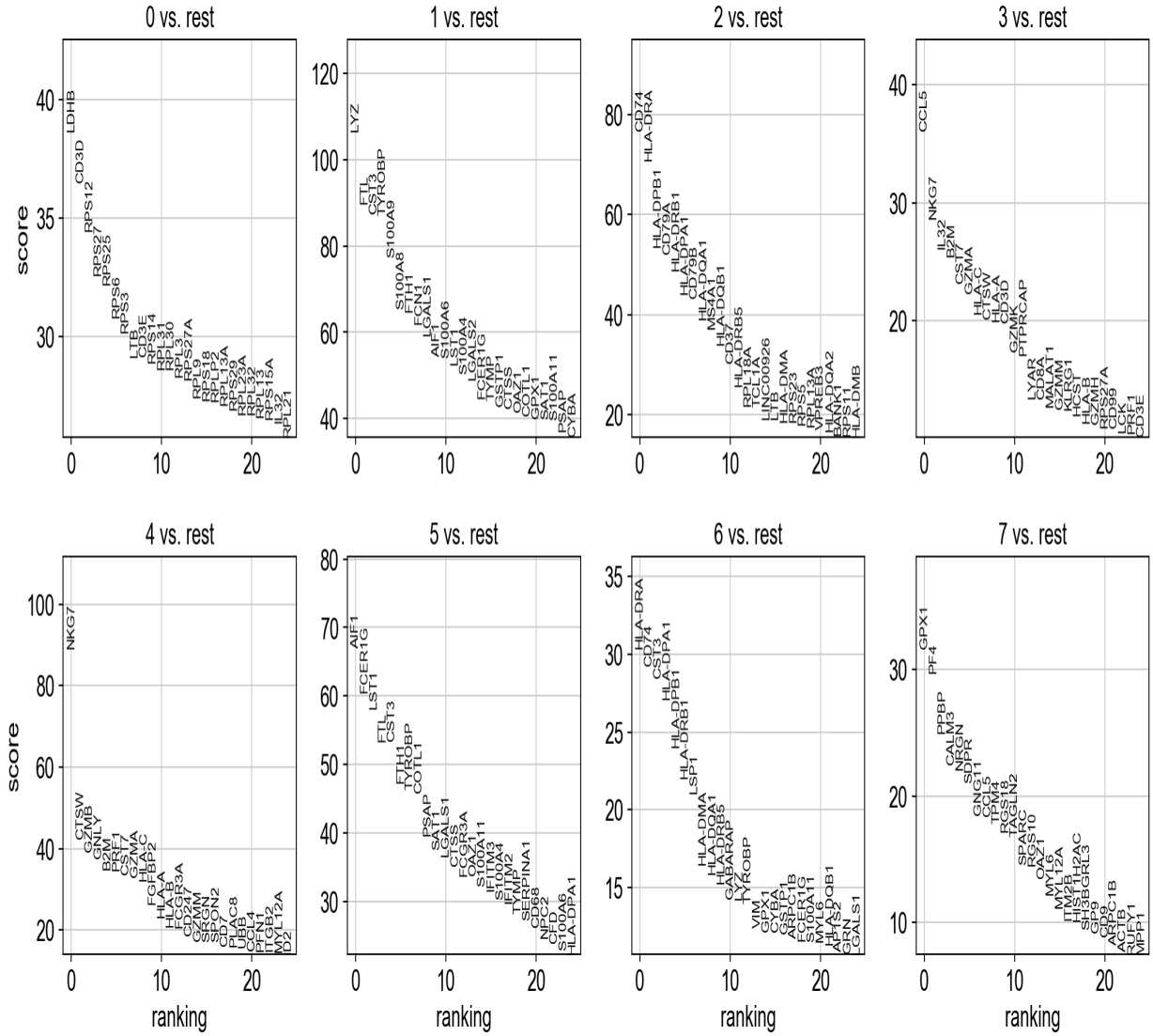
[33]:
```
adata.write(results_file)
```

Finding marker genes

Let us compute a ranking for the highly differential genes in each cluster. For this, by default, the `.raw` attribute of AnnData is used in case it has been initialized before. The simplest and fastest method to do so is the t-test.

[34]:
```
sc.tl.rank_genes_groups(adata, 'leiden', method='t-test')
sc.pl.rank_genes_groups(adata, n_genes=25, sharey=False)
```

```
ranking genes
    finished: added to `.uns['rank_genes_groups']`
    'names', sorted np.recarray to be indexed by group ids
    'scores', sorted np.recarray to be indexed by group ids
    'logfoldchanges', sorted np.recarray to be indexed by group ids
    'pvals', sorted np.recarray to be indexed by group ids
    'pvals_adj', sorted np.recarray to be indexed by group ids (0:00:00)
```

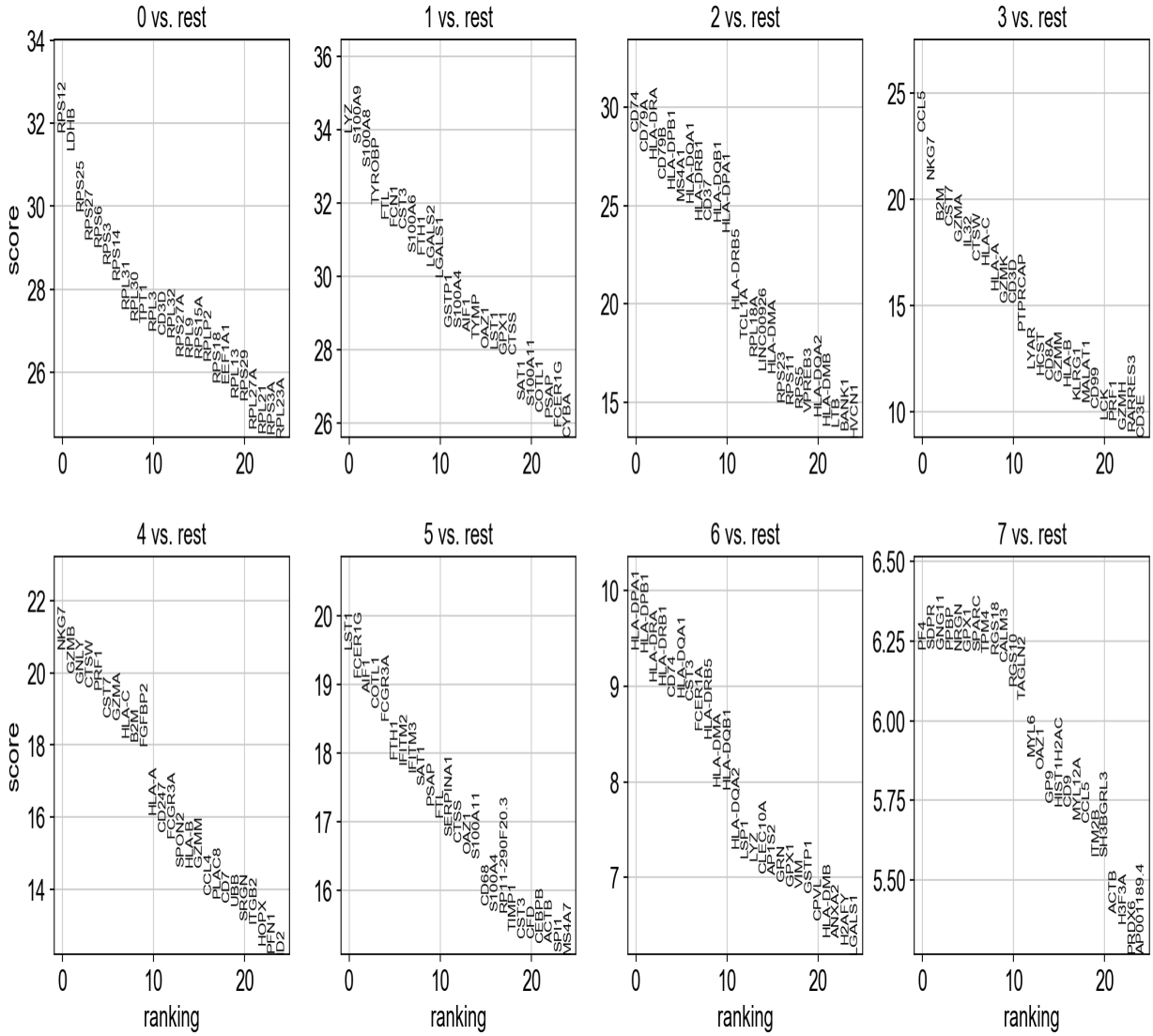


[35]:
```
sc.settings.verbosity = 2  # reduce the verbosity
```

The result of a Wilcoxon rank-sum (Mann-Whitney-U) test is very similar. We recommend using the latter in publications, see e.g., Sonison & Robinson (2018). You might also consider much more powerful differential testing packages like MAST, limma, DESeq2 and, for python, the recent diffxpy.

[36]:
```
sc.tl.rank_genes_groups(adata, 'leiden', method='wilcoxon')
sc.pl.rank_genes_groups(adata, n_genes=25, sharey=False)
```

```
ranking genes
    finished (0:00:02)
```



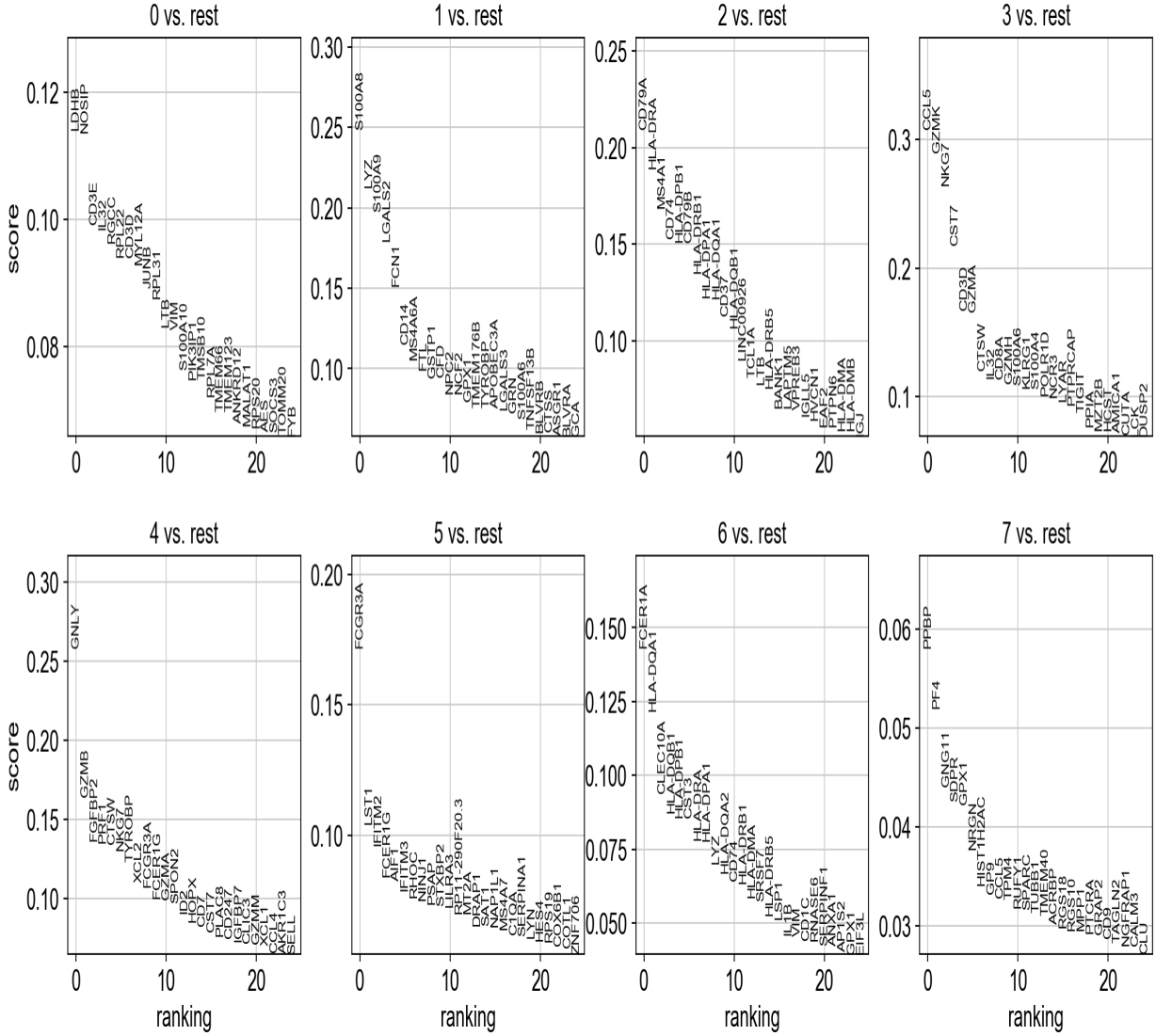
Save the result.

[37]:
```
adata.write(results_file)
```

As an alternative, let us rank genes using logistic regression. For instance, this has been suggested by Natranos et al. (2018). The essential difference is that here, we use a multi-variate appraoch whereas conventional differential tests are uni-variate. Clark et al. (2014) has more details.

[38]:
```
sc.tl.rank_genes_groups(adata, 'leiden', method='logreg')
sc.pl.rank_genes_groups(adata, n_genes=25, sharey=False)
```

```
ranking genes
    finished (0:00:04)
```



With the exceptions of *IL7R*, which is only found by the t-test and *FCER1A*, which is only found by the other two appraoches, all marker genes are recovered in all approaches.

Louvain Group	Markers	Cell Type
0	IL7R	CD4 T cells
1	CD14, LYZ	CD14+ Monocytes
2	MS4A1	B cells
3	CD8A	CD8 T cells
4	GNLY, NKG7	NK cells
5	FCGR3A, MS4A7	FCGR3A+ Monocytes
6	FCER1A, CST3	Dendritic Cells
7	PPBP	Megakaryocytes

Let us also define a list of marker genes for later reference.

[39]:
```
marker_genes = ['IL7R', 'CD79A', 'MS4A1', 'CD8A', 'CD8B', 'LYZ', 'CD14',
                'LGALS3', 'S100A8', 'GNLY', 'NKG7', 'KLRB1',
                'FCGR3A', 'MS4A7', 'FCER1A', 'CST3', 'PPBP']
```

Reload the object that has been save with the Wilcoxon Rank-Sum test result.

[40]:
```
adata = sc.read(results_file)
```

Show the 10 top ranked genes per cluster 0, 1, ..., 7 in a dataframe.

[41]:
```
pd.DataFrame(adata.uns['rank_genes_groups']['names']).head(5)
```

[41]:

	0	1	2	3	4	5	6	7
0	RPS12	LYZ	CD74	CCL5	NKG7	LST1	HLA-DPA1	PF4
1	LDHB	S100A9	CD79A	NKG7	GZMB	FCER1G	HLA-DPB1	SDPR
2	RPS25	S100A8	HLA-DRA	B2M	GNLY	AIF1	HLA-DRA	GNG11
3	RPS27	TYROBP	CD79B	CST7	CTSW	COTL1	HLA-DRB1	PPBP
4	RPS6	FTL	HLA-DPB1	GZMA	PRF1	FCGR3A	CD74	NRGN

Get a table with the scores and groups.

[42]:
```
result = adata.uns['rank_genes_groups']
groups = result['names'].dtype.names
pd.DataFrame(
    {group + '_' + key[:1]: result[key][group]
    for group in groups for key in ['names', 'pvals']}).head(5)
```

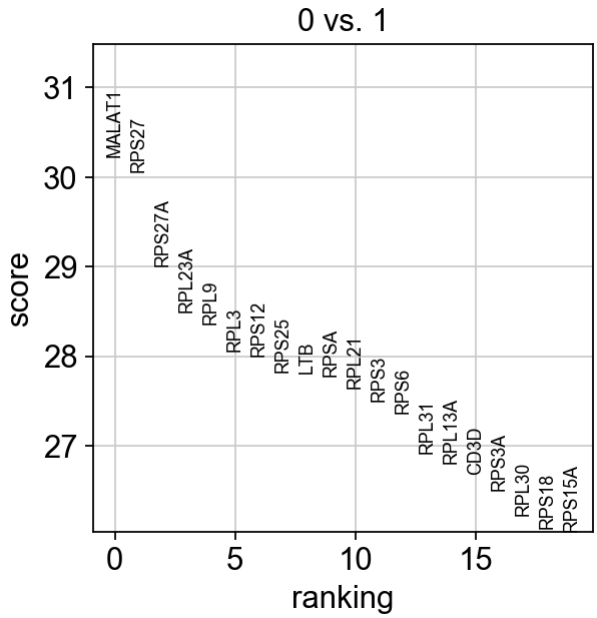
[42]:

	0_n	0_p	1_n	1_p	2_n	2_p	3_n	3_p	4_n	
0	RPS12	3.642456e-222	LYZ	1.007060e-252	CD74	3.043536e-182	CCL5	3.896273e-119	NKG7	4.6890
1	LDHB	3.242464e-216	S100A9	3.664292e-248	CD79A	6.860832e-170	NKG7	1.170992e-97	GZMB	2.3813
2	RPS25	1.394016e-196	S100A8	9.457377e-239	HLA-DRA	8.398068e-166	B2M	3.032705e-81	GNLY	9.3221
3	RPS27	9.718451e-188	TYROBP	2.209430e-224	CD79B	1.171444e-153	CST7	1.129293e-78	CTSW	1.0350
4	RPS6	1.771786e-185	FTL	3.910903e-219	HLA-DPB1	6.167786e-148	GZMA	4.263559e-73	PRF1	3.3641

Compare to a single cluster:

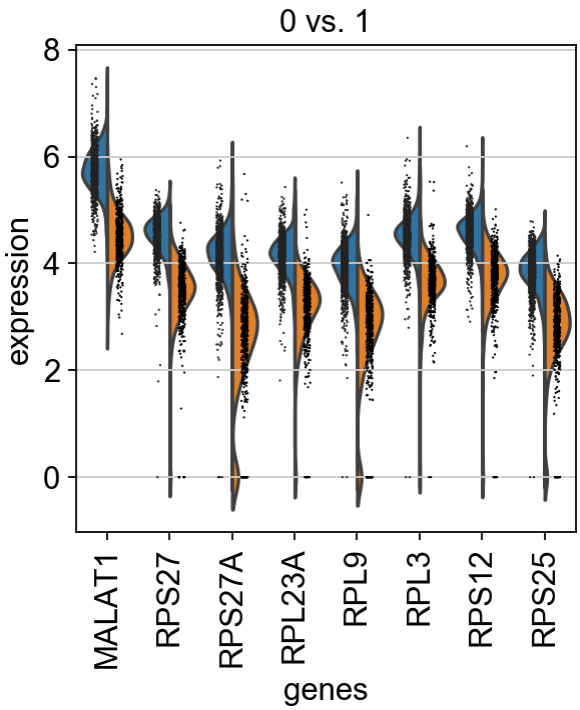
[43]:
```
sc.tl.rank_genes_groups(adata, 'leiden', groups=['0'], reference='1',
method='wilcoxon')
sc.pl.rank_genes_groups(adata, groups=['0'], n_genes=20)
```

```
ranking genes
    finished (0:00:01)
```



If we want a more detailed view for a certain group, use `sc.pl.rank_genes_groups_violin`.

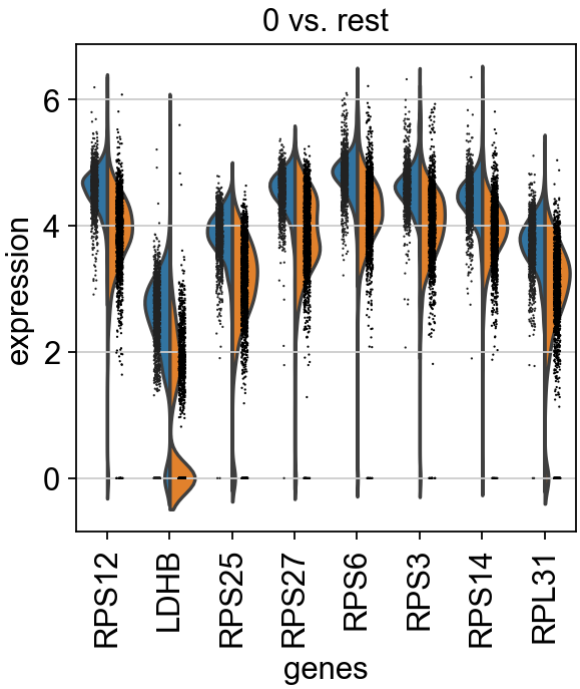
[44]:
```
sc.pl.rank_genes_groups_violin(adata, groups='0', n_genes=8)
```



Reload the object with the computed differential expression (i.e. DE via a comparison with the rest of the groups):

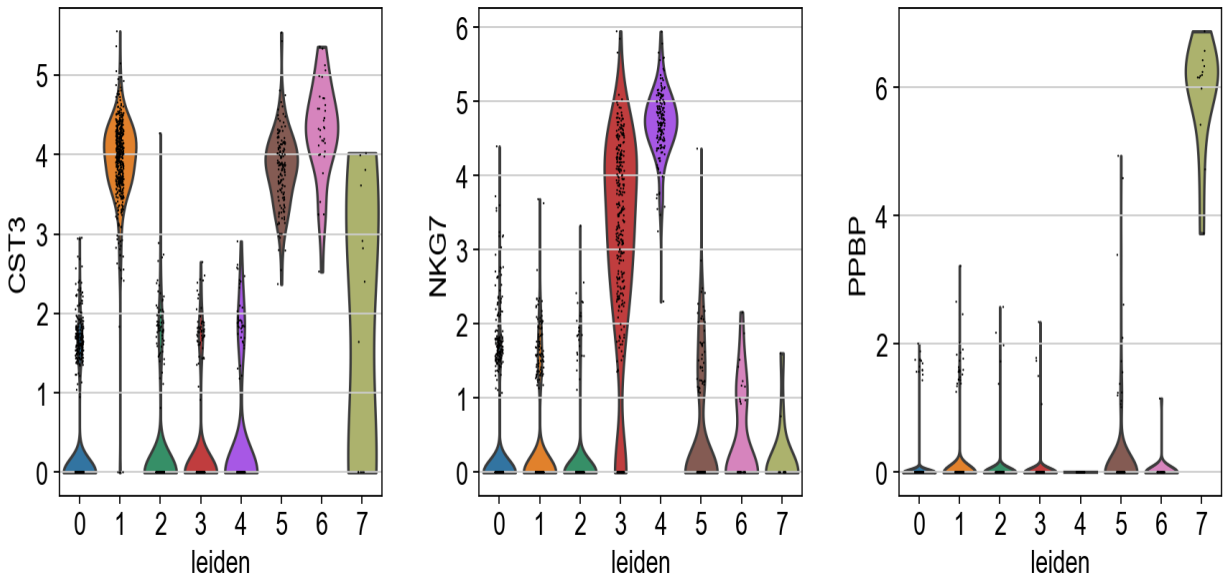
[45]:
```
adata = sc.read(results_file)
```

[46]:
```
sc.pl.rank_genes_groups_violin(adata, groups='0', n_genes=8)
```



If you want to compare a certain gene across groups, use the following.

[47]:
```
sc.pl.violin(adata, ['CST3', 'NKG7', 'PPBP'], groupby='leiden')
```

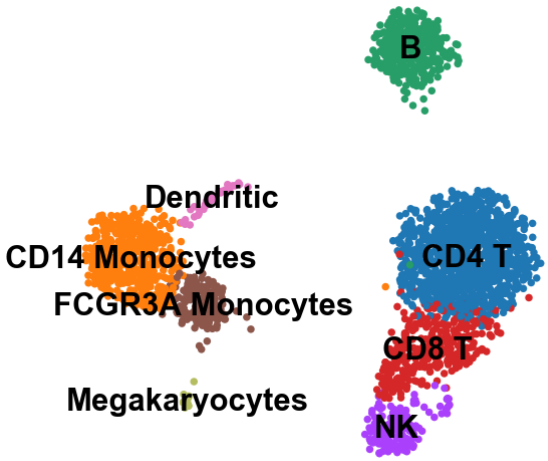


Actually mark the cell types.

[48]:
```
new_cluster_names = [
    'CD4 T', 'CD14 Monocytes',
    'B', 'CD8 T',
    'NK', 'FCGR3A Monocytes',
    'Dendritic', 'Megakaryocytes']
adata.rename_categories('leiden', new_cluster_names)
```

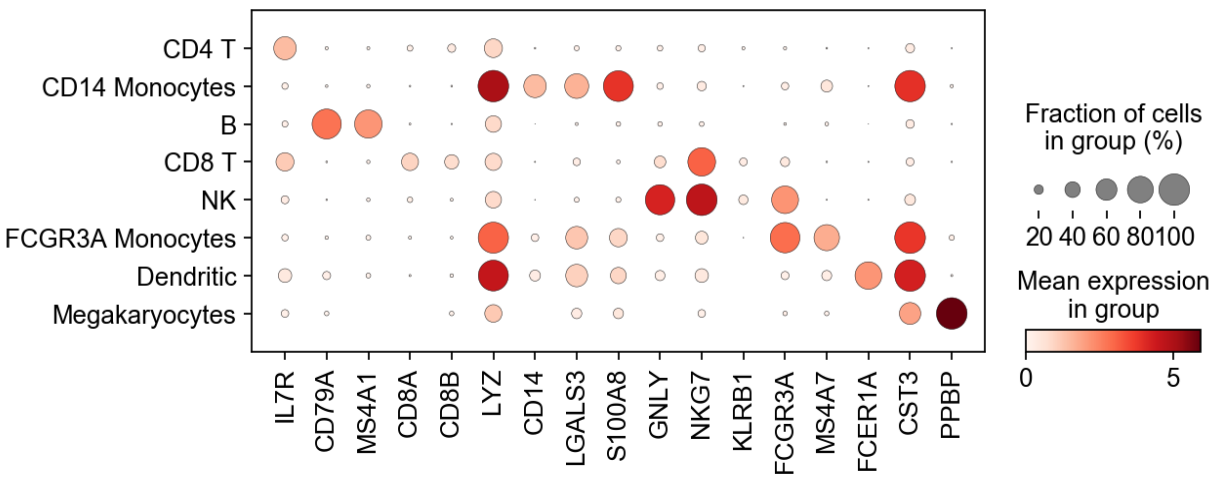
[49]:
```
sc.pl.umap(adata, color='leiden', legend_loc='on data', title='', frameon=False,
save='.pdf')
```

```
WARNING: saving figure to file figures/umap.pdf
```



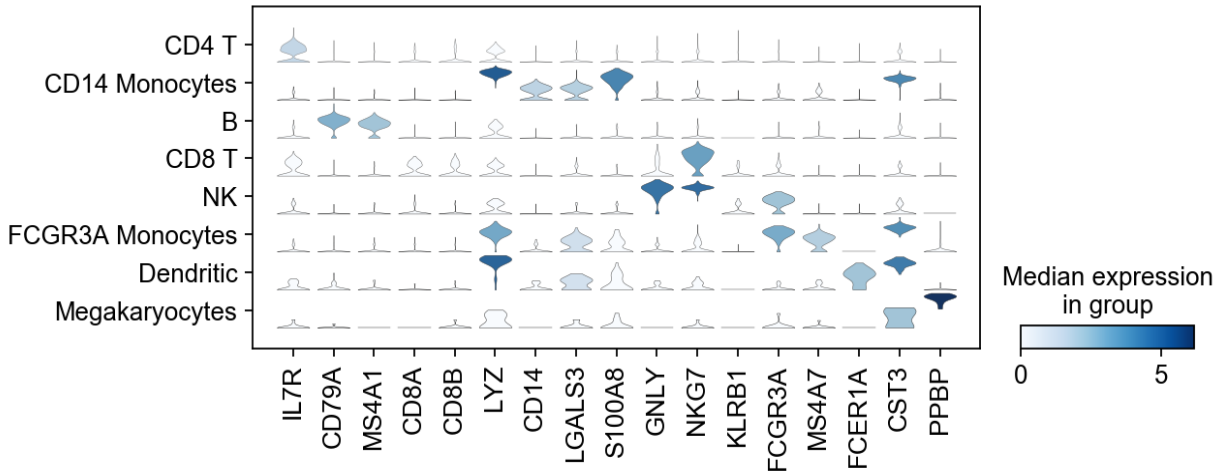
Now that we annotated the cell types, let us visualize the marker genes.

[50]:
```
sc.pl.dotplot(adata, marker_genes, groupby='leiden');
```



There is also a very compact violin plot.

[51]:
```
sc.pl.stacked_violin(adata, marker_genes, groupby='leiden', rotation=90);
```



During the course of this analysis, the AnnData accumlated the following annotations.

[52]:
```
adata
```

[52]:
```
AnnData object with n_obs × n_vars = 2638 × 1838
    obs: 'n_genes', 'n_genes_by_counts', 'total_counts', 'total_counts_mt',
'pct_counts_mt', 'leiden'
    var: 'gene_ids', 'n_cells', 'mt', 'n_cells_by_counts', 'mean_counts',
'pct_dropout_by_counts', 'total_counts', 'highly_variable', 'means', 'dispersions',
'dispersions_norm', 'mean', 'std'
    uns: 'hvg', 'leiden', 'leiden_colors', 'neighbors', 'pca', 'rank_genes_groups',
'umap'
    obsm: 'X_pca', 'X_umap'
    varm: 'PCs'
    obsp: 'connectivities', 'distances'
```

[53]:
```
adata.write(results_file, compression='gzip')  # `compression='gzip'` saves disk
space, but slows down writing and subsequent reading
```

Get a rough overview of the file using `h5ls`, which has many options - for more details see here. The file format might still be subject to further optimization in the future. All reading functions will remain backwards-compatible, though.

If you want to share this file with people who merely want to use it for visualization, a simple way to reduce the file size is by removing the dense scaled and corrected data matrix. The file still contains the raw data used in the visualizations in `adata.raw`.

[54]:
```
adata.raw.to_adata().write('./write/pbmc3k_withoutX.h5ad')
```

If you want to export to “csv”, you have the following options:

[55]:
```
# Export single fields of the annotation of observations
# adata.obs[['n_counts', 'louvain_groups']].to_csv(
#     './write/pbmc3k_corrected_louvain_groups.csv')

# Export single columns of the multidimensional annotation
# adata.obsm.to_df()[['X_pca1', 'X_pca2']].to_csv(
#     './write/pbmc3k_corrected_X_pca.csv')

# Or export everything except the data using `.write_csvs`.
# Set `skip_data=False` if you also want to export the data.
# adata.write_csvs(results_file[:-5], )
```