

ComplexHeatmap Complete Reference

Zuguang Gu

last revised on 2022-02-23

Contents

About	7
1 Introduction	9
1.1 General design	9
1.2 A brief description of following chapters	13
2 A Single Heatmap	15
2.1 Colors	17
2.2 Titles	28
2.3 Clustering	36
2.4 Set row and column orders	54
2.5 Seriation	56
2.6 Dimension labels	59
2.7 Heatmap split	69
2.8 Heatmap as raster image	100
2.9 Customize the heatmap body	104
2.10 Size of the heatmap	120
2.11 Plot the heatmap	123
2.12 Extract orders and dendrograms	123
2.13 Subset a heatmap	125
3 Heatmap Annotations	129
3.1 Simple annotation	132
3.2 Simple annotation as an annotation function	137
3.3 Empty annotation	141
3.4 Block annotation	144
3.5 Image annotation	153
3.6 Points annotation	155
3.7 Line annotation	158
3.8 Barplot annotation	159
3.9 Boxplot annotation	162
3.10 Histogram annotation	164
3.11 Density annotation	166
3.12 Joyplot annotation	171

3.13	Horizon chart annotation	173
3.14	Text annotation	178
3.15	Numeric labels annotation	184
3.16	Completely customized annotation	190
3.17	Mark annotation	194
3.18	Zoom/link annotation	195
3.19	Text box annotation	198
3.20	Summary annotation	211
3.21	Multiple annotations	214
3.22	Utility functions	225
3.23	Implement new annotation functions	227
4	A List of Heatmaps	235
4.1	Titles	237
4.2	Size of heatmaps	238
4.3	Gap between heatmaps	240
4.4	Automatic adjustment to the main heatmap	241
4.5	Control main heatmap in draw() function	244
4.6	Annotations as components are adjusted	245
4.7	Concatenate with annotations	249
4.8	Concatenate only the annotations	253
4.9	Vertical concatenation	255
4.10	Subset the heatmap list	260
4.11	Plot the heatmap list	261
4.12	Get orders and dendrograms	262
4.13	Change parameters globally	263
4.14	Adjust blank space caused by annotations	266
4.15	Manually increase space around the plot	268
5	Legends	269
5.1	Continuous legends	269
5.2	Discrete legends	281
5.3	A list of legends	288
5.4	Heatmap and annotation legends	292
5.5	Add customized legends	300
5.6	The side of legends	302
6	Heatmap Decoration	305
6.1	Decoration functions	307
6.2	Examples	309
7	OncoPrint	313
7.1	General settings	313
7.2	Apply to cBioPortal dataset	329
8	UpSet plot	343

8.1	Input data	343
8.2	Mode	345
8.3	Make the combination matrix	346
8.4	Utility functions	352
8.5	Make the plot	357
8.6	UpSet plots as heatmaps	365
8.7	Example with the movies dataset	376
8.8	Example with the genomic regions	389
9	Interactive ComplexHeatmap	393
10	Integrate with other packages	395
10.1	pheatmap	395
10.2	cowplot	415
10.3	gridtext	418
11	Other High-level Plots	427
11.1	Density heatmap	427
11.2	Stacked summary plot	434
12	Three-dimensional ComplexHeatmap	441
12.1	Motivation	441
12.2	Implementation of 3D heatmap	444
13	Genome-level heatmap	451
14	More Examples	463
14.1	Add more information for gene expression matrix	463
14.2	The measles vaccine heatmap	465
14.3	Visualize Cell Heterogeneity from Single Cell RNASeq	466
14.4	Correlations between methylation, expression and other genomic features	470
14.5	Visualize Methylation Profile with Complex Annotations	472
14.6	Add multiple boxplots for single row	478
15	Other Tricks	481
15.1	Set the same cell size for different heatmaps with different dimensions	481

About

This is the documentation of the **ComplexHeatmap** package. Examples in the book are generated under version 2.11.1.

You can get a stable Bioconductor version from <http://bioconductor.org/packages/release/bioc/html/ComplexHeatmap.html>, but the most up-to-date version is always on Github and you can install it by:

```
library(devtools)
install_github("jokergoo/ComplexHeatmap")
```

The development branch on Bioconductor is basically synchronized to the Github repository.

The **ComplexHeatmap** package is inspired by the **pheatmap** package. You can find many arguments in **ComplexHeatmap** have the same names as in **pheatmap**. Also you can find this old package that I tried to develop by modifying **pheatmap**.

Please note, this documentation is not completely compatible with older versions (< 1.99.0, before Oct, 2018), but the major functionality keeps the same.

If you use **ComplexHeatmap** in your publications, I am appreciated if you can cite:

Gu, Z. (2016) Complex heatmaps reveal patterns and correlations in multidimensional genomic data. DOI: 10.1093/bioinformatics/btw313

Session info:

```
sessionInfo()

## R version 4.1.2 (2021-11-01)
## Platform: x86_64-apple-darwin17.0 (64-bit)
## Running under: macOS Big Sur 10.16
##
## Matrix products: default
## BLAS:    /Library/Frameworks/R.framework/Versions/4.1/Resources/lib/libRblas.0.dylib
```

```
## LAPACK: /Library/Frameworks/R.framework/Versions/4.1/Resources/lib/libRlapack.dylib
##
## locale:
## [1] C/UTF-8/C/C/C/C
##
## attached base packages:
## [1] grid      stats     graphics grDevices utils      datasets  methods
## [8] base
##
## other attached packages:
## [1] dendsort_0.3.4      dendextend_1.15.2    circlize_0.4.14
## [4] ComplexHeatmap_2.11.1
##
## loaded via a namespace (and not attached):
## [1] shape_1.4.6          GetoptLong_1.0.5   tidyselect_1.1.1
## [4] xfun_0.29            purrr_0.3.4       colorspace_2.0-2
## [7] vctrs_0.3.8          generics_0.1.1    htmltools_0.5.2
## [10] stats4_4.1.2         viridisLite_0.4.0 yaml_2.2.1
## [13] utf8_1.2.2          rlang_0.4.12     pillar_1.6.4
## [16] glue_1.6.0           DBI_1.1.2        BiocGenerics_0.40.0
## [19] RColorBrewer_1.1-2  matrixStats_0.61.0 foreach_1.5.1
## [22] lifecycle_1.0.1     stringr_1.4.0     munsell_0.5.0
## [25] gtable_0.3.0         GlobalOptions_0.1.2 codetools_0.2-18
## [28] evaluate_0.14        knitr_1.37       IRanges_2.28.0
## [31] fastmap_1.1.0        doParallel_1.0.16 parallel_4.1.2
## [34] fansi_1.0.2          scales_1.1.1     S4Vectors_0.32.3
## [37] gridExtra_2.3        rjson_0.2.21     ggplot2_3.3.5
## [40] png_0.1-7            digest_0.6.29    stringi_1.7.6
## [43] bookdown_0.24        dplyr_1.0.7      clue_0.3-60
## [46] tools_4.1.2          magrittr_2.0.1    tibble_3.1.6
## [49] cluster_2.1.2        crayon_1.4.2     pkgconfig_2.0.3
## [52] ellipsis_0.3.2       assertthat_0.2.1 rmarkdown_2.11
## [55] iterators_1.0.13     viridis_0.6.2    R6_2.5.1
## [58] compiler_4.1.2
```

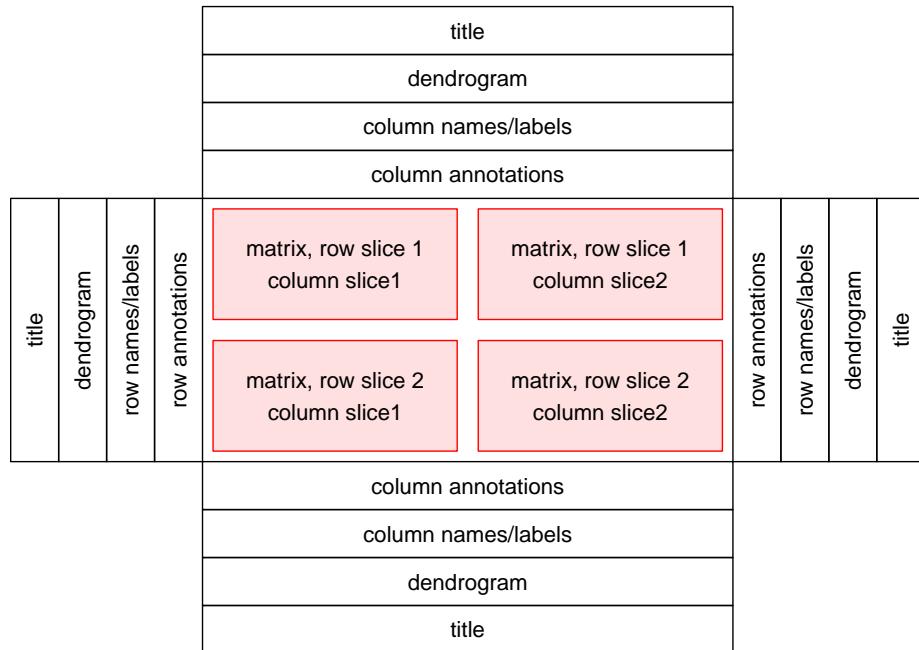
Chapter 1

Introduction

Complex heatmaps are efficient to visualize associations between different sources of data sets and reveal potential patterns. Here the **ComplexHeatmap** package provides a highly flexible way to arrange multiple heatmaps and supports self-defined annotation graphics.

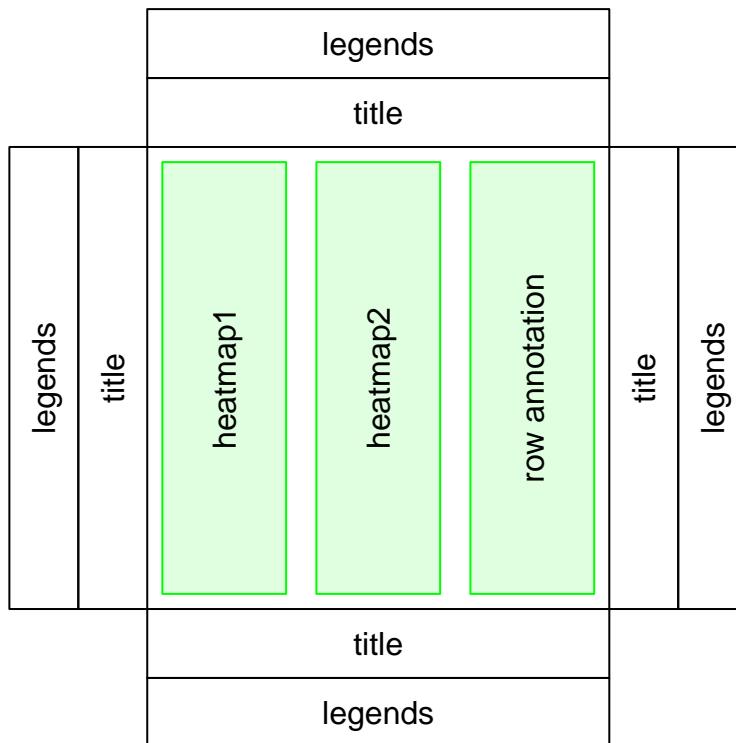
1.1 General design

A single heatmap is composed by the heatmap body and the heatmap components. The heatmap body can be split by rows and columns. The heatmap components are titles, dendograms, row/column names/labels and heatmap annotations, which are put on the four sides of the heatmap body. The heatmap components are reordered or split according to the heatmap body.

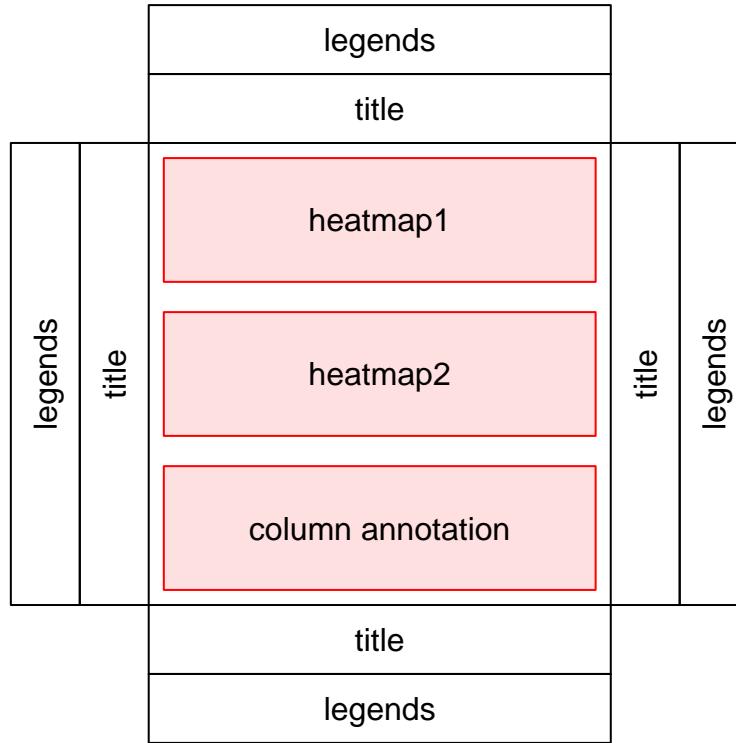


A heatmap list is concatenation of a list of heatmaps and heatmap annotations. Surrounding the heatmap list, there are global-level titles and legends.

One important thing for the heatmap list is that rows for all heatmaps and annotations are all adjusted so that the same row in all heatmaps and annotations corresponds to a same feature.



The heatmaps and annotations (now it is column annotation) can also be arranged vertically.



And the heatmap list can also be split by rows and by columns.

The **ComplexHeatmap** package is implemented in an object-oriented way. To describe a heatmap list, there are following classes:

- **Heatmap** class: a single heatmap containing heatmap body, row/column names, titles, dendograms and row/column annotations.
- **HeatmapList** class: a list of heatmaps and heatmap annotations.
- **HeatmapAnnotation** class: defines a list of row annotations and column annotations. The heatmap annotations can be components of heatmap, also they can be independent as heatmaps.

There are also several internal classes:

- **SingleAnnotation** class: defines a single row annotation or column annotation. The **HeatmapAnnotation** object contains a list of **SingleAnnotation** objects.
- **ColorMapping** class: mapping from values to colors. The color mappings of the main matrix and the annotations are controlled by **ColorMapping** class.
- **AnnotationFunction** class: constructs user-defined annotations. This is the base of creating user-defined annotation graphics.

ComplexHeatmap is implemented under **grid** system, so users need to know

basic `grid` functionality to get full use of the package.

1.2 A brief description of following chapters

- **A Single Heatmap**

This chapter describes the configurations of a single heatmap.

- **Heatmap Annotations**

This chapter describes the concept of the heatmap annotation and demonstrates how to make simple annotations as well as complex annotations. Also, the chapter explains the difference between column annotations and row annotations.

- **A List of Heatmaps**

This chapter describes how to concatenate a list of heatmaps and annotations and how adjustment is applied to keep the correspondence of the heatmaps.

- **Legends**

This chapter describes how to configurate the heatmap legends and annotation legends, also how to create self-defined legends.

- **Heatmap Decoration**

This chapter describes methods to add more self-defined graphics to the heatmaps after the heatmaps are generated.

- **OncoPrint**

This chapter describes how to make oncoPrints and how to integrate other functionalities from **ComplexHeatmap** to oncoPrints.

- **UpSet plot**

This chapter describes how to make enhanced UpSet plots.

- **Other High-level Plots**

This chapter describes functions implemented in **ComplexHeatmap** for specific use, e.g. visualizing distributions.

- **Integrate with other packages**

This chapter describes how other packages are integrated with **ComplexHeatmap**. Currently, we demonstrate the two packages of `gridtext` and `pheatmap`.

- **Interactive heatmap**

This chapter describes how to make heatmaps interactive.

- **More Examples**

More simulated and real-world examples are demonstrated in this chapter.

Chapter 2

A Single Heatmap

A single heatmap is the most used approach for visualizing data. Although “the shining point” of the **ComplexHeatmap** package is that it can visualize a list of heatmaps in parallel, however, as the basic unit of the heatmap list, it is still very important to have the single heatmap well configured.

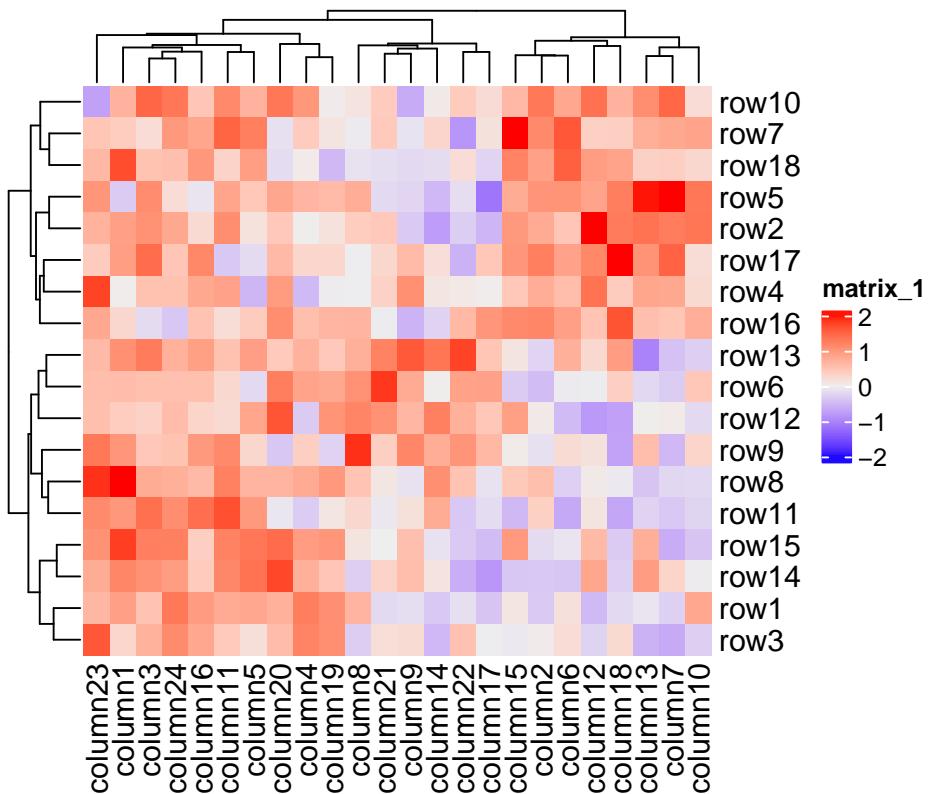
First let’s generate a random matrix where there are three groups in the columns and three groups in the rows:

```
set.seed(123)
nr1 = 4; nr2 = 8; nr3 = 6; nr = nr1 + nr2 + nr3
nc1 = 6; nc2 = 8; nc3 = 10; nc = nc1 + nc2 + nc3
mat = cbind(rbind(matrix(rnorm(nr1*nc1, mean = 1, sd = 0.5), nr = nr1),
                  matrix(rnorm(nr2*nc1, mean = 0, sd = 0.5), nr = nr2),
                  matrix(rnorm(nr3*nc1, mean = 0, sd = 0.5), nr = nr3)),
              rbind(matrix(rnorm(nr1*nc2, mean = 0, sd = 0.5), nr = nr1),
                    matrix(rnorm(nr2*nc2, mean = 1, sd = 0.5), nr = nr2),
                    matrix(rnorm(nr3*nc2, mean = 0, sd = 0.5), nr = nr3)),
              rbind(matrix(rnorm(nr1*nc3, mean = 0.5, sd = 0.5), nr = nr1),
                    matrix(rnorm(nr2*nc3, mean = 0.5, sd = 0.5), nr = nr2),
                    matrix(rnorm(nr3*nc3, mean = 1, sd = 0.5), nr = nr3)))
)
mat = mat[sample(nr, nr), sample(nc, nc)] # random shuffle rows and columns
rownames(mat) = paste0("row", seq_len(nr))
colnames(mat) = paste0("column", seq_len(nc))
```

Following command contains the minimal argument for the **Heatmap()** function which visualizes the matrix as a heatmap with default settings. Very similar as other heatmap tools, it draws the dendrograms, the row/column names and the heatmap legend. The default color schema is “blue-white-red” which is mapped to the minimal-mean-maximal values in the matrix. The title for the legend is

assigned with an internal index number.

```
Heatmap(mat)
```



The title for the legend is taken from the “name” of the heatmap by default. Each heatmap has a name which is like a unique identifier for the heatmap and it is important when you have a list of heatmaps. In later chapters, you will find the heatmap name is used for setting the “main heatmap” and is used for decoration on heatmaps. If the name is not assigned, an internal name is assigned to the heatmap in a form of `matrix_%d`. In the following examples in this chapter, we give the name `mat` to the heatmap (for which you will see the change of legend title in the next plot).

If you put `Heatmap()` inside a function or a `for/if/while` chunk, or the R script is run under command-line, you won’t see the heatmap after executing `Heatmap()`. In this case, you need to use `draw()` function explicitly as follows. We will explain this point in more detail in Section 2.11.

```
for(...) {
    ht = Heatmap(mat)
    draw(ht)
```

```
}
```

2.1 Colors

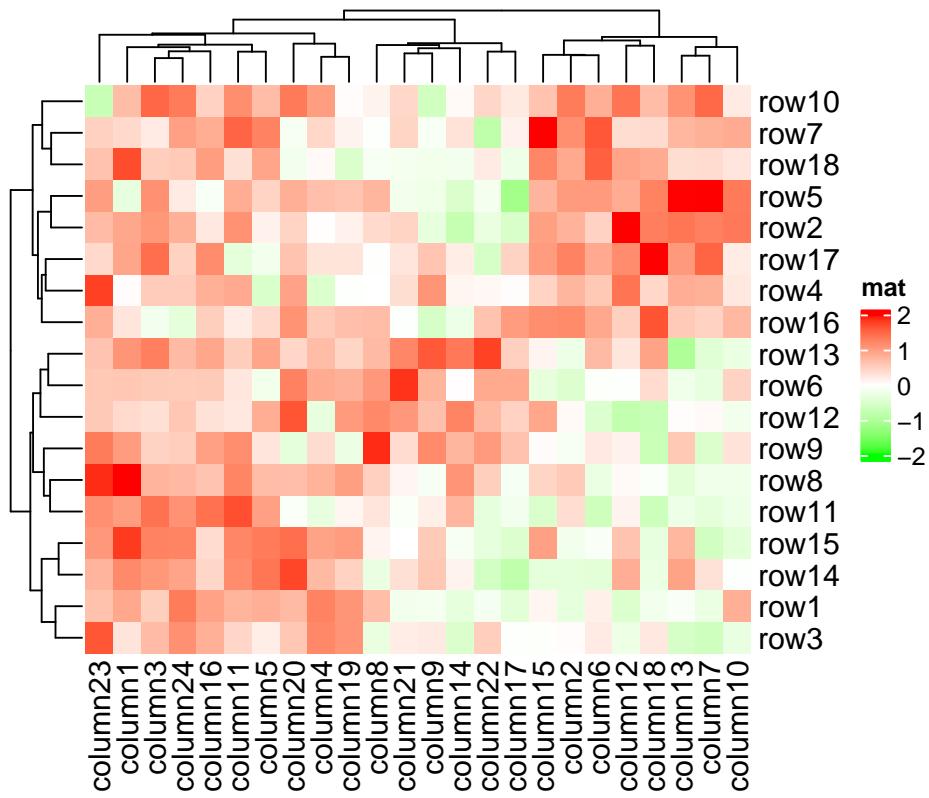
For heatmap visualization, colors are the major representation of the data matrix. In most cases, the heatmap visualizes a matrix with continuous numeric values. In this case, users should provide a color mapping function. A color mapping function should accept a vector of values and return a vector of corresponding colors. **Users should always use `circlize::colorRamp2()` function to generate the color mapping function** in `Heatmap()`. The two arguments for `colorRamp2()` is a vector of break values and a vector of corresponding colors. `colorRamp2()` linearly interpolates colors in every interval through LAB color space. Also using `colorRamp2()` helps to generate a legend with proper tick marks.

In following example, values between -2 and 2 are linearly interpolated to get corresponding colors, values larger than 2 are all mapped to red and values less than -2 are all mapped to green.

```
library(circlize)
col_fun = colorRamp2(c(-2, 0, 2), c("green", "white", "red"))
col_fun(seq(-3, 3))

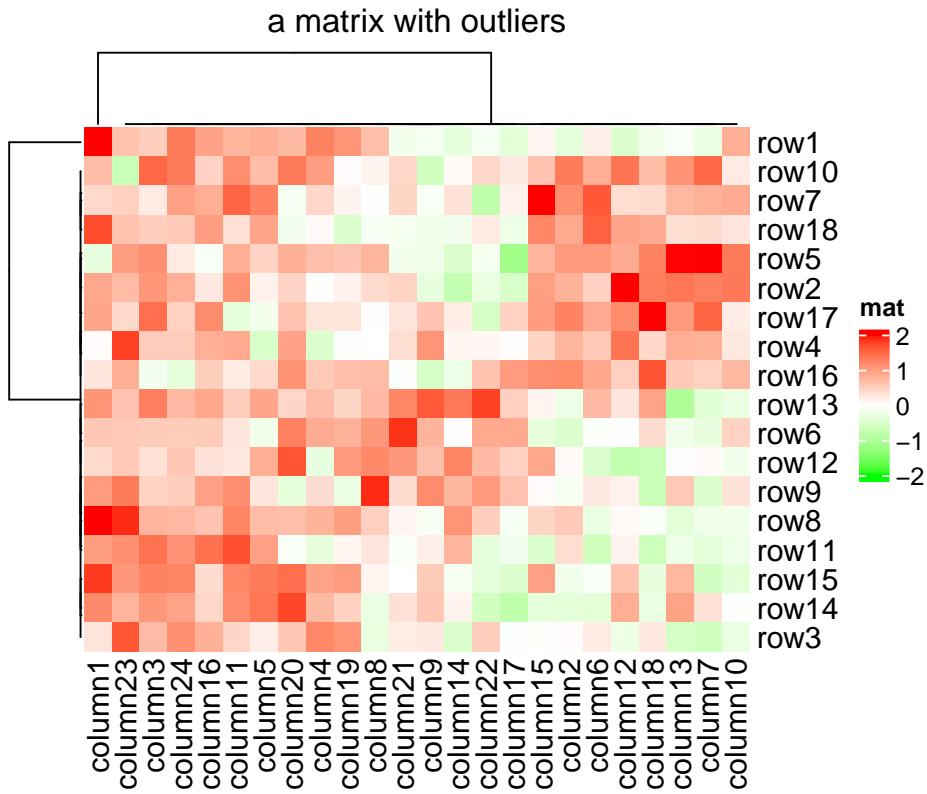
## [1] "#00FF00FF" "#00FF00FF" "#B1FF9AFF" "#FFFFFFFF" "#FF9E81FF" "#FF0000FF"
## [7] "#FF0000FF"

Heatmap(mat, name = "mat", col = col_fun)
```



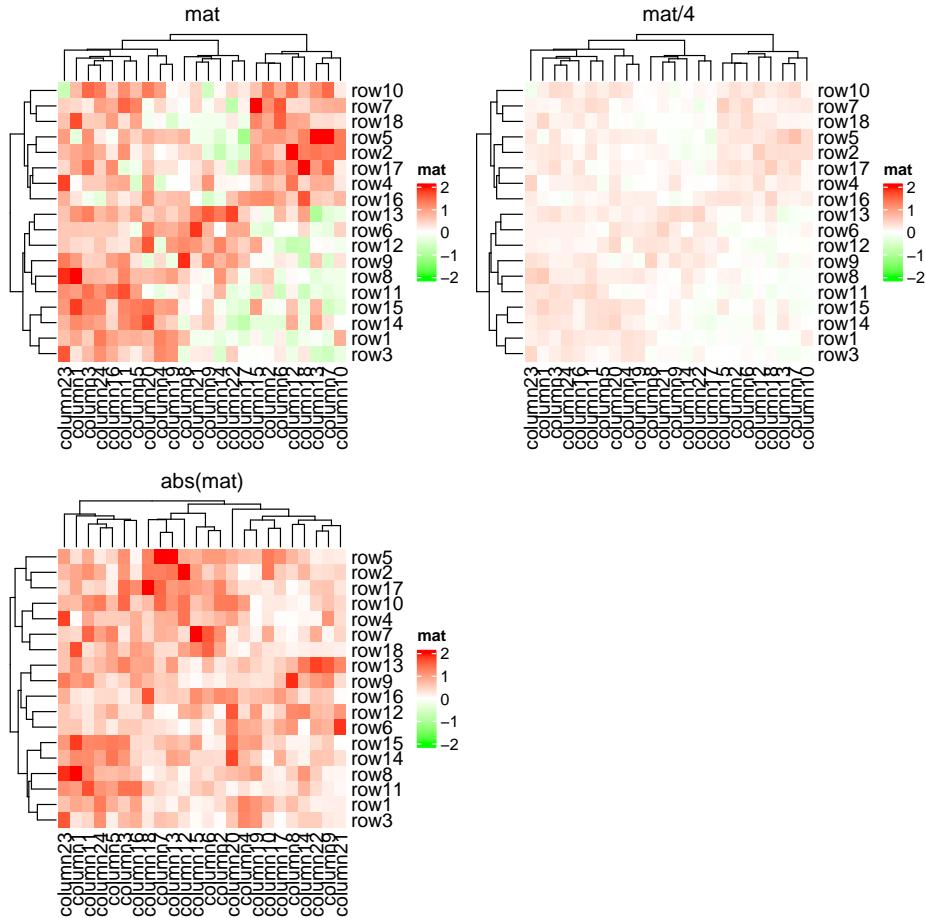
As you can see, the color mapping function exactly maps negative values to green and positive values to red, even when the distribution of negative values and positive values are not centric to zero. Also this color mapping function is **not affected by outliers**. In following plot, the clustering is heavily affected by the outlier (see the dendrogram) but not the color mapping.

```
mat2 = mat
mat2[1, 1] = 100000
Heatmap(mat2, name = "mat", col = col_fun,
        column_title = "a matrix with outliers")
```



More importantly, `colorRamp2()` makes colors in multiple heatmaps comparable if they are set with a same color mapping function. In following three heatmaps, a same color always corresponds to a same value.

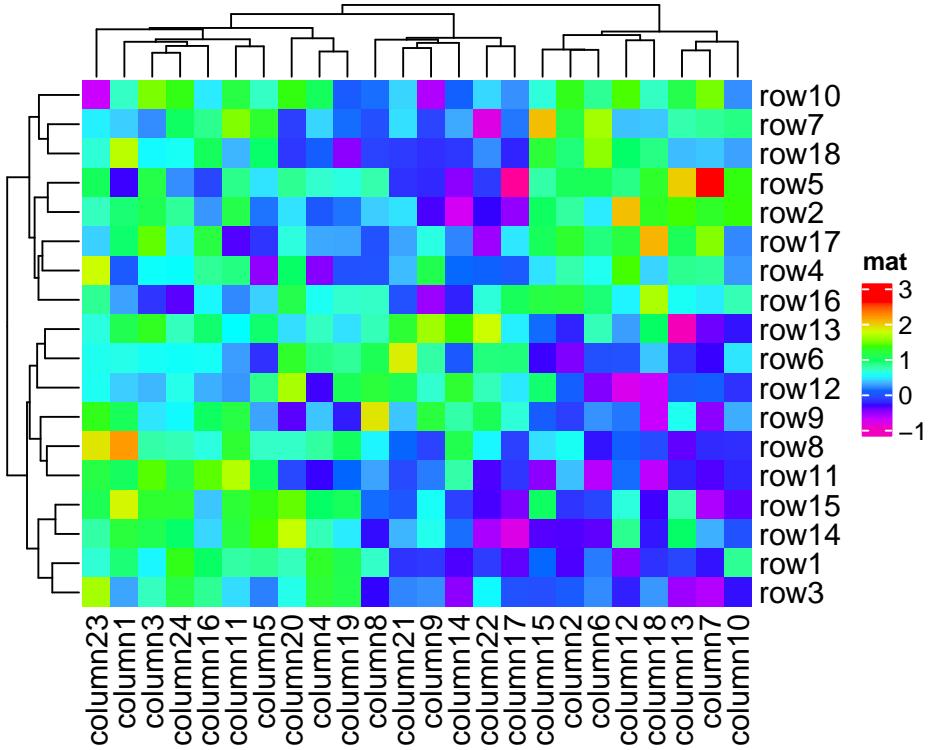
```
Heatmap(mat, name = "mat", col = col_fun, column_title = "mat")
Heatmap(mat/4, name = "mat", col = col_fun, column_title = "mat/4")
Heatmap(abs(mat), name = "mat", col = col_fun, column_title = "abs(mat)")
```



If the matrix is continuous, you can also simply provide a vector of colors and colors will be linearly interpolated. But remember this method is not robust to outliers because the mapping starts from the minimal value in the matrix and ends with the maximal value. Following color mapping setting is identical to `colorRamp2(seq(min(mat), max(mat), length = 10), rev(rainbow(10)))`.

```
Heatmap(mat, name = "mat", col = rev(rainbow(10)),
        column_title = "set a color vector for a continuous matrix")
```

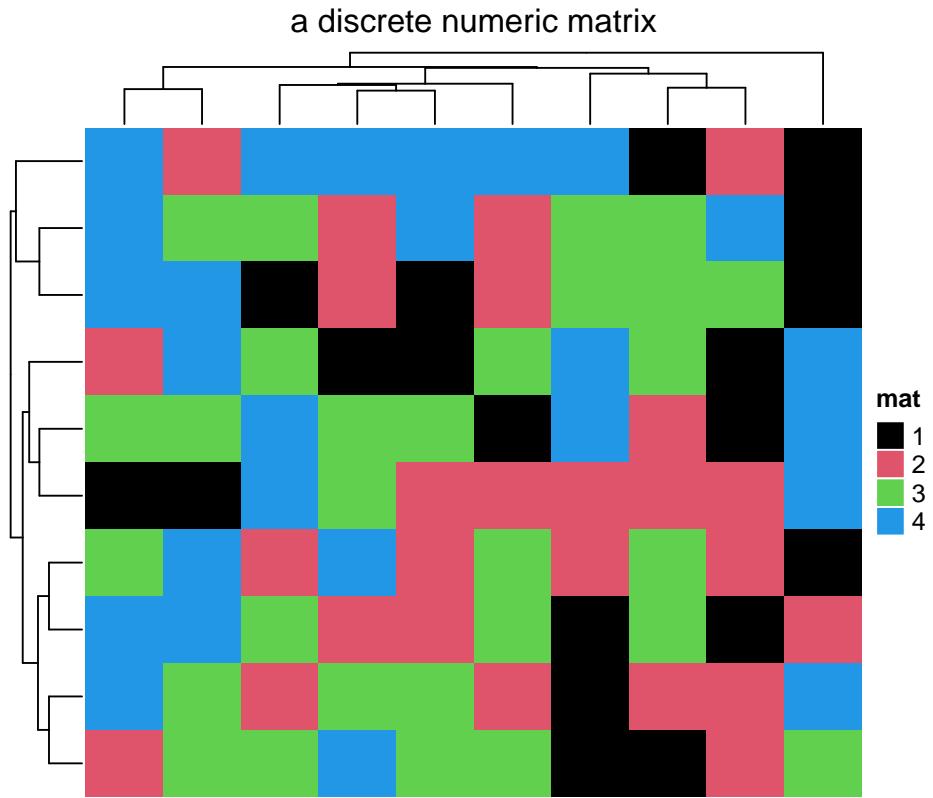
set a color vector for a continuous matrix



If the matrix contains discrete values (either numeric or character), colors should be specified as a named vector to make it possible for the mapping from discrete values to colors. If there is no name for the color vector, the order of colors corresponds to the order of `unique(mat)`. Note now the legend is generated from the color mapping vector and it is a “discrete legend.”

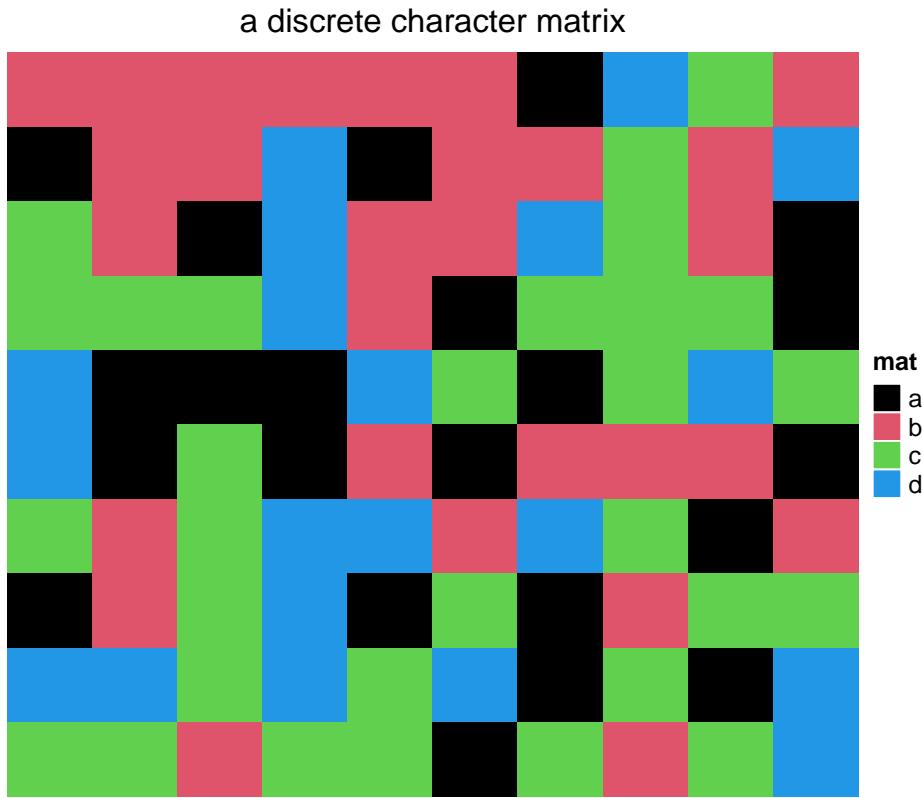
In the following example, we set colors for a discrete numeric matrix. You don’t need to convert it to a character matrix, just set the numbers as “names” of the color vector.

```
discrete_mat = matrix(sample(1:4, 100, replace = TRUE), 10, 10)
colors = structure(1:4, names = c("1", "2", "3", "4")) # black, red, green, blue
Heatmap(discrete_mat, name = "mat", col = colors,
       column_title = "a discrete numeric matrix")
```



For a character matrix, it is straightforward.

```
discrete_mat = matrix(sample(letters[1:4], 100, replace = TRUE), 10, 10)
colors = structure(1:4, names = letters[1:4])
Heatmap(discrete_mat, name = "mat", col = colors,
        column_title = "a discrete character matrix")
```

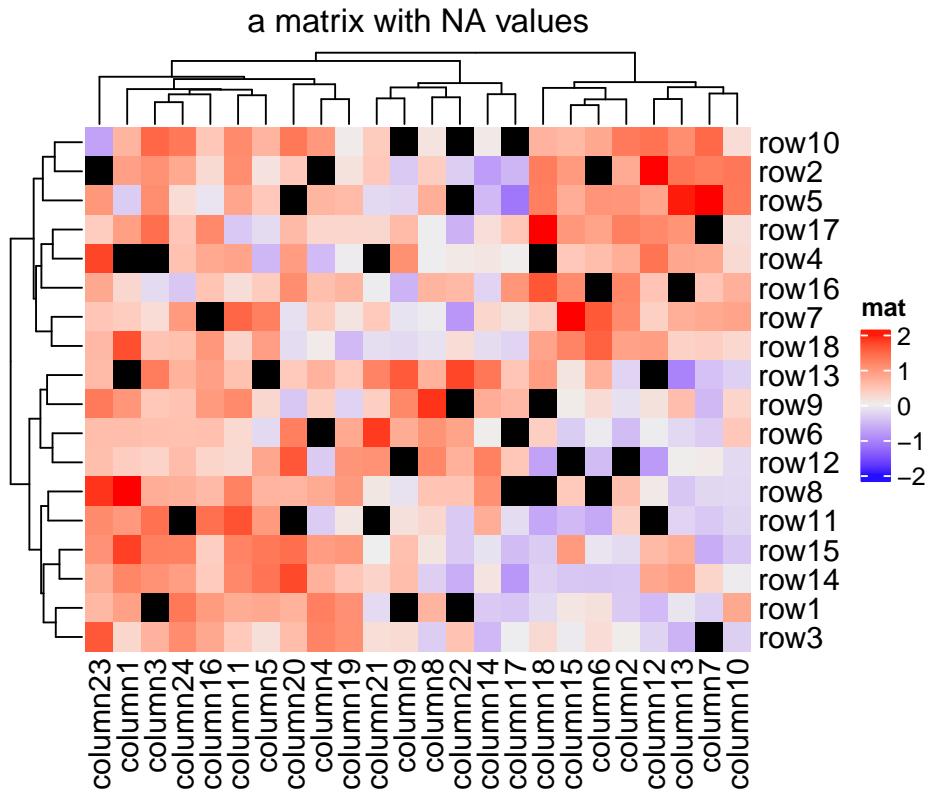


As you can see in the two examples above, for the numeric matrix (no matter the color is continuous mapping or discrete mapping), by default clustering is applied on both dimensions, while for character matrix, clustering is turned off (but you can still cluster a character matrix if you provide a proper distance metric for two character vectors, see example in Section 2.3.1).

NA is allowed in the matrix. You can control the color of NA by `na_col` argument (by default it is grey for NA). **A matrix that contains NA can be clustered by `Heatmap()`** as long as there is no NA distances between any of the rows or columns respectively. Usually these cases correspond to sparse matrices (filled with a lot of NA values), and indicate that the unknown values should be predicted via other methods first.

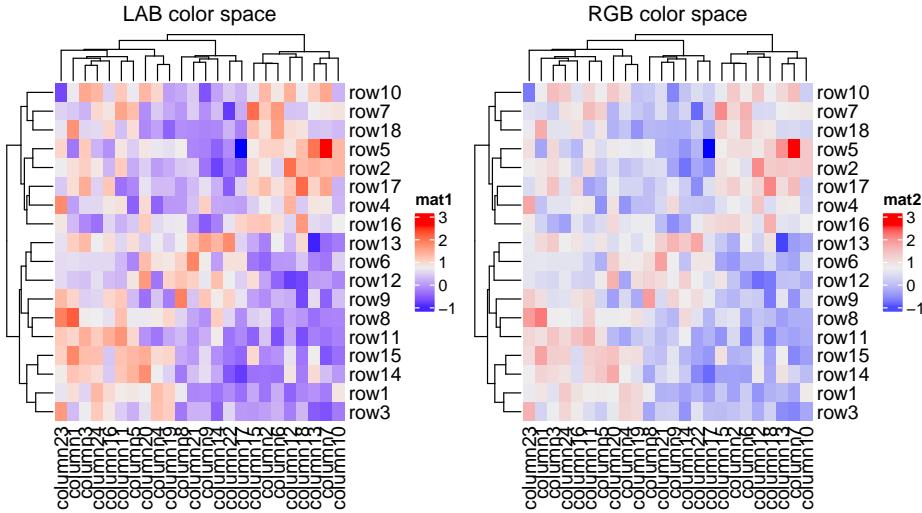
Note the NA value is not presented in the legend.

```
mat_with_na = mat
na_index = sample(c(TRUE, FALSE), nrow(mat)*ncol(mat), replace = TRUE, prob = c(1, 9))
mat_with_na[na_index] = NA
Heatmap(mat_with_na, name = "mat", na_col = "black",
        column_title = "a matrix with NA values")
```

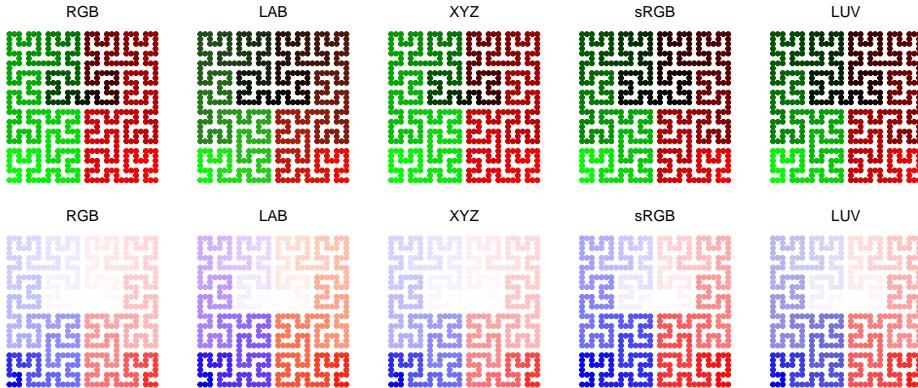


Color space is important for interpolating colors. By default, colors are linearly interpolated in LAB color space, but you can select the color space in `colorRamp2()` function. Compare following two plots. Can you see the difference?

```
f1 = colorRamp2(seq(min(mat), max(mat), length = 3), c("blue", "#EEEEEE", "red"))
f2 = colorRamp2(seq(min(mat), max(mat), length = 3), c("blue", "#EEEEEE", "red"), space = "RGB")
Heatmap(mat, name = "mat1", col = f1, column_title = "LAB color space")
Heatmap(mat, name = "mat2", col = f2, column_title = "RGB color space")
```



In following plots, corresponding values change evenly on the folded lines, you can see how colors change under different color spaces (top plots: green-black-red, bottom plots: blue-white-red. The plot is made by **HilbertCurve** package).

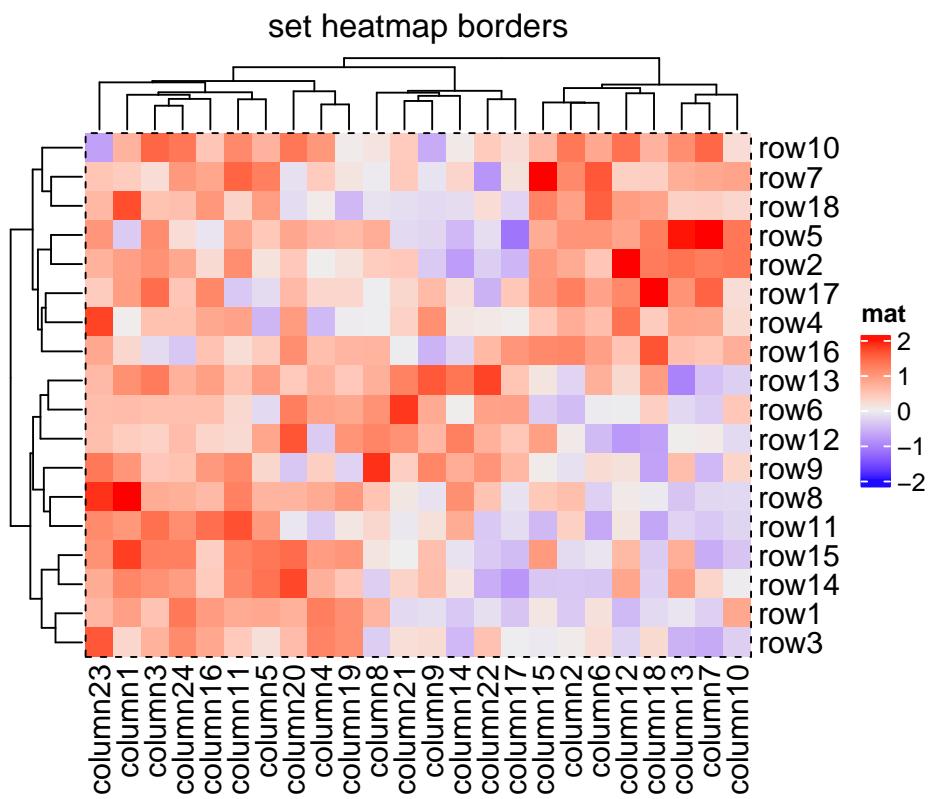


Last but not the least, colors for the heatmap borders can be set by the `border`/`border_gp` and `rect_gp` arguments. `border`/`border_gp` controls the global border of the heatmap body and `rect_gp` controls the border of the grids/cells in the heatmap.

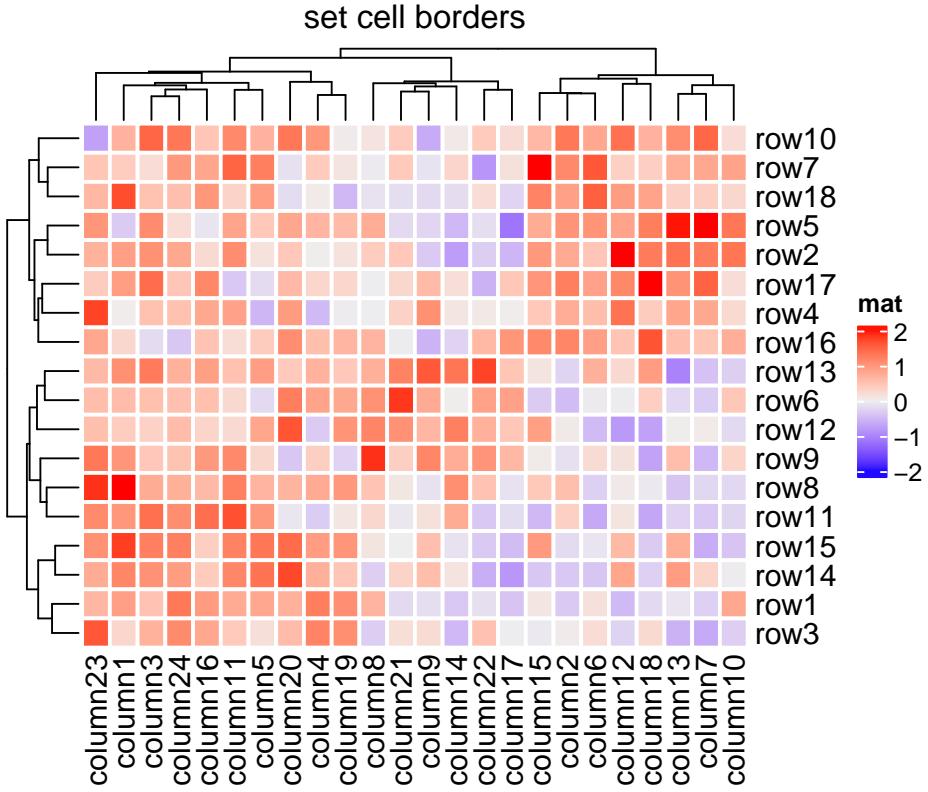
The value of `border` can be logical (`TRUE` corresponds to `black`) or a character of color (e.g. `red`). The use for `border` argument is only for historical reason, here you can also set `border_gp` argument which should be a `gpar` object.

`rect_gp` is a `gpar` object which means you can only set it by `grid::gpar()`. Since the filled color is already controlled by the heatmap color mapping, you can only set the `col`/`lwd`/`lty` parameters in `gpar()` to control the border of the heatmap grids.

```
Heatmap(mat, name = "mat", border_gp = gpar(col = "black", lty = 2),
        column_title = "set heatmap borders")
```



```
Heatmap(mat, name = "mat", rect_gp = gpar(col = "white", lwd = 2),
        column_title = "set cell borders")
```



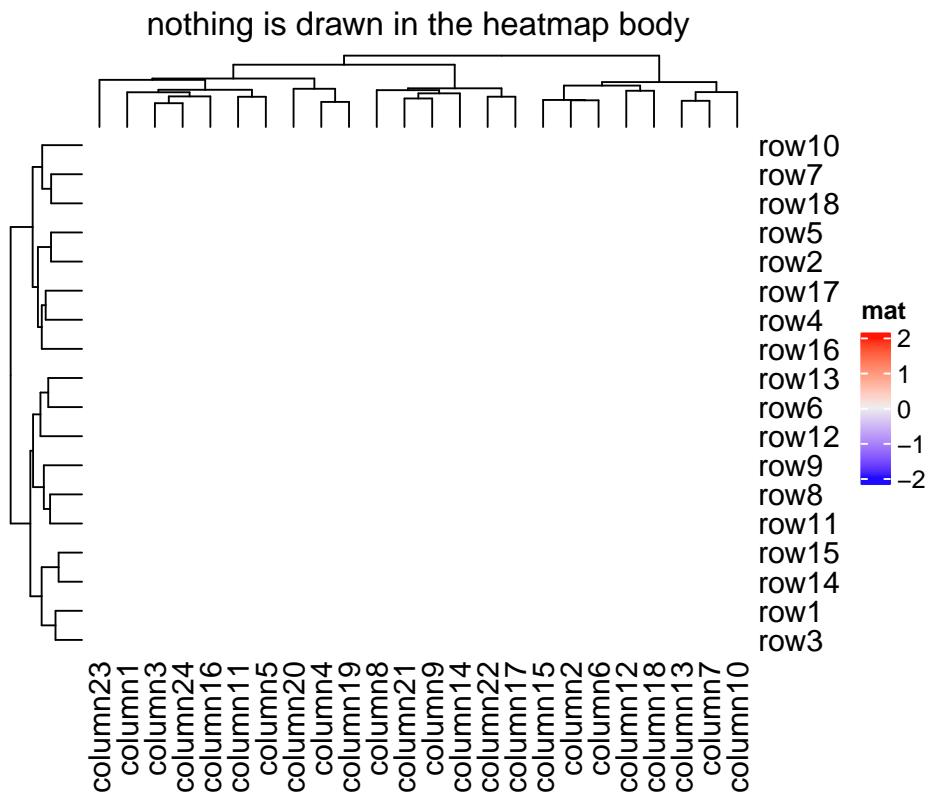
If `col` is not set, the default color mapping by `Heatmap()` is designed with trying to be as convenient and meaningful as possible. Following are the rules for the default color mapping (by `ComplexHeatmap:::default_col()`):

- If the values are characters, the colors are generated by `circlize::rand_color()`.
- If the values are from the heatmap annotation and are numeric, colors are mapped between white and one random color by linearly interpolating to the minimum and maximum.
- If the values are from the matrix (let's denote it as M):
 - If the fraction of positive values in M is between 25% and 75%, colors are mapped to blue, white and red by linearly interpolating to $-q$, 0 and q , where q is the maximum of $|M|$ if the number of unique values is less than 100, or q is the 99th percentile of $|M|$ if there are more than 100 values in the matrix. This color mapping is centric to zero.
 - Or else the colors are mapped to blue, white and red by linearly interpolating to q_1 , $(q_1 + q_2)/2$ and q_2 , where q_1 and q_2 are minimum and maximum if the number of unique values is M is less than 100. If there are more than 100 values in the matrix, q_1 is the 1st percentile and q_2 is the 99th percentile in M .

`rect_gp` allows a non-standard parameter `type`. If it is set to "none", the

clustering is still applied but nothing is drawn on the heatmap body. The customized graphics on heatmap body can be later added via a self-defined `cell_fun` or `layer_fun` (explained in Section 2.9).

```
Heatmap(mat, name = "mat", rect_gp = gpar(type = "none"),
        column_title = "nothing is drawn in the heatmap body")
```

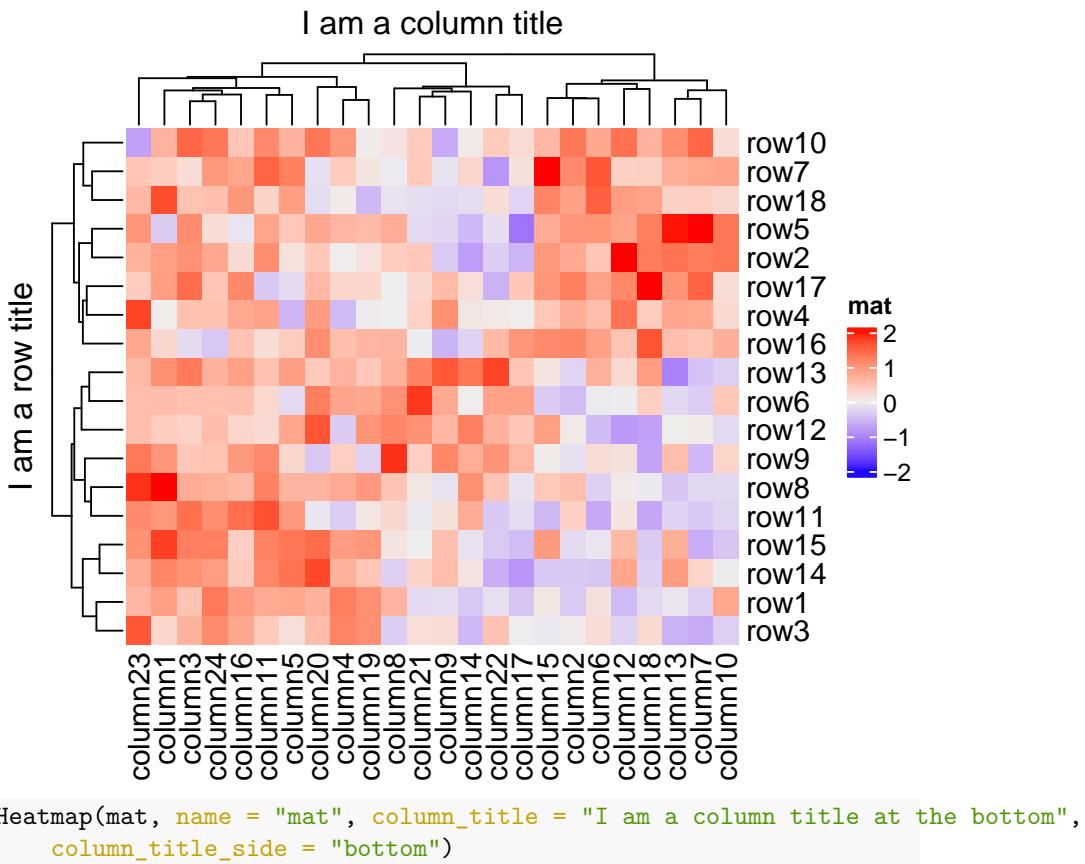


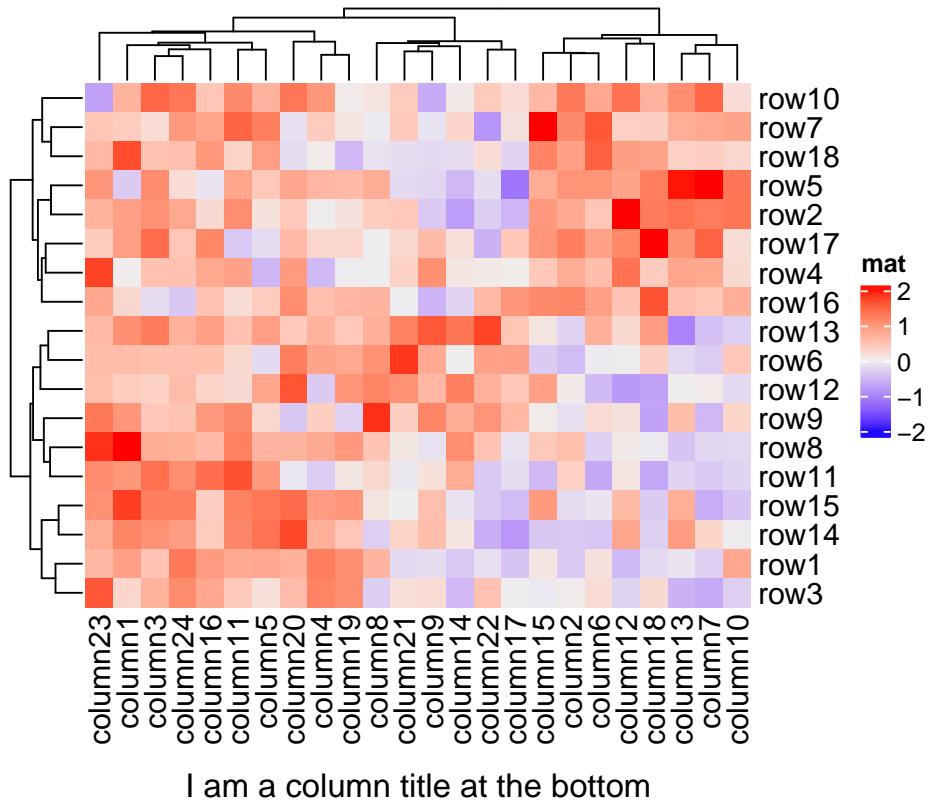
2.2 Titles

The title of the heatmap basically tells what the plot is about. In **Complex-Heatmap** package, you can set heatmap title either by the row or/and by the column. Note at a same time you can only put e.g. column title either on the top or at the bottom of the heatmap.

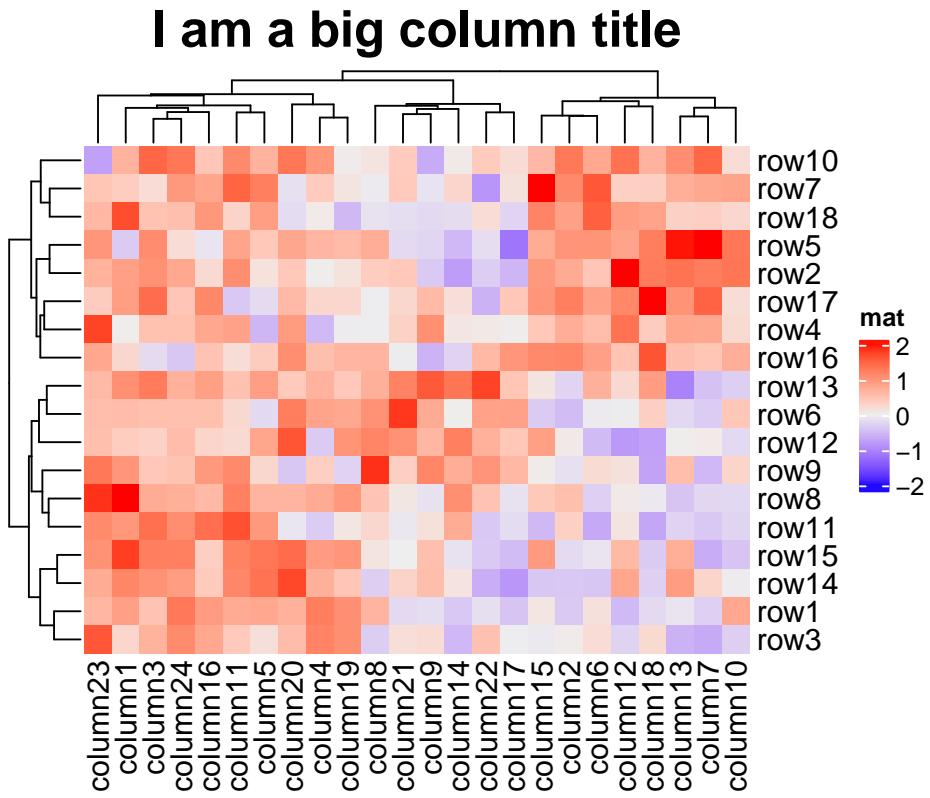
The graphic parameters can be set by `row_title_gp` and `column_title_gp` respectively. Please remember you should use `gpar()` to specify graphics parameters.

```
Heatmap(mat, name = "mat", column_title = "I am a column title",
        row_title = "I am a row title")
```



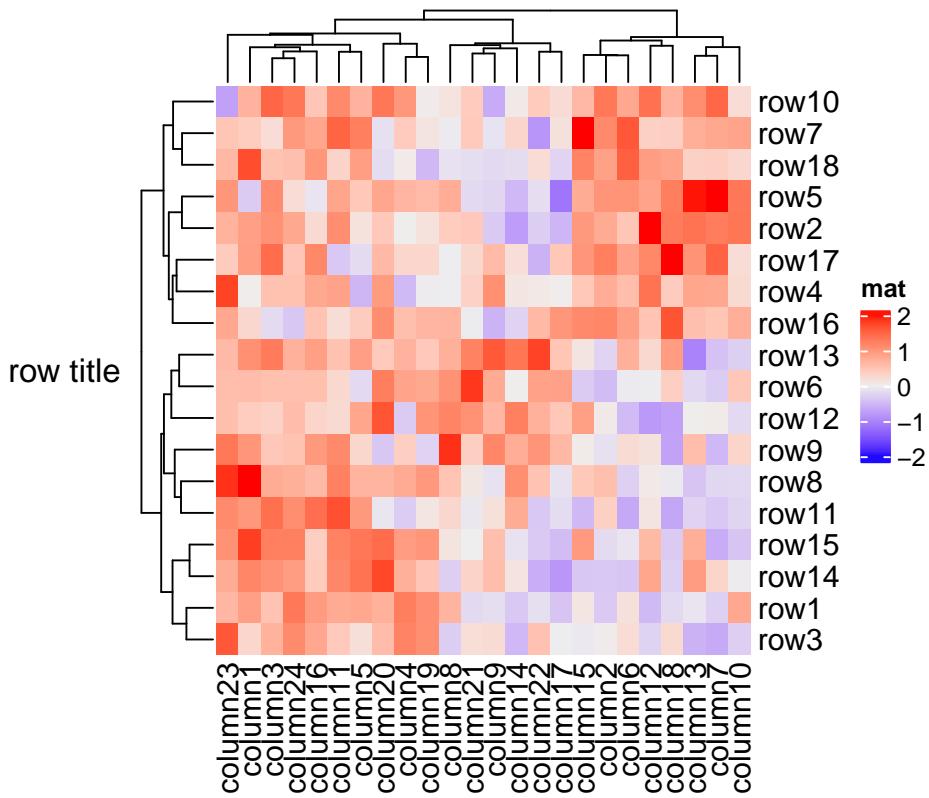


```
Heatmap(mat, name = "mat", column_title = "I am a big column title",
        column_title_gp = gpar(fontsize = 20, fontface = "bold"))
```



Rotations for titles can be set by `row_title_rot` and `column_title_rot`, but only horizontal and vertical rotations are allowed.

```
Heatmap(mat, name = "mat", row_title = "row title", row_title_rot = 0)
```

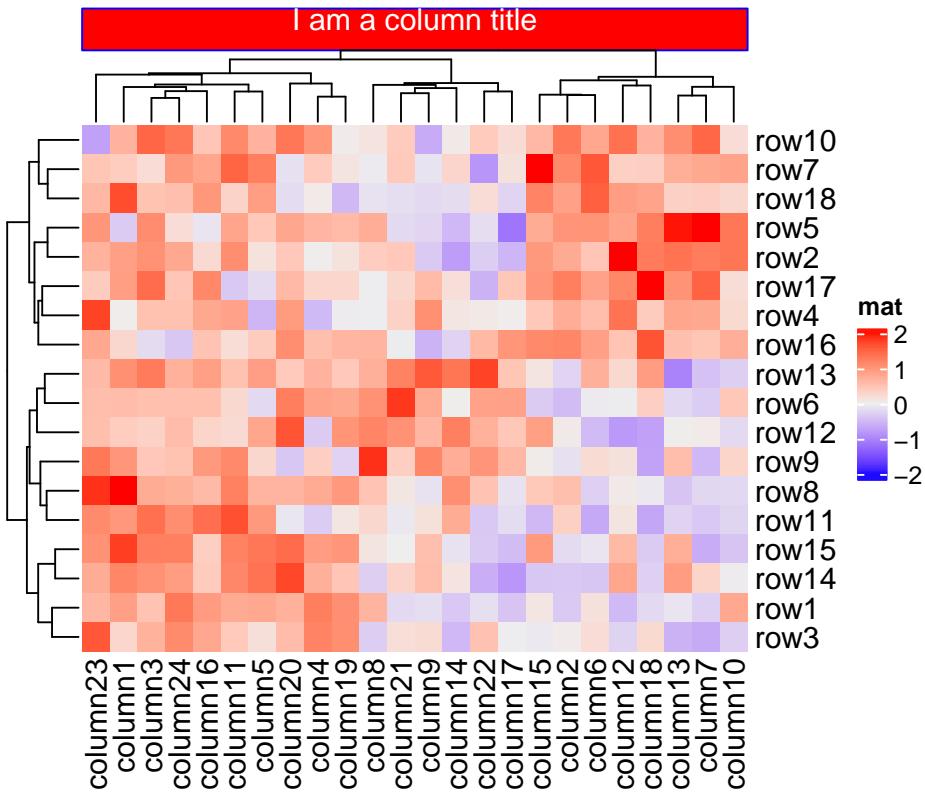


Row or column title supports as a template which is used when rows or columns are split in the heatmap (because there will be multiple row/column titles). This functionality is introduced in Section 2.7. A quick example would be:

```
# code only for demonstration
# row title would be cluster_1 and cluster_2
Heatmap(mat, name = "mat", row_km = 2, row_title = "cluster_%s")
```

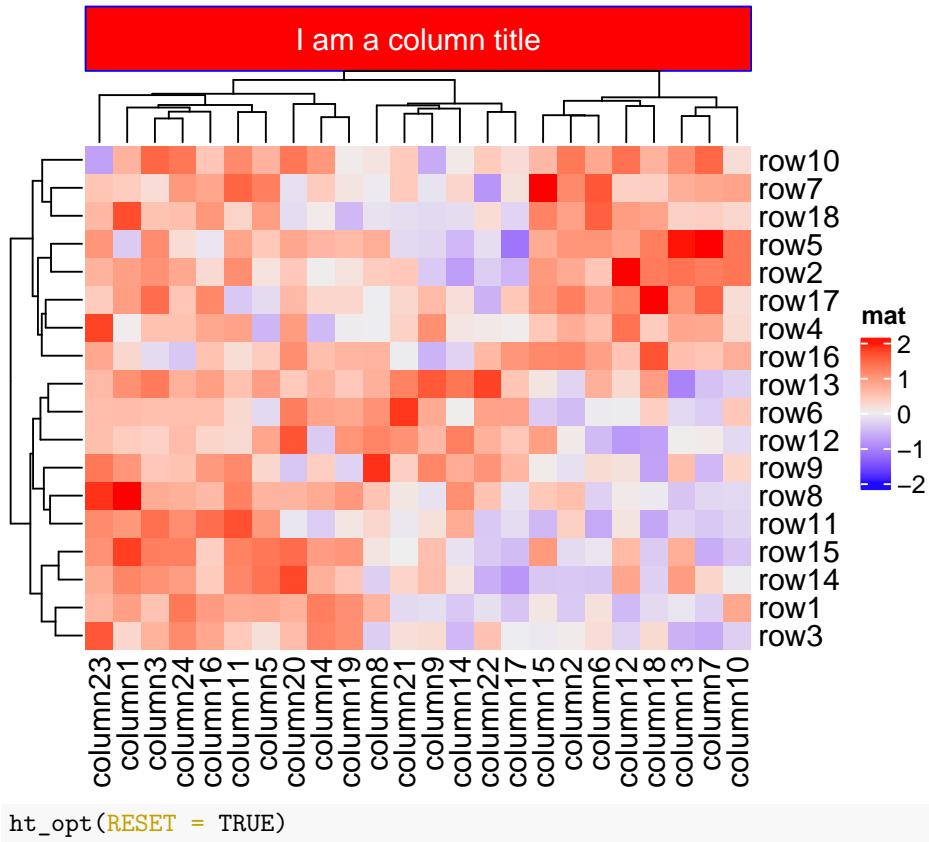
You can set `fill` parameter in `row_title_gp` and `column_title_gp` to set the background color of titles. Since `col` in e.g. `row_title_gp` controls the color of text, `border` is used to control the color of the background border.

```
Heatmap(mat, name = "mat", column_title = "I am a column title",
       column_title_gp = gpar(fill = "red", col = "white", border = "blue"))
```



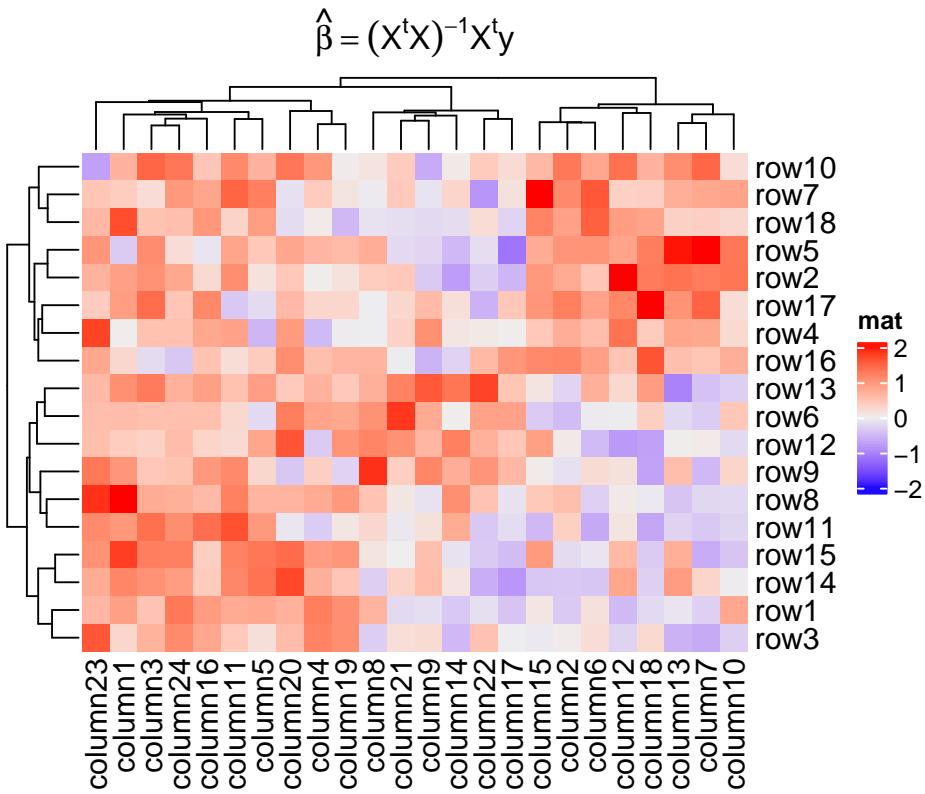
You might have found there is no space on the top of the column title when the background is colored. This is mainly for being consistent to the figure titles of ggplot2 figures (when you make a multiple-panel figure mixed with ComplexHeatmap and ggplot2 plots). This can be adjusted by setting the global parameter `ht_opt$TITLE_PADDING`. The use for `ht_opt()` will be introduced in Section 4.13.

```
ht_opt$TITLE_PADDING = unit(c(8.5, 8.5), "points")
Heatmap(mat, name = "mat", column_title = "I am a column title",
        column_title_gp = gpar(fill = "red", col = "white", border = "blue"))
```



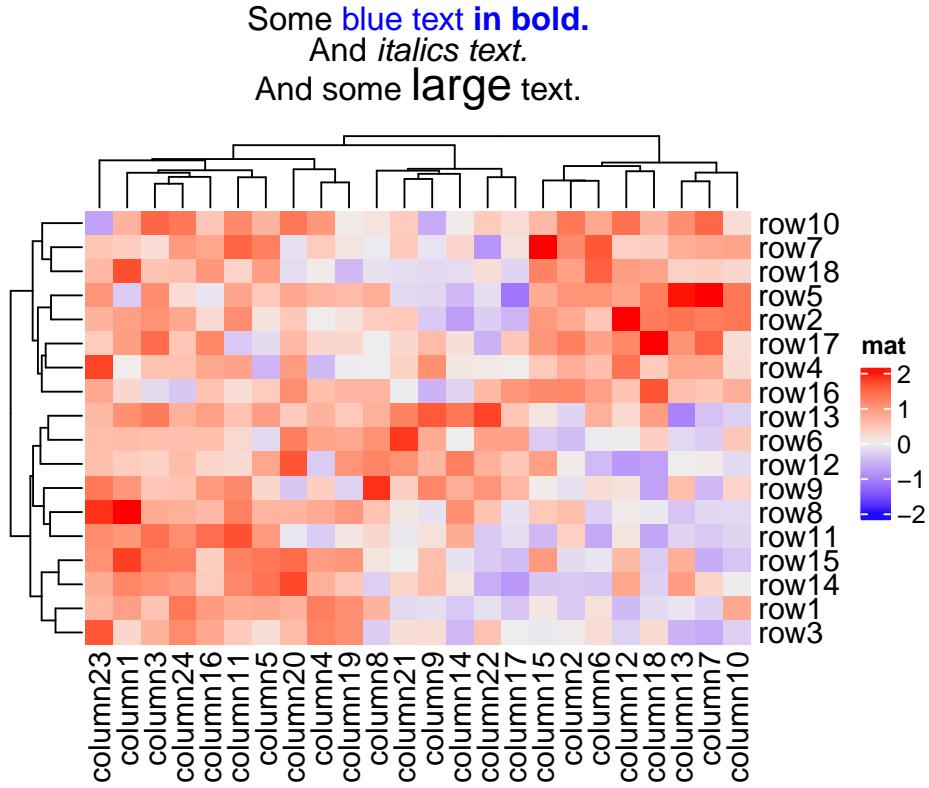
Title can be set as mathematical formulas.

```
Heatmap(mat, name = "mat",
        column_title = expression(hat(beta) == (X^t * X)^{-1} * X^t * y))
```



More complicated text can be drawn as the heatmap title by using the `gridtext` package. Following plot fives a quick example. See Section 10.3.1 for more examples.

```
Heatmap(mat, name = "mat",
        column_title = gt_render(
            paste0("Some <span style='color:blue'>blue text **in bold.**</span><br>",
                  "And *italics text.*<br>And some ",
                  "<span style='font-size:18pt; color:black'>large</span> text."),
            r = unit(2, "pt"),
            padding = unit(c(2, 2, 2, 2), "pt")
        )
    )
```



2.3 Clustering

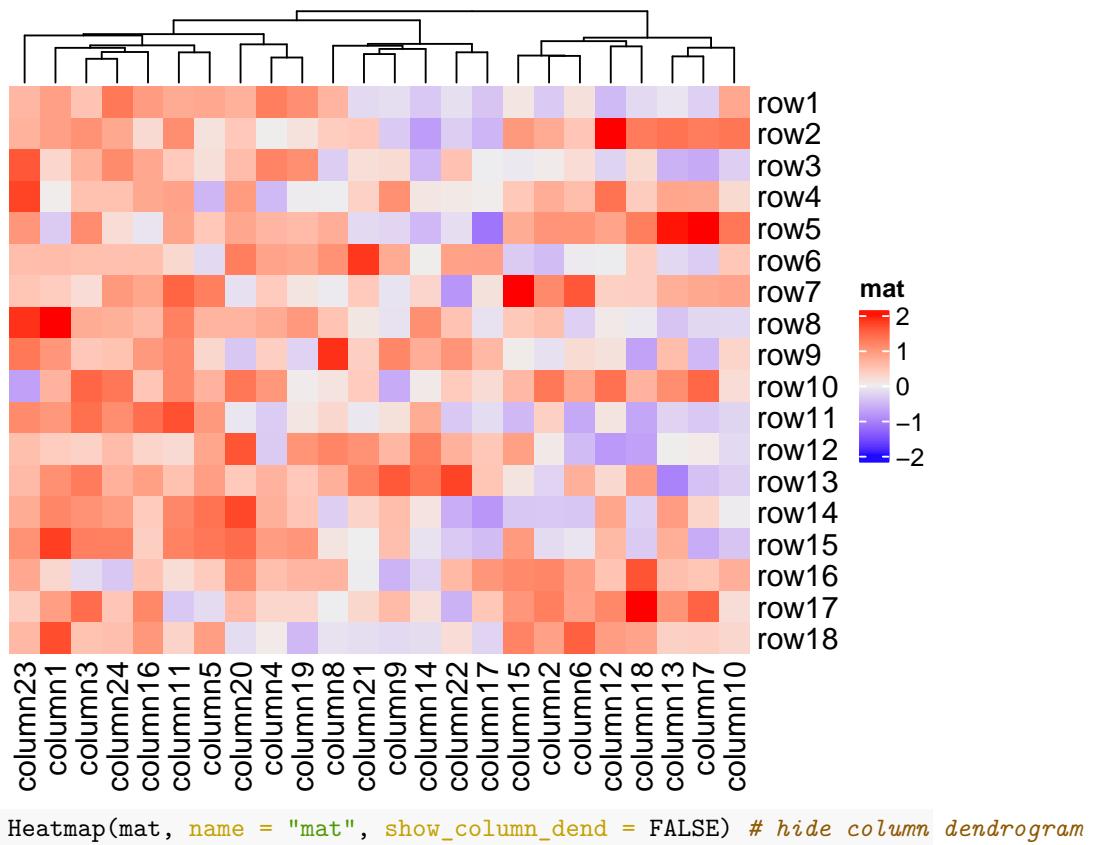
Clustering might be the key component of heatmap visualization. In **Complex-Heatmap** package, hierarchical clustering is supported with great flexibility. You can specify the clustering either by:

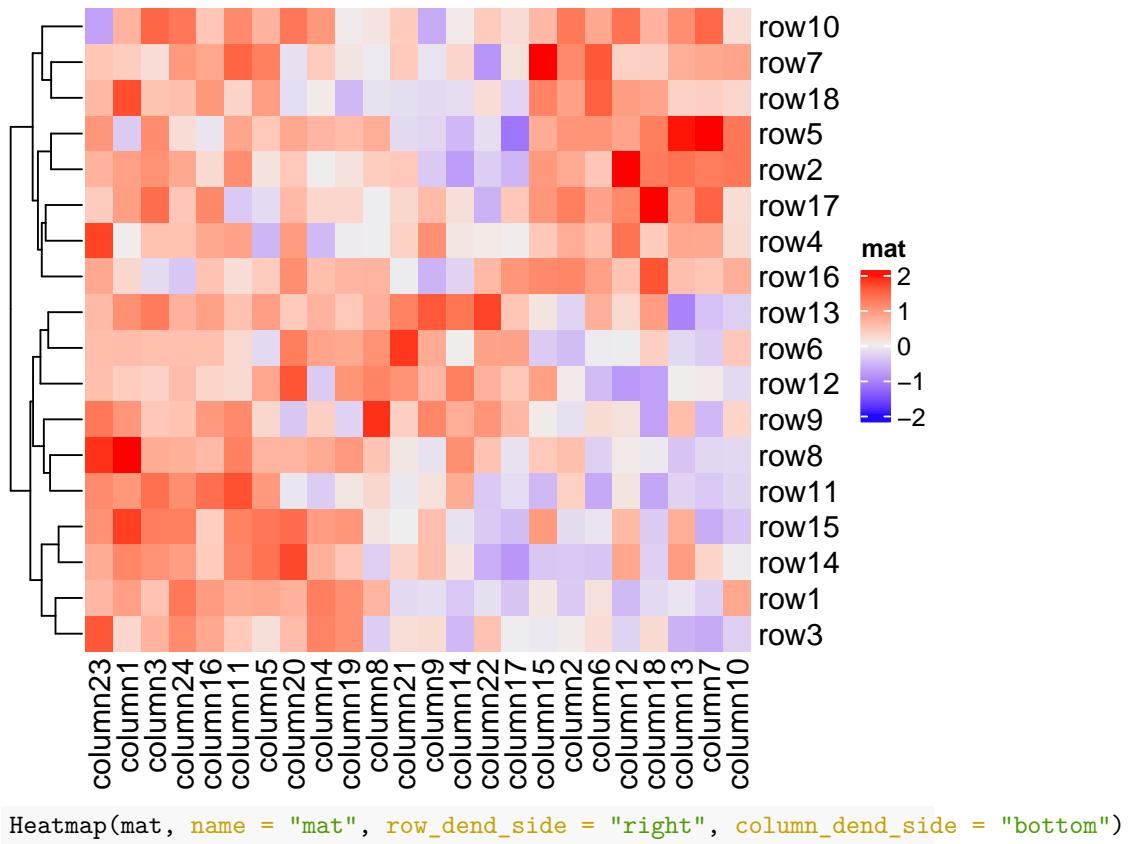
- a pre-defined distance method (e.g. "euclidean" or "pearson"),
- a distance function,
- a object that already contains clustering (a `hclust` or `dendrogram` object or object that can be coerced to `dendrogram` class),
- a clustering function.

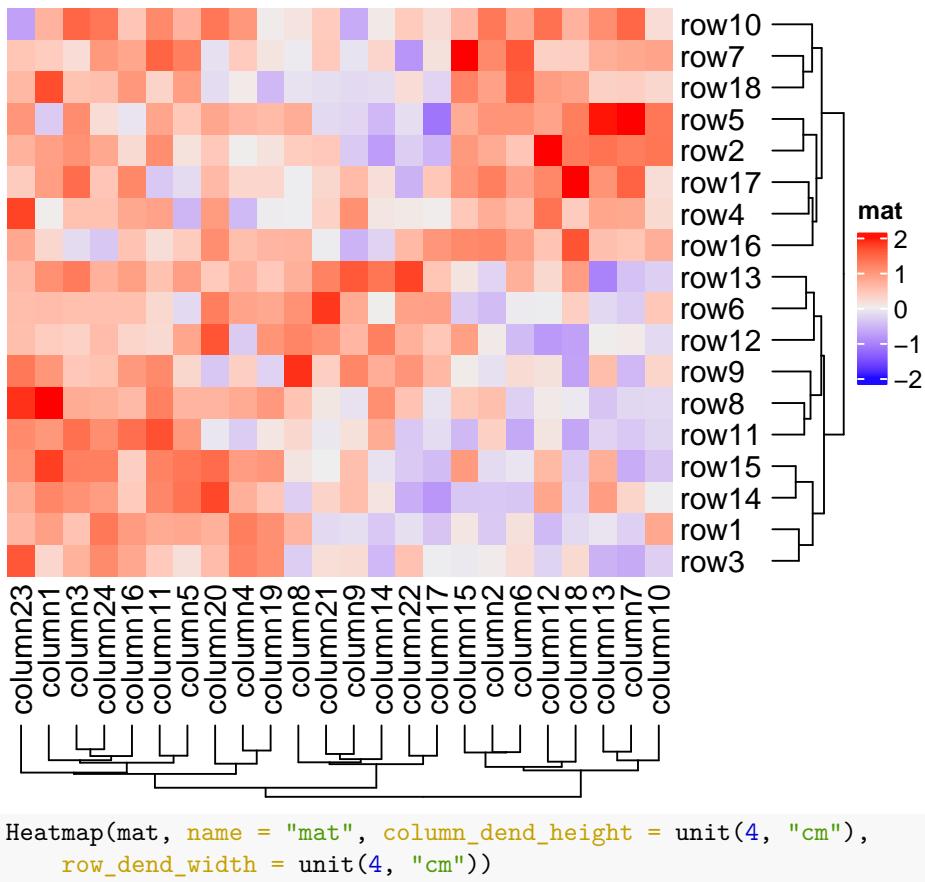
It is also possible to render the dendograms with different colors and styles for different nodes and branches for better revealing structures of the dendrogram (e.g. by `dendextend::color_branches()`).

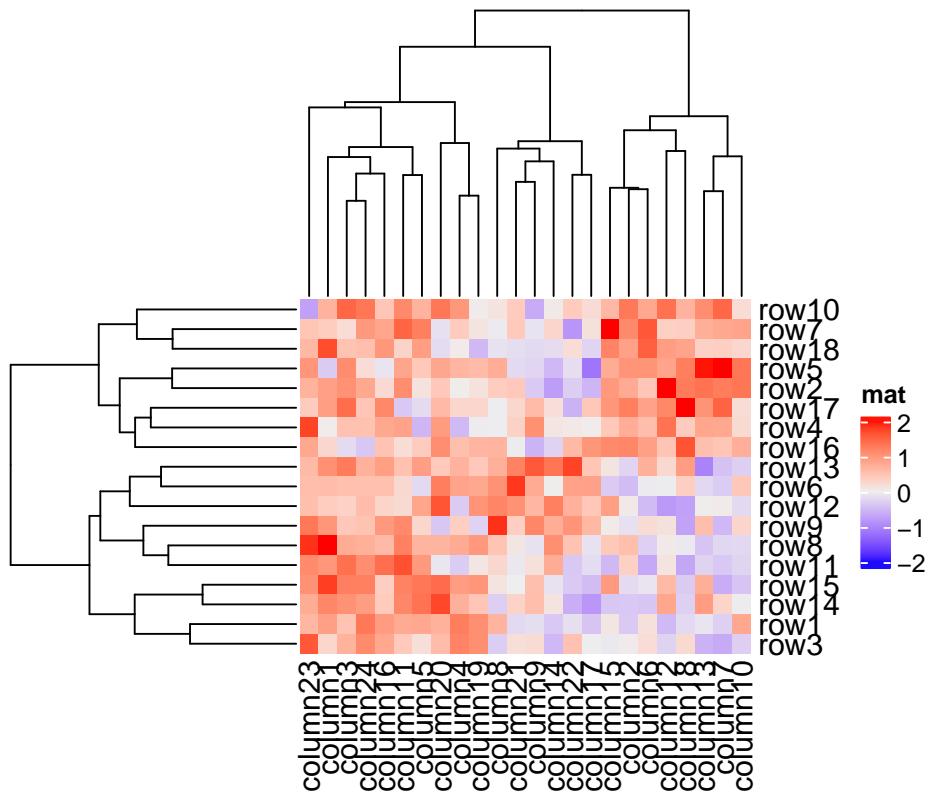
First, there are general settings for the clustering, e.g. whether to apply clustering or show dendograms, the side of the dendograms and heights of the dendograms.

```
Heatmap(mat, name = "mat", cluster_rows = FALSE) # turn off row clustering
```







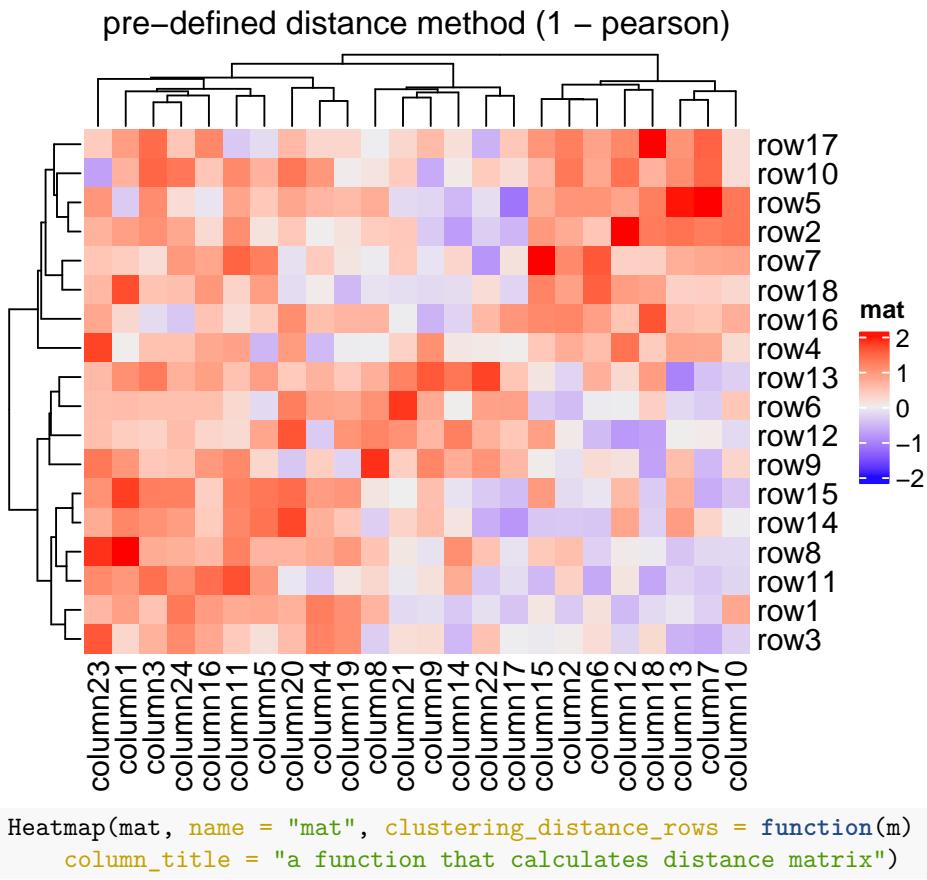


2.3.1 Distance methods

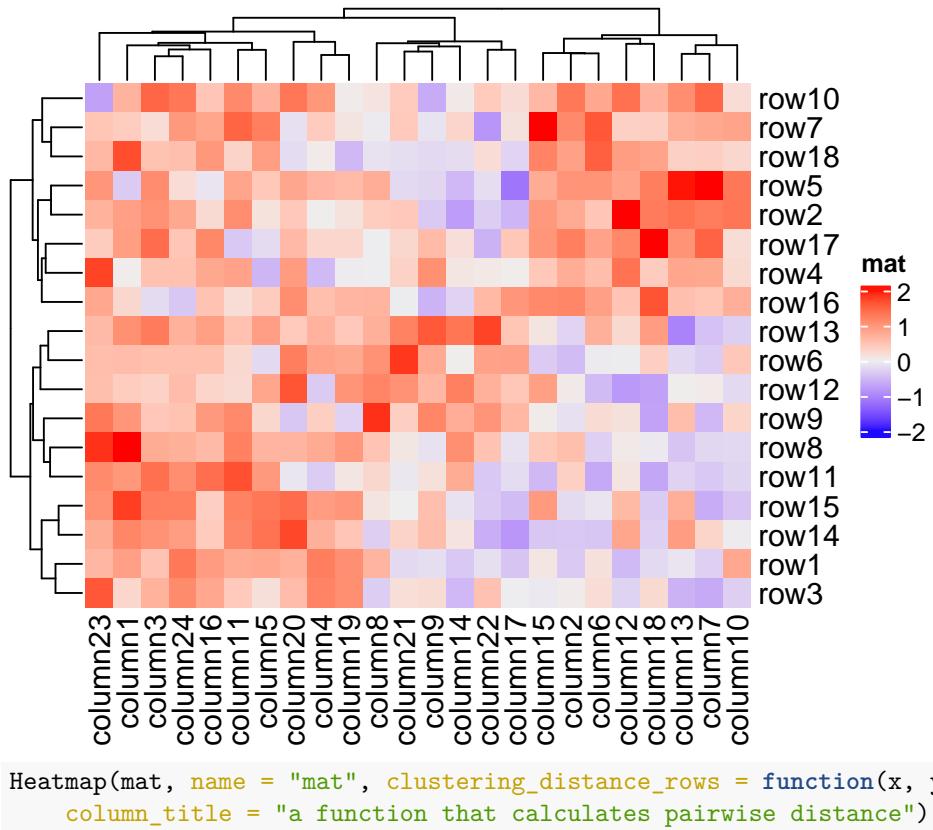
Hierarchical clustering is performed in two steps: calculate the distance matrix and apply clustering. There are three ways to specify distance metric for clustering:

- specify distance as a pre-defined option. The valid values are the supported methods in `dist()` function and in "pearson", "spearman" and "kendall". The correlation distance is defined as $1 - \text{cor}(x, y, \text{method})$. All these built-in distance methods allow NA values.
- a self-defined function which calculates distance from a matrix. The function should only contain one argument. Please note for clustering on columns, the matrix will be transposed automatically.
- a self-defined function which calculates distance from two vectors. The function should only contain two arguments. Note this might be slow because it is implemented by two nested `for` loop.

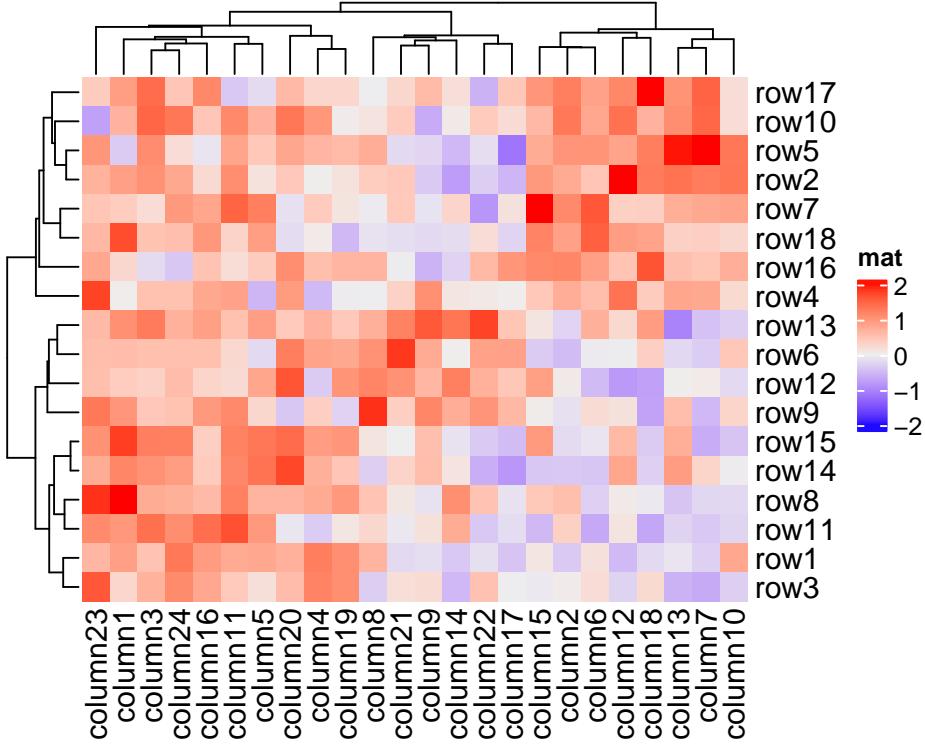
```
Heatmap(mat, name = "mat", clustering_distance_rows = "pearson",
        column_title = "pre-defined distance method (1 - pearson)")
```



a function that calculates distance matrix



a function that calculates pairwise distance



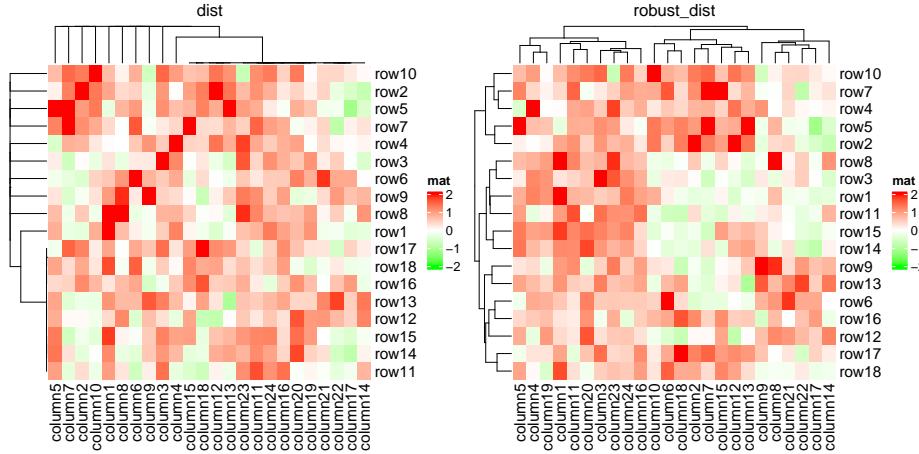
Based on these features, we can apply clustering which is robust to outliers based on the pairwise distance. Note here we set the color mapping function because we don't want outliers affect the colors.

```
mat_with_outliers = mat
for(i in 1:10) mat_with_outliers[i, i] = 1000
robust_dist = function(x, y) {
  qx = quantile(x, c(0.1, 0.9))
  qy = quantile(y, c(0.1, 0.9))
  l = x > qx[1] & x < qx[2] & y > qy[1] & y < qy[2]
  x = x[l]
  y = y[l]
  sqrt(sum((x - y)^2))
}
```

We can compare the two heatmaps with or without the robust distance method:

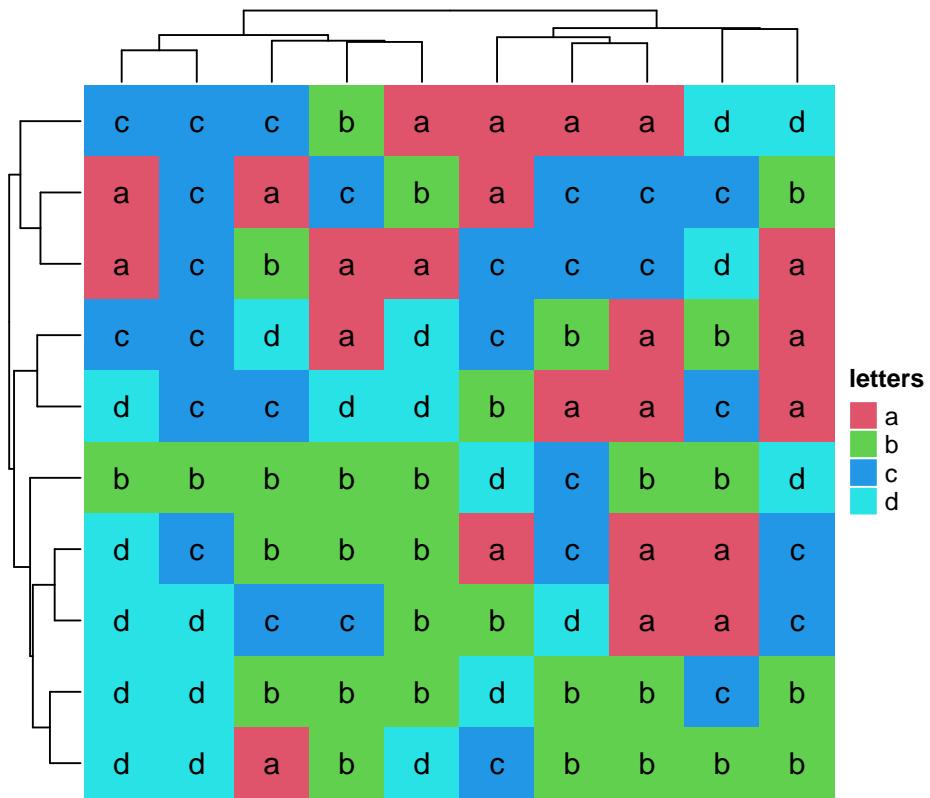
```
Heatmap(mat_with_outliers, name = "mat",
       col = colorRamp2(c(-2, 0, 2), c("green", "white", "red")),
       column_title = "dist")
Heatmap(mat_with_outliers, name = "mat",
       col = colorRamp2(c(-2, 0, 2), c("green", "white", "red")),
```

```
clustering_distance_rows = robust_dist,
clustering_distance_columns = robust_dist,
column_title = "robust_dist")
```



If there are proper distance methods (like methods in **stringdist** package), you can also cluster a character matrix. **cell_fun** argument will be introduced in Section 2.9.

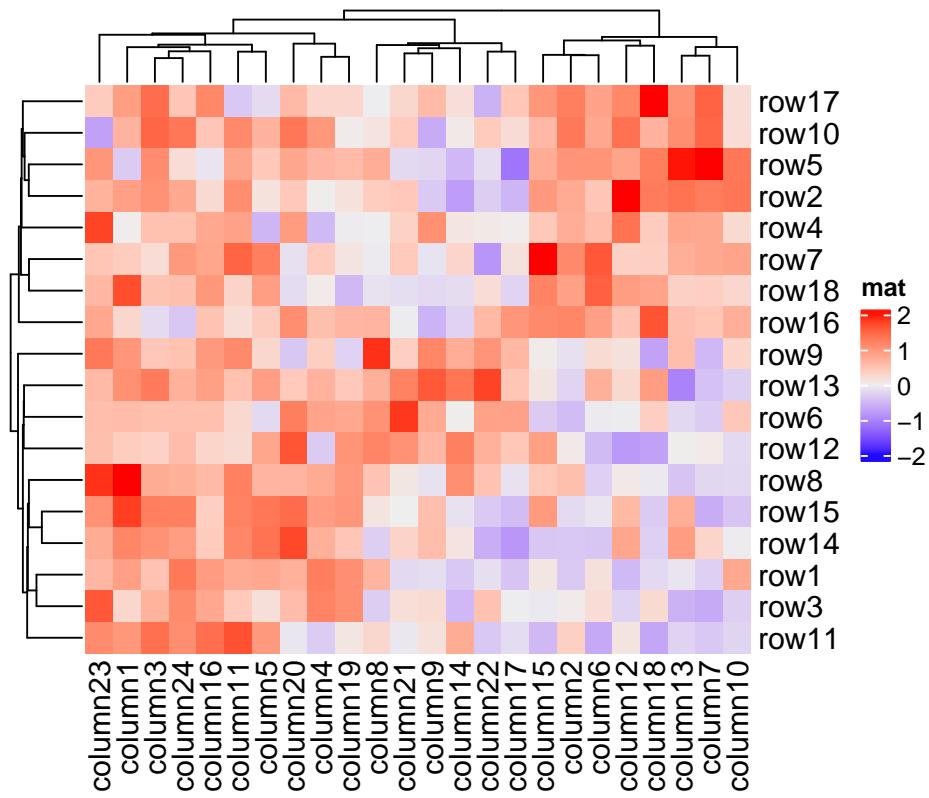
```
mat_letters = matrix(sample(letters[1:4], 100, replace = TRUE), 10)
# distance in the ASCII table
dist_letters = function(x, y) {
  x = strtoi(charToRaw(paste(x, collapse = "")), base = 16)
  y = strtoi(charToRaw(paste(y, collapse = "")), base = 16)
  sqrt(sum((x - y)^2))
}
Heatmap(mat_letters, name = "letters", col = structure(2:5, names = letters[1:4]),
        clustering_distance_rows = dist_letters,
        clustering_distance_columns = dist_letters,
        cell_fun = function(j, i, x, y, w, h, col) { # add text to each grid
          grid.text(mat_letters[i, j], x, y)
})
```



2.3.2 Clustering methods

Method to perform hierarchical clustering can be specified by `clustering_method_rows` and `clustering_method_columns`. Possible methods are those supported in `hclust()` function.

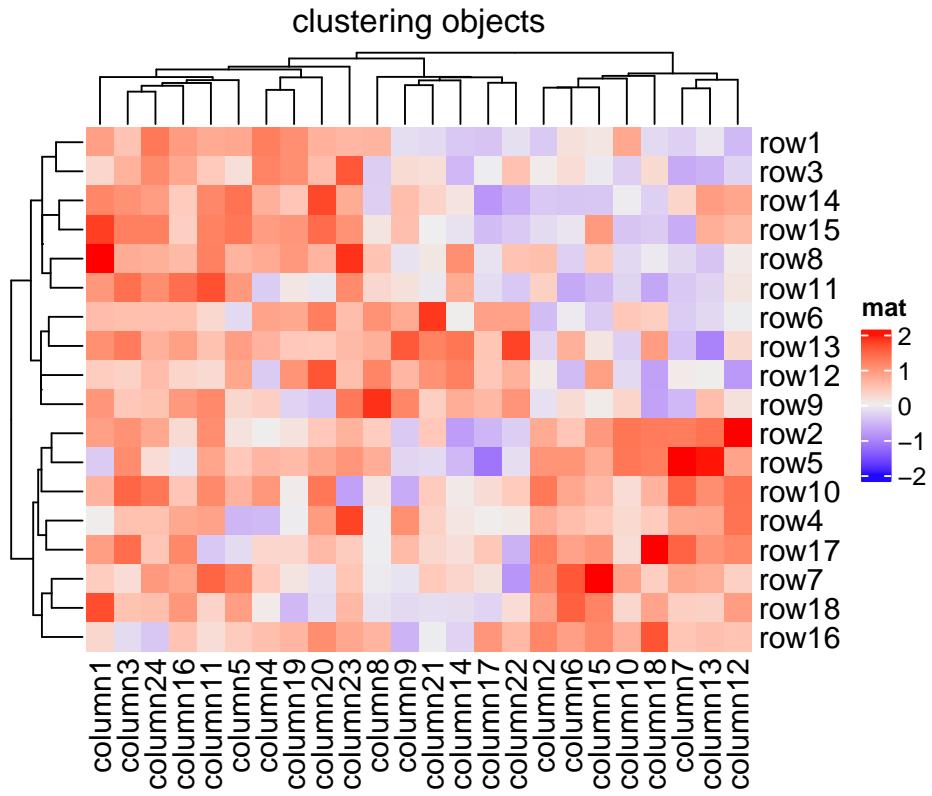
```
Heatmap(mat, name = "mat", clustering_method_rows = "single")
```



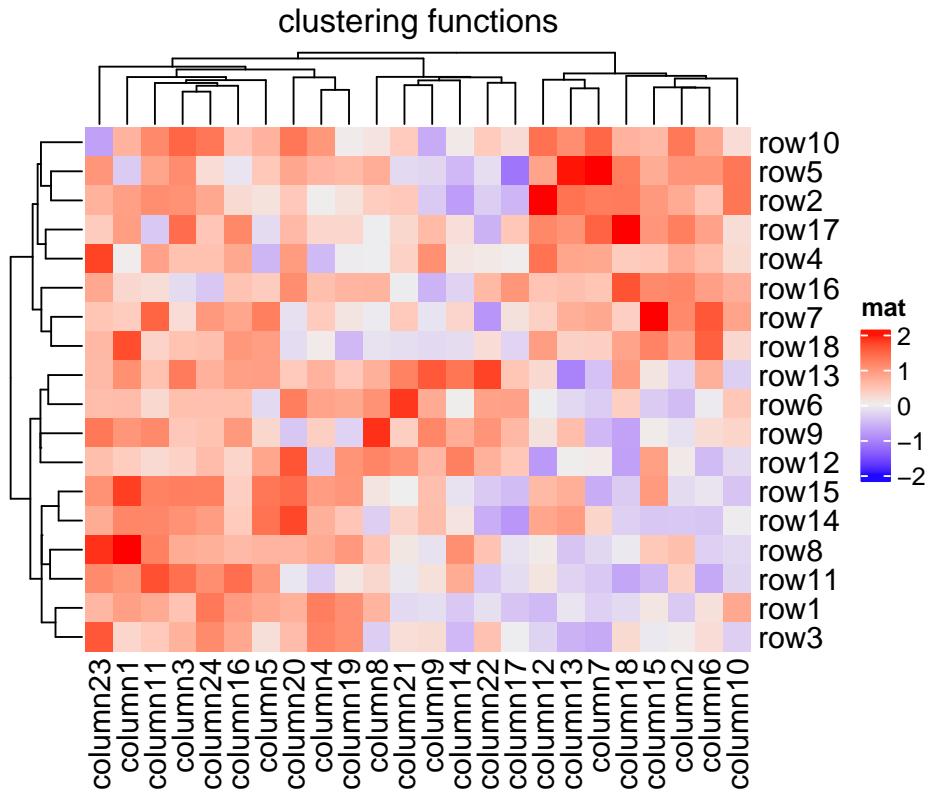
If you already have a clustering object or a function which directly returns a clustering object, you can ignore the distance settings and set `cluster_rows` or `cluster_columns` to the clustering objects or clustering functions. If it is a clustering function, the only argument should be the matrix and it should return a `hclust` or `dendrogram` object or a object that can be converted to the `dendrogram` class.

In following example, we perform clustering with methods from `cluster` package either by a pre-calculated clustering object or a clustering function:

```
library(cluster)
Heatmap(mat, name = "mat", cluster_rows = diana(mat),
        cluster_columns = agnes(t(mat)), column_title = "clustering objects")
```



```
# if cluster_columns is set as a function, you don't need to transpose the matrix
Heatmap(mat, name = "mat", cluster_rows = diana,
       cluster_columns = agnes, column_title = "clustering functions")
```



The last command is as same as :

```
# code only for demonstration
Heatmap(mat, name = "mat", cluster_rows = function(m) as.dendrogram(diana(m)),
        cluster_columns = function(m) as.dendrogram(agnes(m)),
        column_title = "clustering functions")
```

Please note, when `cluster_rows` is set as a function, the argument `m` is the input `mat` itself, while for `cluster_columns`, `m` is the transpose of `mat`.

`fastcluster::hclust` implements a faster version of `hclust()`. You can set it to `cluster_rows` and `cluster_columns` to use the faster version of `hclust()`.

```
# code only for demonstration
fh = function(x) fastcluster::hclust(dist(x))
Heatmap(mat, name = "mat", cluster_rows = fh, cluster_columns = fh)
```

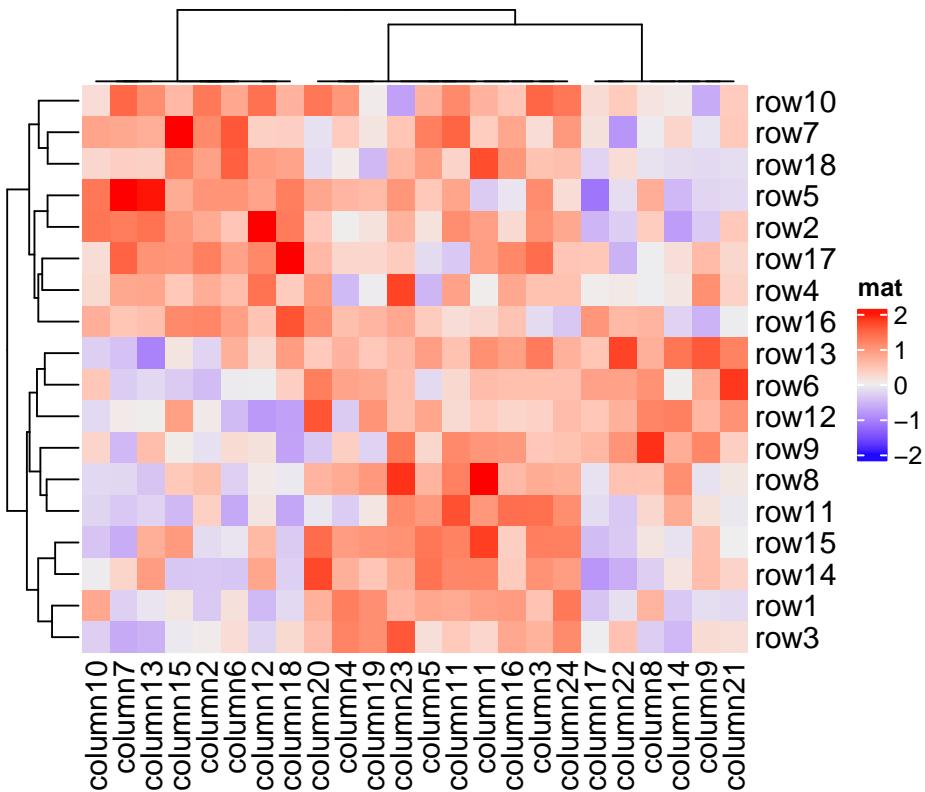
To make it more convinient to use the faster version of `hclust()` (assuming you have many heatmaps to construct), it can be set as a global option. The usage of `ht_opt` is introduced in Section 4.13.

```
# code only for demonstration
ht_opt$fast_hclust = TRUE
```

```
# now fastcluster::hclust is used in all heatmaps
```

This is one special scenario where you might already have a subgroup classification for the matrix rows or columns, and you only want to perform clustering for the features in the same subgroup. You can split the heatmap by the subgroup variable (see Section 2.7), or you can use `cluster_within_group()` clustering function to generate a special dendrogram.

```
group = kmeans(t(mat), centers = 3)$cluster
Heatmap(mat, name = "mat", cluster_columns = cluster_within_group(mat, group))
```



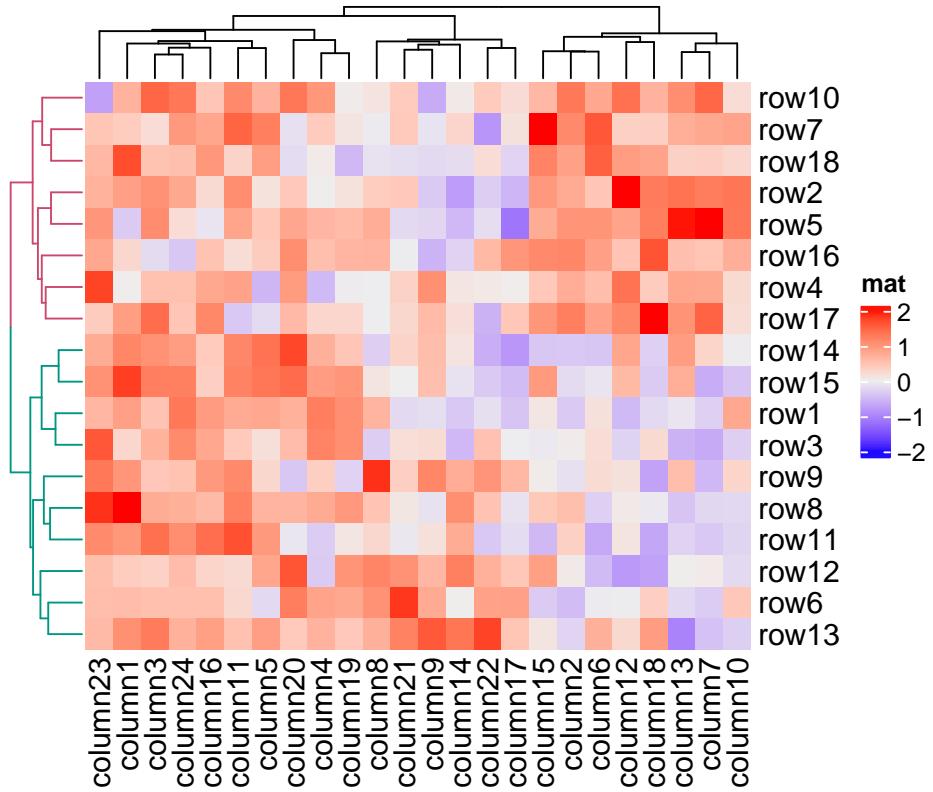
In above example, columns in a same group are still clustered, but the dendrogram is degenerated as a flat line. The dendrogram on columns shows the hierarchy of the groups.

2.3.3 Render dendrograms

If you want to render the dendrogram, normally you need to generate a `dendrogram` object and render it through `nodePar` and `edgePar` parameter in advance, then send it to the `cluster_rows` or `cluster_columns` argument.

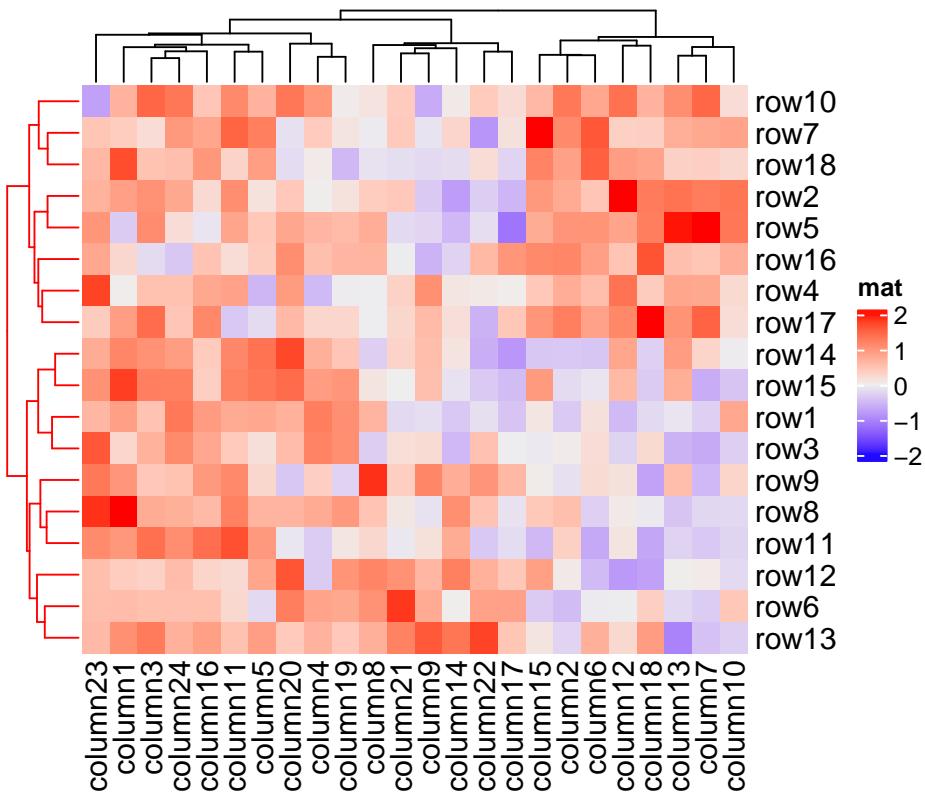
You can render your `dendrogram` object by the `dendextend` package to make a more customized visualization of the dendrogram.

```
library(dendextend)
row_dend = as.dendrogram(hclust(dist(mat)))
row_dend = color_branches(row_dend, k = 2) # `color_branches()` `returns a dendrogram o
Heatmap(mat, name = "mat", cluster_rows = row_dend)
```



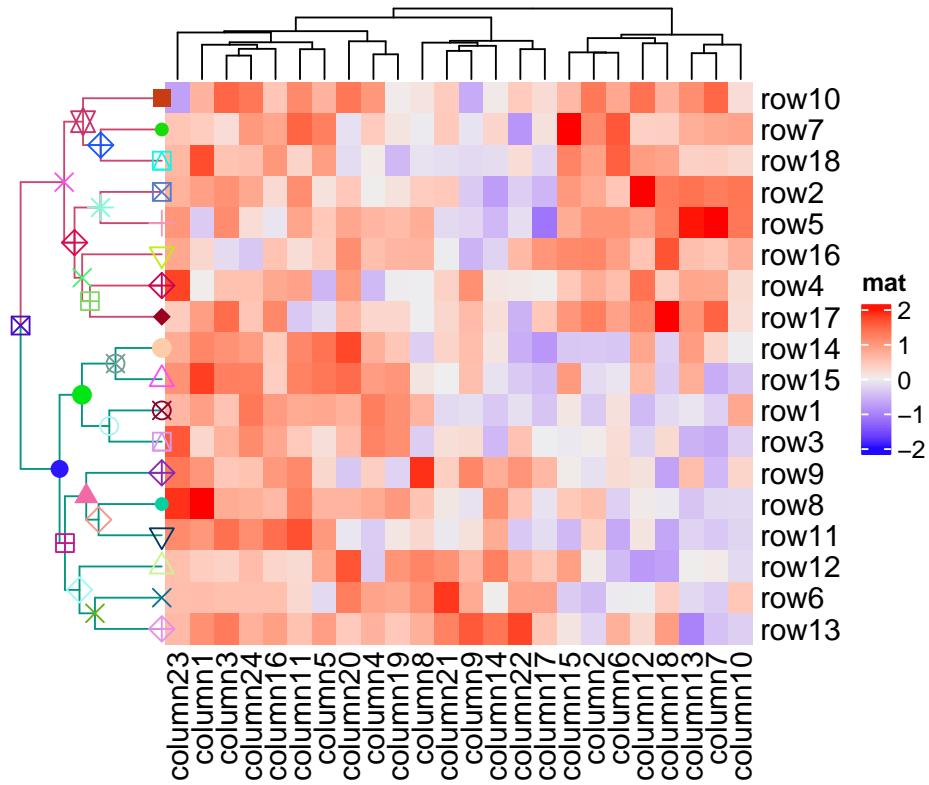
`row_dend_gp` and `column_dend_gp` control the global graphics setting for dendograms. Note e.g. graphics settings in `row_dend` will be overwritten by `row_dend_gp`.

```
Heatmap(mat, name = "mat", cluster_rows = row_dend,
        row_dend_gp = gpar(col = "red"))
```



From version 2.5.6, you can also add graphics on the nodes of the dendrogram by setting a proper `nodePar`.

```
row_dend = dendrapply(row_dend, function(d) {
  attr(d, "nodePar") = list(cex = 0.8, pch = sample(20, 1), col = rand_color(1))
  return(d)
})
Heatmap(mat, name = "mat", cluster_rows = row_dend, row_dend_width = unit(2, "cm"))
```



Currently, only points can be added on the nodes, however, if you have the need to add more types of graphics, especially the self-defined graphics, just send me a message and I will think about it.

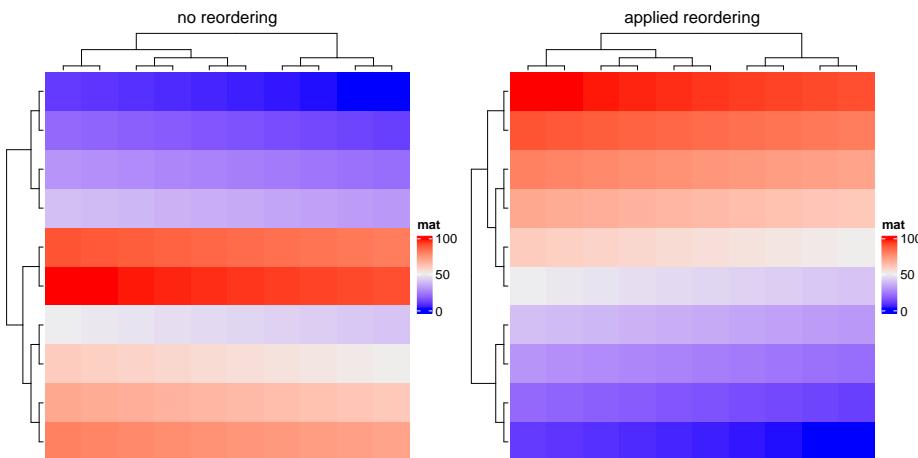
2.3.4 Reorder dendograms

In the `Heatmap()` function, dendograms are reordered to make features with larger difference more separated from each others (please refer to the documentation of `reorder.dendrogram()`). Here the difference (or it is called the weight) is measured by the row means if it is a row dendrogram or by the column means if it is a column dendrogram. `row_dend_reorder` and `column_dend_reorder` control whether to apply dendrogram reordering if the value is set as logical. The two arguments also control the weight for the reordering if they are set to numeric vectors (it will be sent to the `wts` argument of `reorder.dendrogram()`). The reordering can be turned off by setting e.g. `row_dend_reorder = FALSE`.

By default, dendrogram reordering is turned on if `cluster_rows/cluster_columns` is set as logical value or a clustering function. It is turned off if `cluster_rows/cluster_columns` is set as clustering object.

Compare following two heatmaps:

```
m2 = matrix(1:100, nr = 10, byrow = TRUE)
Heatmap(m2, name = "mat", row_dend_reorder = FALSE, column_title = "no reordering")
Heatmap(m2, name = "mat", row_dend_reorder = TRUE, column_title = "apply reordering")
```

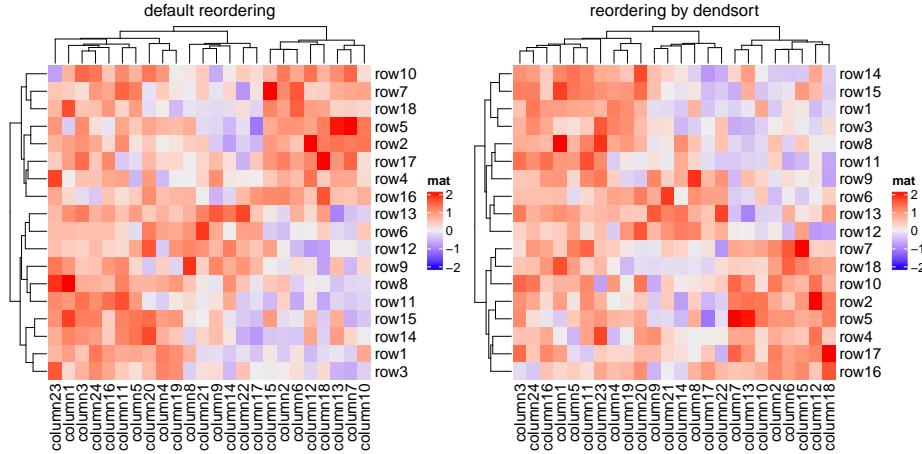


There are many other methods for reordering dendograms, e.g. the **dendsort** package. Basically, all these methods still return a dendrogram that has been reordered, thus, we can firstly generate the row or column dendrogram based on the data matrix, reorder it by a certain method, and assign it back to `cluster_rows` or `cluster_columns`.

Compare following two reorderings on both dimensions. Can you tell which is better?

```
Heatmap(mat, name = "mat", column_title = "default reordering")

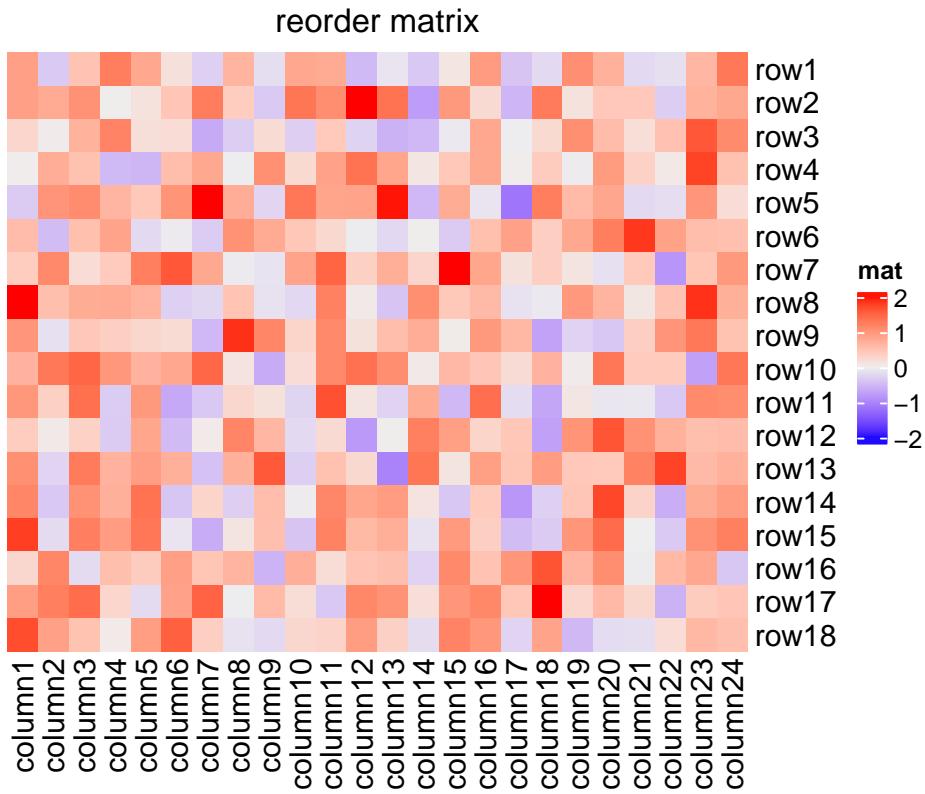
library(dendsort)
row_dend = dendsort(hclust(dist(mat)))
col_dend = dendsort(hclust(dist(t(mat))))
Heatmap(mat, name = "mat", cluster_rows = row_dend, cluster_columns = col_dend,
        column_title = "reorder by dendsort")
```



2.4 Set row and column orders

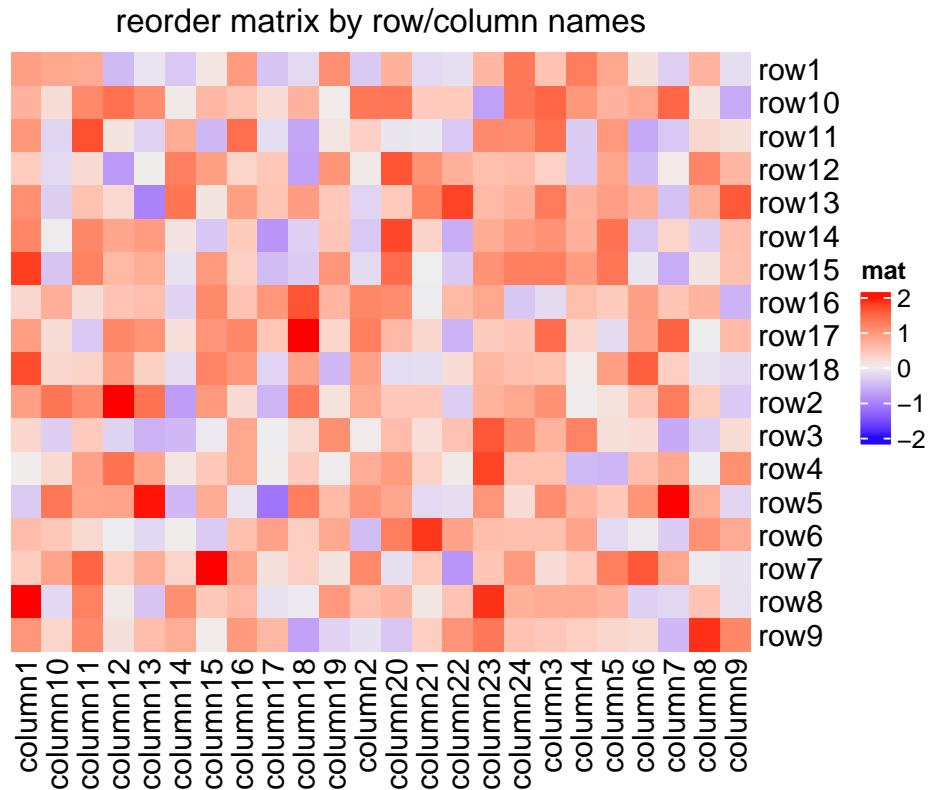
Clustering is used to adjust row orders and column orders of the heatmap, but you can still set the order manually by `row_order` and `column_order`. If e.g. `row_order` is set, row clustering is turned off by default.

```
Heatmap(mat, name = "mat",
        row_order = order(as.numeric(gsub("row", "", rownames(mat)))),
        column_order = order(as.numeric(gsub("column", "", colnames(mat)))),
        column_title = "reorder matrix")
```



The orders can be character vectors if they are just shuffles of the matrix row names or column names.

```
Heatmap(mat, name = "mat", row_order = sort(rownames(mat)),
        column_order = sort(colnames(mat)),
        column_title = "reorder matrix by row/column names")
```



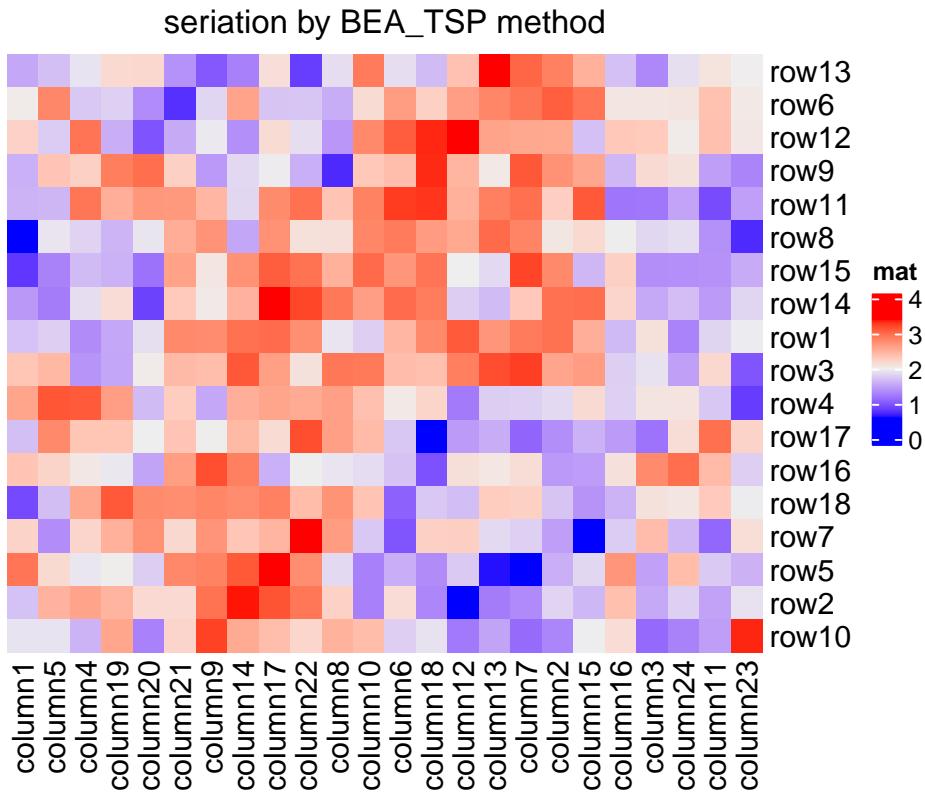
Note `row_dend_reorder` and `row_order` are two different things. `row_dend_reorder` is applied on the dendrogram. For any node in the dendrogram, rotating its two branches actually gives an identical dendrogram, thus, reordering the dendrogram by automatically rotating sub-dendrogram at every node can help to separate features further from each other which show more difference. As a comparison, `row_order` is simply applied to the matrix and normally dendograms should be turned off.

2.5 Seriation

Seriation is an interesting technique for ordering the matrix (see this interesting post: <http://nicolas.kruchten.com/content/2018/02/seriation/>). The powerful **seriation** package implements quite a lot of methods for seriation. Since it is easy to extract row orders and column orders from the object returned by the core function `seriate()` from **seriation** package, they can be directly assigned to `row_order` and `column_order` to make the heatmap.

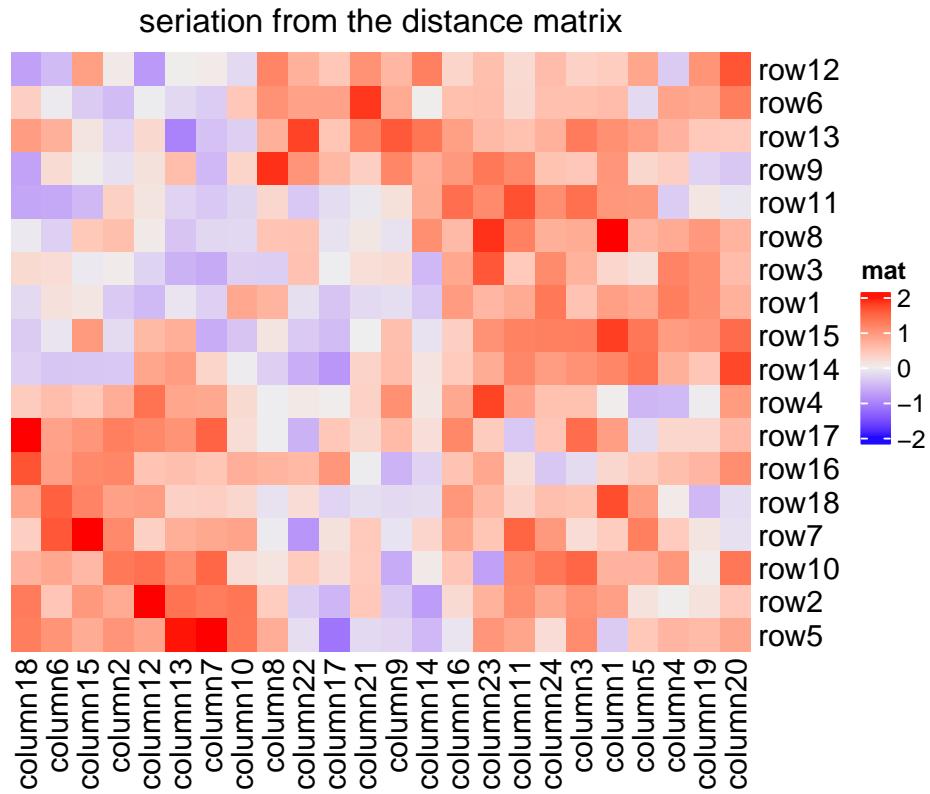
The first example demonstrates directly applying `seriate()` on the matrix. Since the "BEA_TSP" method only allows a non-negative matrix, we modify the matrix to `max(mat) - mat`.

```
library(seriation)
o = seriate(max(mat) - mat, method = "BEA_TSP")
Heatmap(max(mat) - mat, name = "mat",
       row_order = get_order(o, 1), column_order = get_order(o, 2),
       column_title = "seriation by BEA_TSP method")
```



Or you can apply `seriate()` to the distance matrix. Now the order for rows and columns needs to be calculated separately because the distance matrix needs to be calculated separately for columns and rows.

```
o1 = seriate(dist(mat), method = "TSP")
o2 = seriate(dist(t(mat)), method = "TSP")
Heatmap(mat, name = "mat", row_order = get_order(o1), column_order = get_order(o2),
       column_title = "seriation from the distance matrix")
```



Some seriation methods also contain the hierarchical clustering information.
Let's try:

```
o1 = seriate(dist(mat), method = "GW")
o2 = seriate(dist(t(mat)), method = "GW")
```

`o1` and `o2` are actually mainly composed of `hclust` objects:

```
class(o1[[1]])
```

```
## [1] "ser_permutation_vector" "hclust"
```

And the orders are the same by using `hclust$order` or `get_order()`.

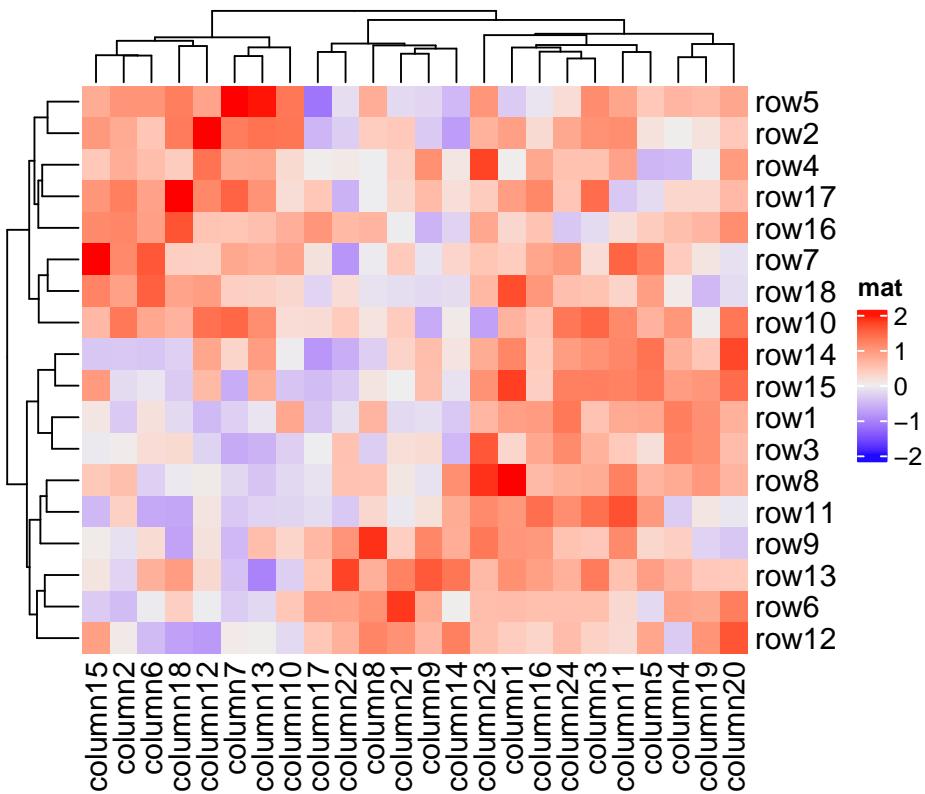
```
o1[[1]]$order
```

```
## [1] 5 2 4 17 16 7 18 10 14 15 1 3 8 11 9 13 6 12
# should be the same as the previous one
get_order(o1)
```

```
## [1] 5 2 4 17 16 7 18 10 14 15 1 3 8 11 9 13 6 12
```

And we can add the dendograms to the heatmap.

```
Heatmap(mat, name = "mat", cluster_rows = as.dendrogram(o1[[1]]),
       cluster_columns = as.dendrogram(o2[[1]]))
```

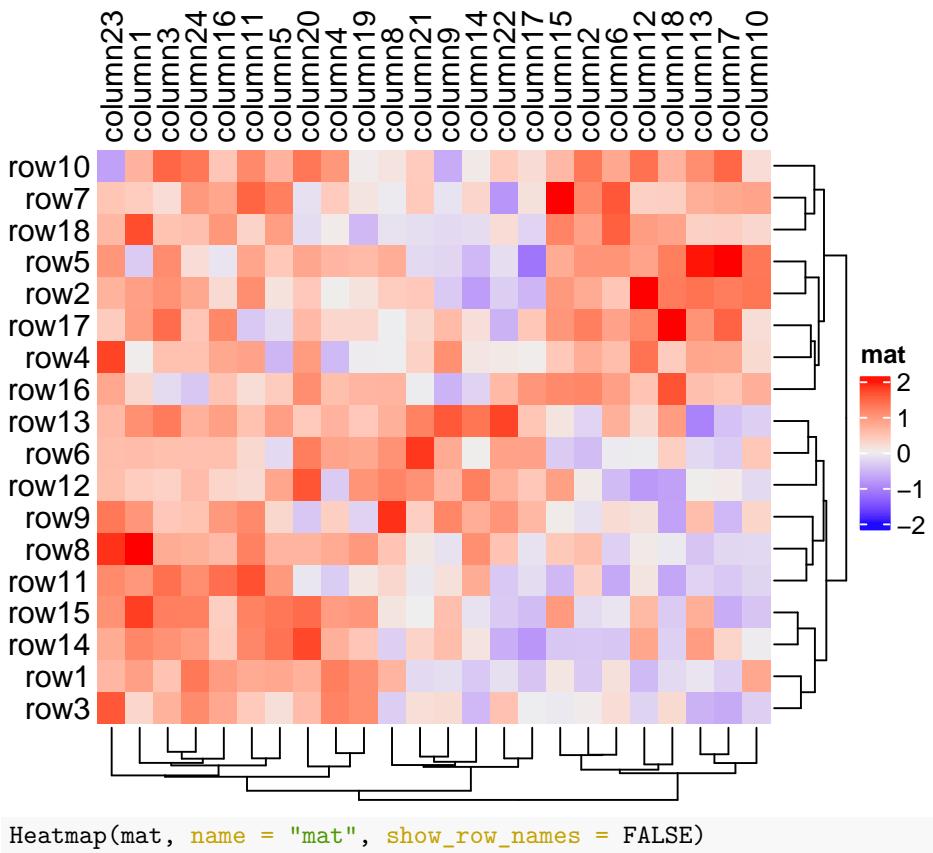


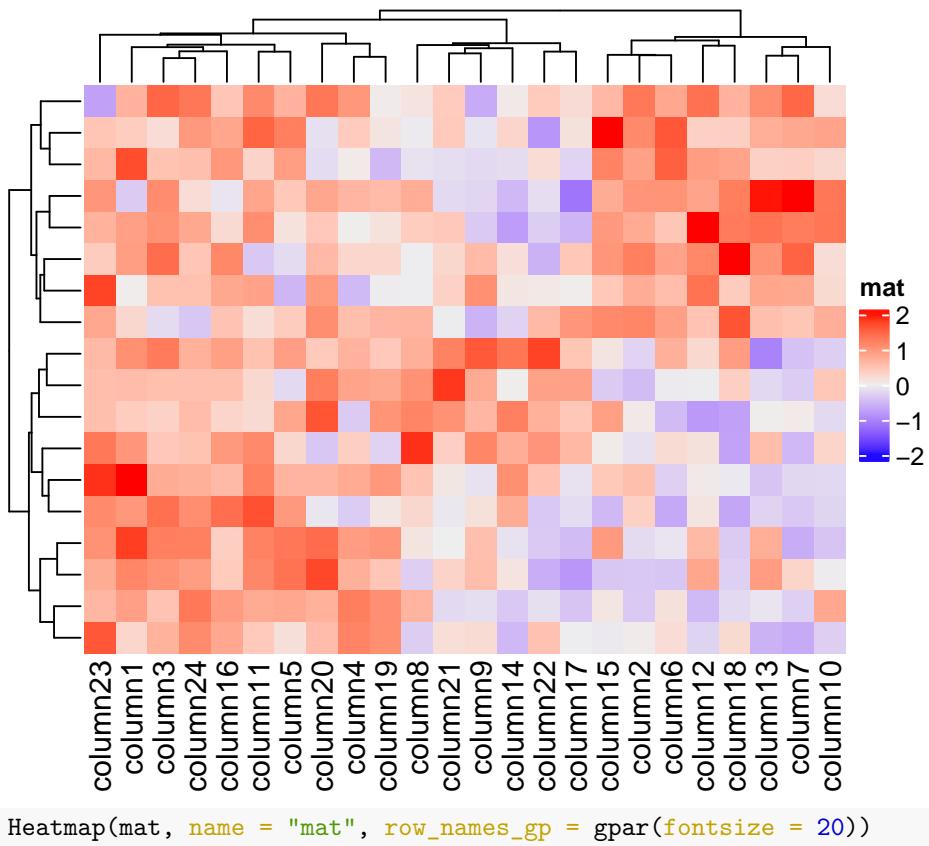
For more use of the `seriate()` function, please refer to the **seriation** package.

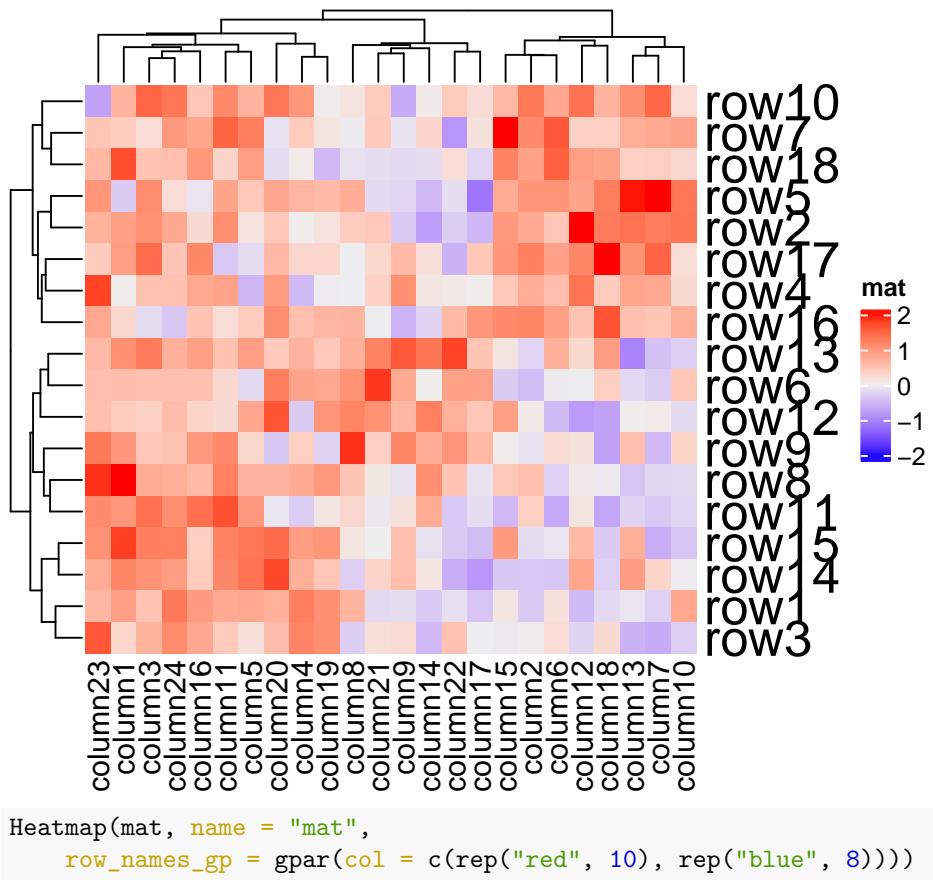
2.6 Dimension labels

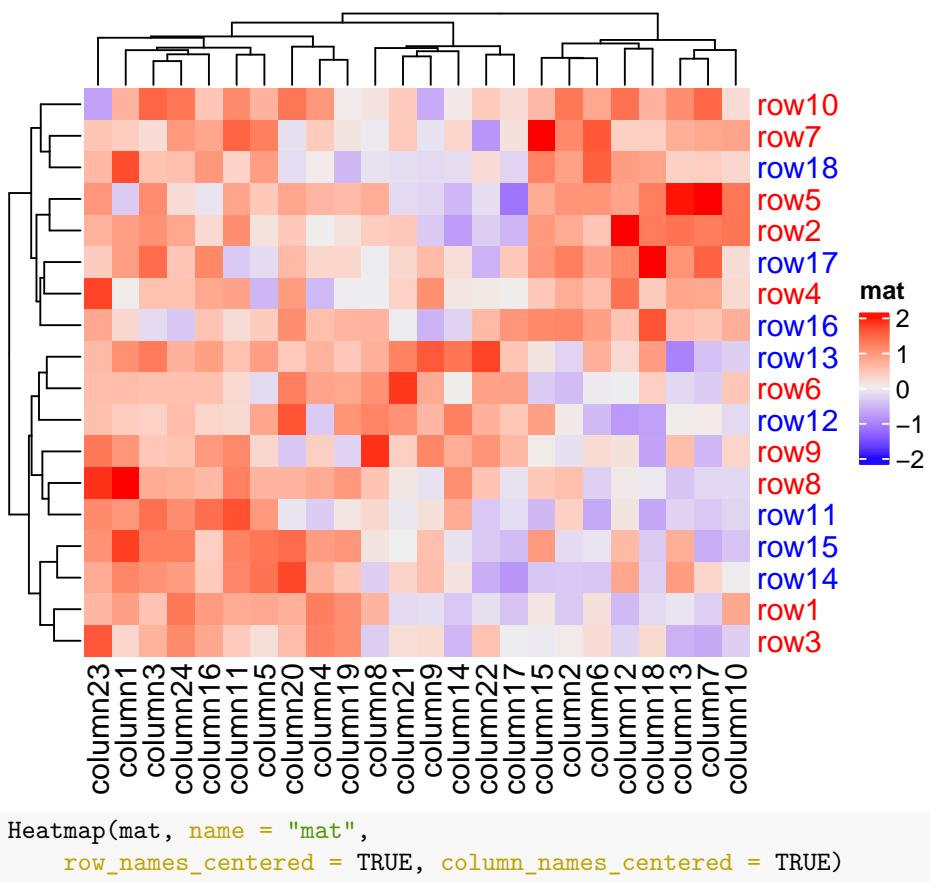
The row names and column names are drawn on the right and bottom sides of the heatmap by default. Side, visibility and graphics parameters for dimension names can be set as follows:

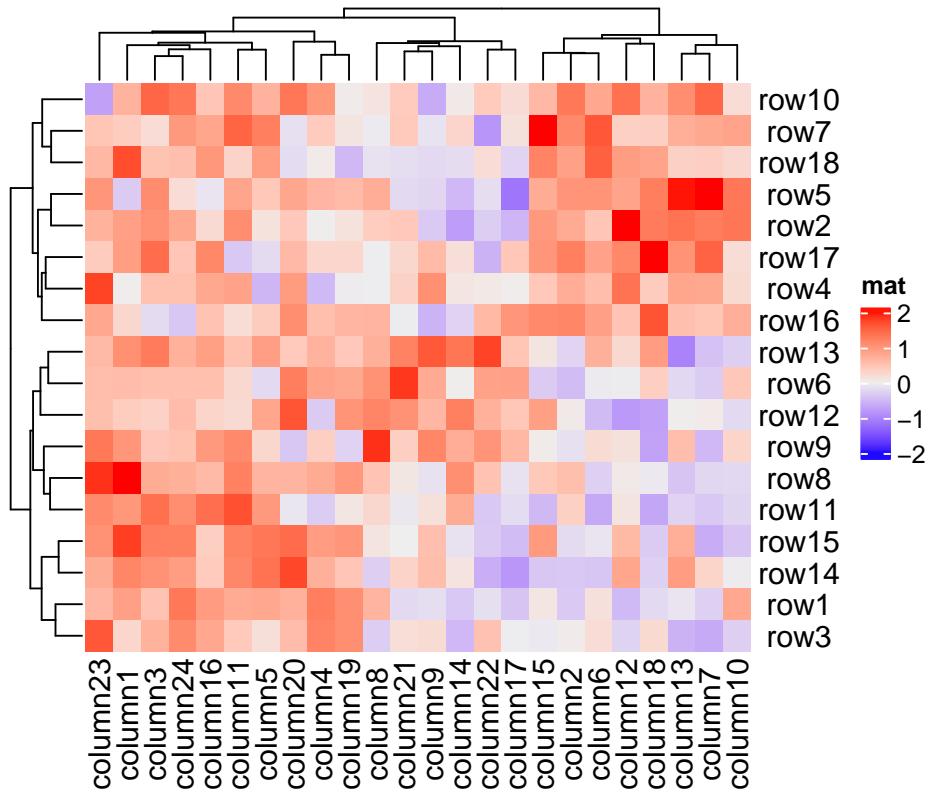
```
Heatmap(mat, name = "mat", row_names_side = "left", row_dend_side = "right",
       column_names_side = "top", column_dend_side = "bottom")
```





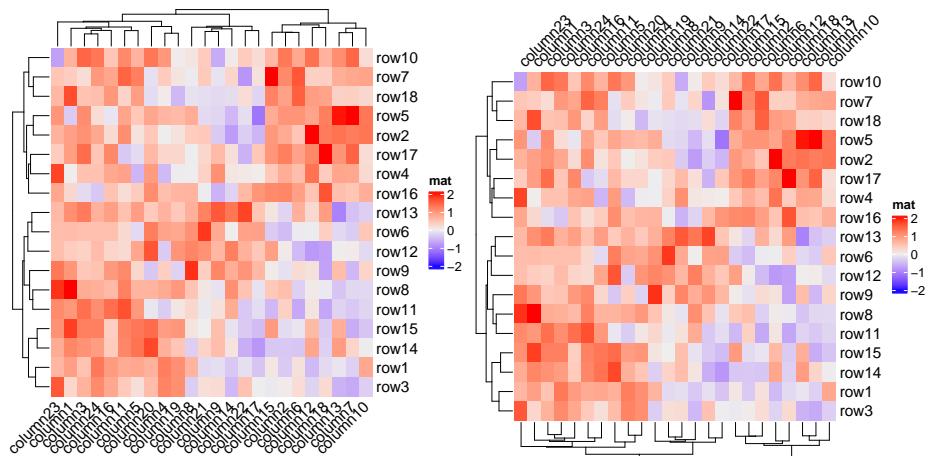






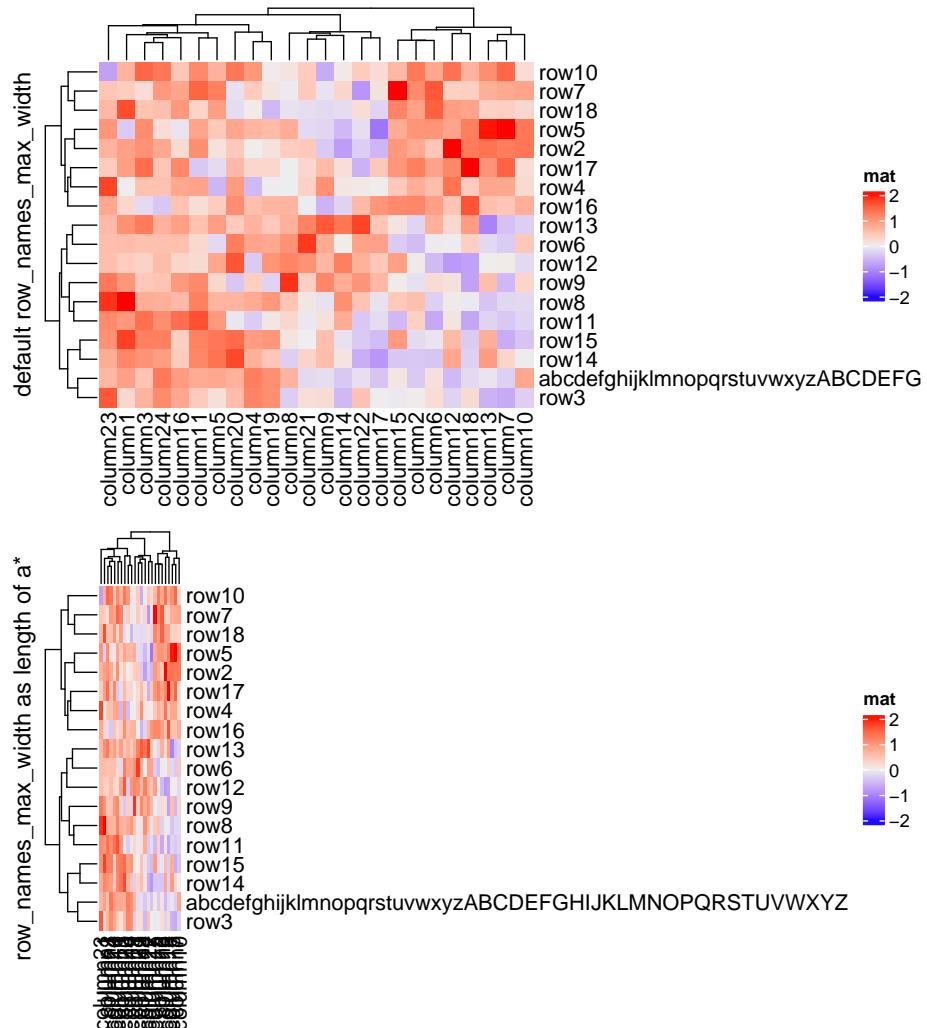
The rotation of column names can be set by `column_names_rot`:

```
Heatmap(mat, name = "mat", column_names_rot = 45)
Heatmap(mat, name = "mat", column_names_rot = 45, column_names_side = "top",
        column_dend_side = "bottom")
```



If you have row names or column names which are too long, `row_names_max_width` or `column_names_max_height` can be used to set the maximal space for them. The default maximal space for row names and column names are all 6 cm. In following code, `max_text_width()` is a helper function to quick calculate maximal width from a vector of text.

```
mat2 = mat
rownames(mat2)[1] = paste(c(letters, LETTERS), collapse = "")
Heatmap(mat2, name = "mat", row_title = "default row_names_max_width")
Heatmap(mat2, name = "mat", row_title = "row_names_max_width as length of a*",
        row_names_max_width = max_text_width(
            rownames(mat2),
            gp = gpar(fontsize = 12)
        ))
```



Instead of directly using the row/column names from the matrix, you can also provide another character vector which corresponds to the rows or columns and set it by `row_labels` or `column_labels`. This is useful because you don't need to change the dimension names of the matrix to change the labels on the heatmap while you can directly provide the new labels.

There is one typical scenario that `row_labels` and `column_labels` are useful. For the gene expression analysis, we might use Ensembl ID as the gene ID which is used as row names of the gene expression matrix. However, the Ensembl ID is for the indexing of the Ensembl database but not for the human reading. Instead, we would prefer to put gene symbols on the heatmap as the row names which is easier to read. To do this, we only need to assign the corresponding gene symbols to `row_labels` without modifying the original matrix.

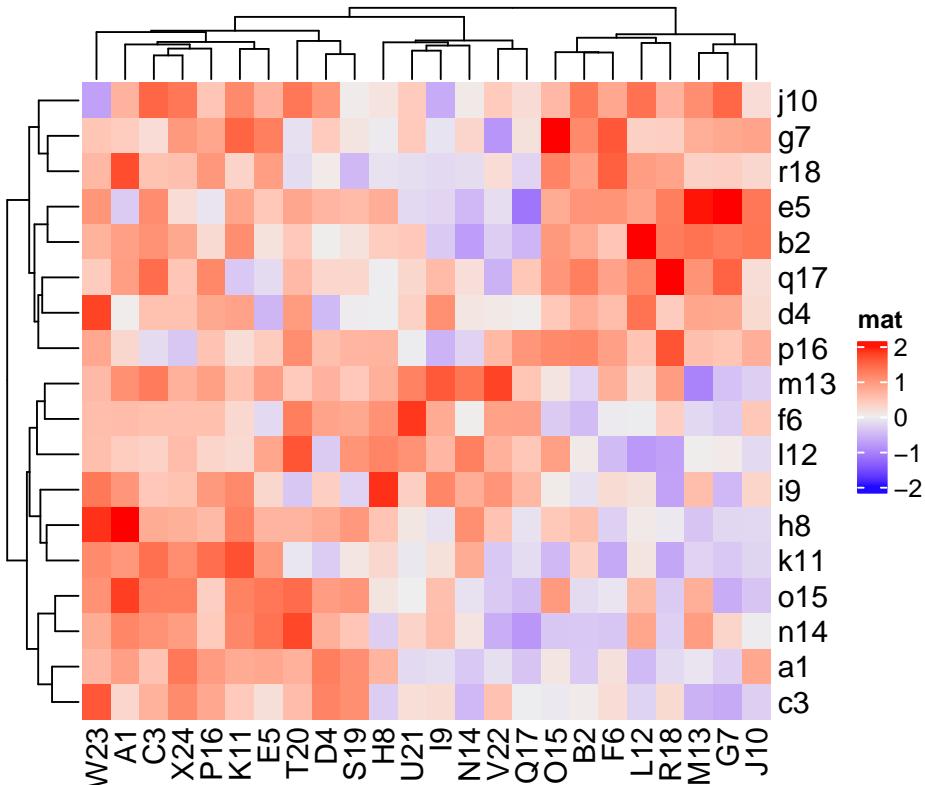
The second advantage is `row_labels` or `column_labels` allows duplicated labels, while duplicated row names or column names are not allowed in the matrix.

Following gives a simple example that we put letters as row labels and column labels:

```
# use a named vector to make sure the correspondance between
# row names and row labels is correct
row_labels = structure(paste0(letters[1:24], 1:24), names = paste0("row", 1:24))
column_labels = structure(paste0(LETTERS[1:24], 1:24), names = paste0("column", 1:24))
row_labels

## row1  row2  row3  row4  row5  row6  row7  row8  row9  row10  row11  row12  row13
## "a1"  "b2"  "c3"  "d4"  "e5"  "f6"  "g7"  "h8"  "i9"  "j10"  "k11"  "l12"  "m13"
## row14  row15  row16  row17  row18  row19  row20  row21  row22  row23  row24
## "n14"  "o15"  "p16"  "q17"  "r18"  "s19"  "t20"  "u21"  "v22"  "w23"  "x24"

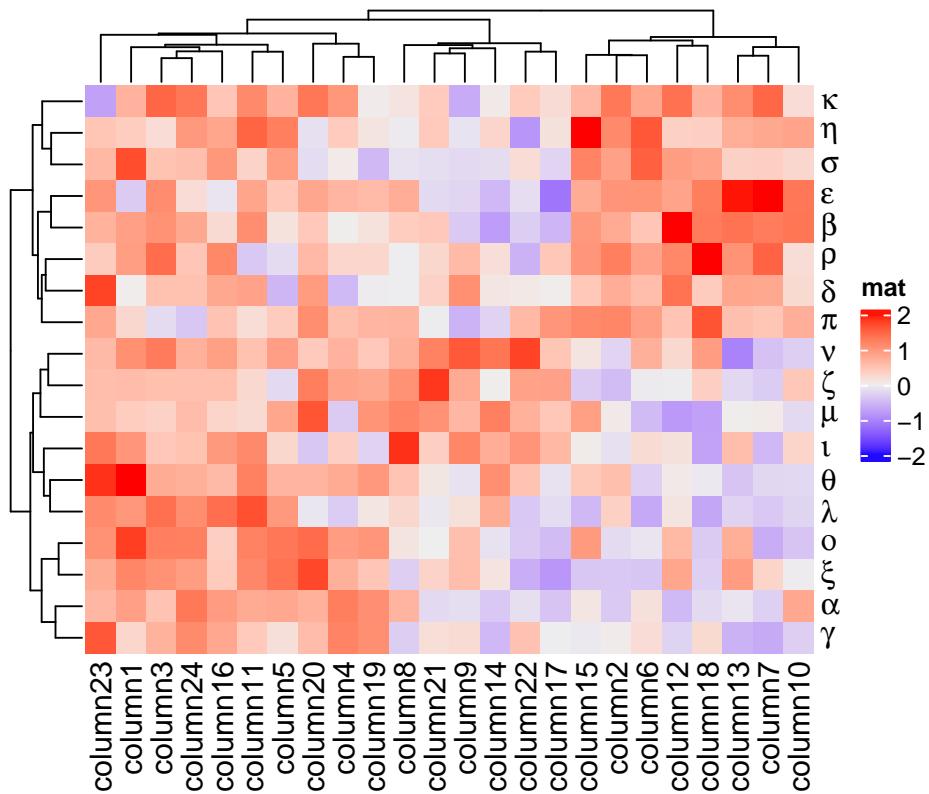
Heatmap(mat, name = "mat", row_labels = row_labels[rownames(mat)],
        column_labels = column_labels[colnames(mat)])
```



The third advantage is mathematical expression can be used as row names in

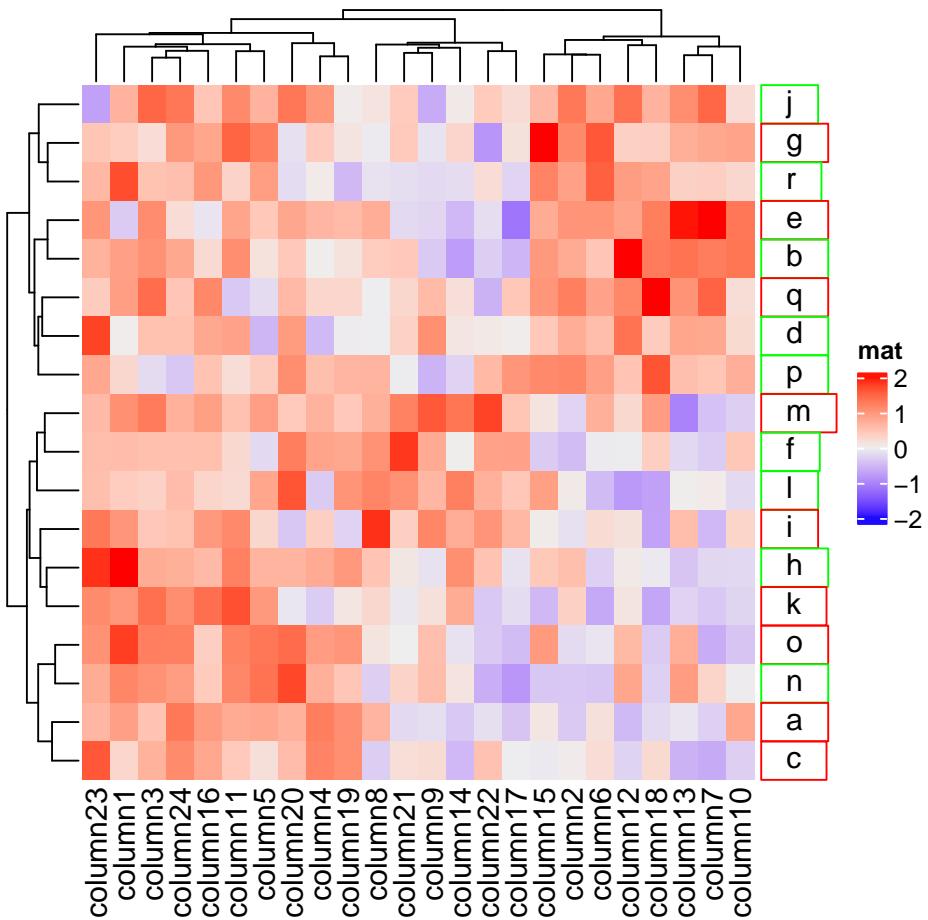
the heatmap.

```
Heatmap(mat, name = "mat", row_labels = expression(alpha, beta, gamma,
    delta, epsilon, zeta, eta, theta, iota, kappa, lambda, mu, nu, xi,
    omicron, pi, rho, sigma)
)
```



The fourth advantage is that you can construct complicated labels with `gridtext` package. The following is a quick example. More examples can be found in Section 10.3.1.

```
Heatmap(mat, name = "mat",
    row_labels = gt_render(letters[1:18], padding = unit(c(2, 10, 2, 10), "pt")),
    row_names_gp = gpar(box_col = rep(c("red", "green"), times = 9)))
```



Internally, row names and columns names are actually implemented by the `anno_text()` function (Section 3.14), in other words, they are treated as special cases for text annotations.

2.7 Heatmap split

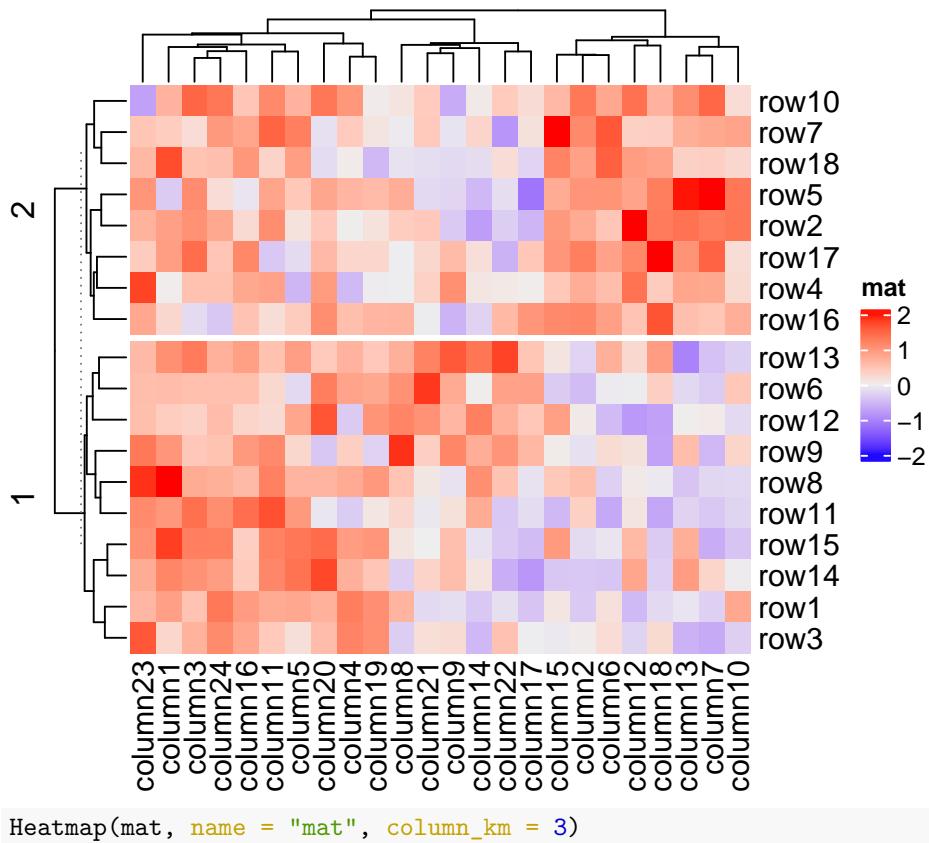
One major advantage of **ComplexHeatmap** package is that it supports splitting the heatmap by rows and columns to better group the features and additionally highlight the patterns.

Following arguments control the splitting: `row_km`, `row_split`, `column_km`, `column_split`. In following, we call the sub-clusters generated by splitting as “*slices*.”

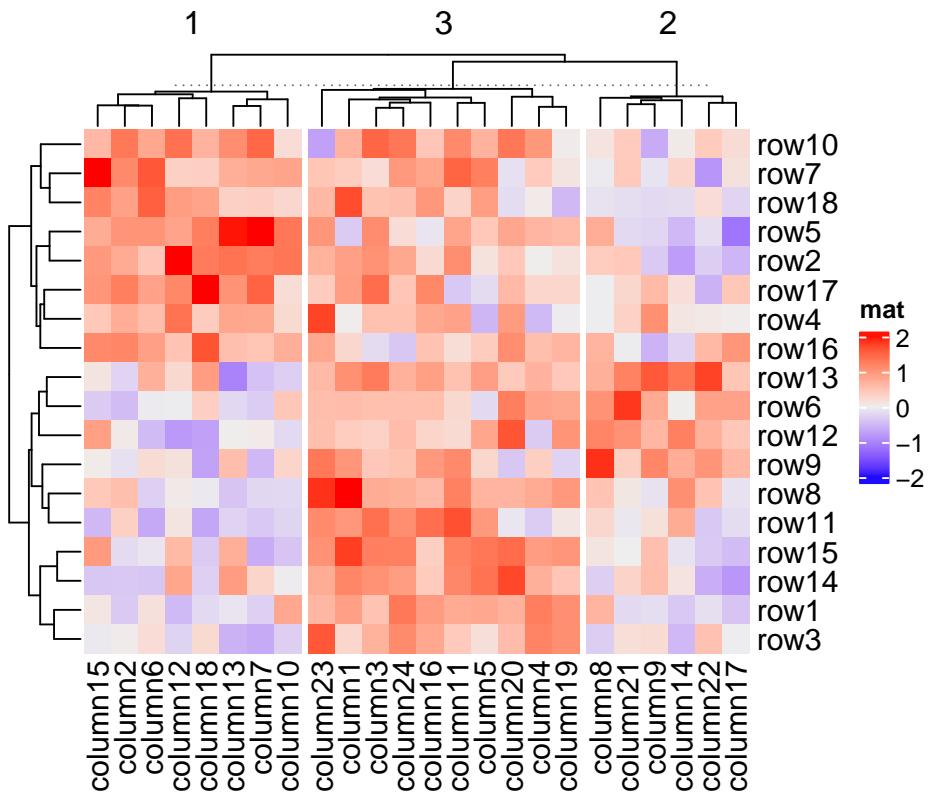
2.7.1 Split by k-means clustering

`row_km` and `column_km` apply k-means partitioning.

```
Heatmap(mat, name = "mat", row_km = 2)
```

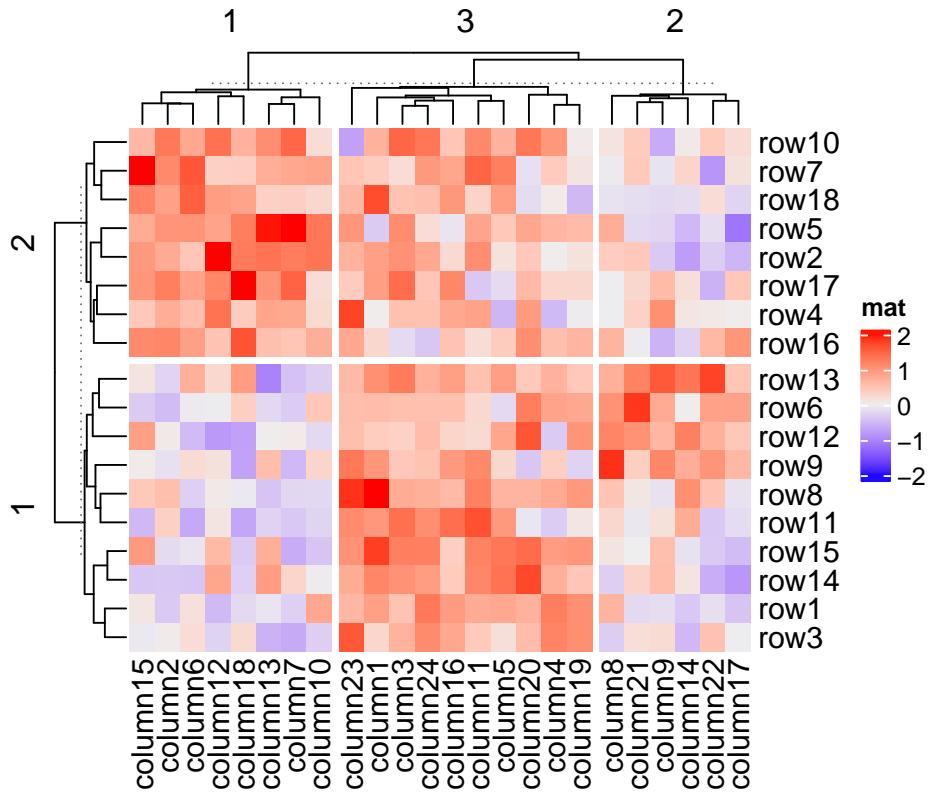


```
Heatmap(mat, name = "mat", column_km = 3)
```



Row splitting and column splitting can be performed simultaneously.

```
Heatmap(mat, name = "mat", row_km = 2, column_km = 3)
```



You might notice there are dashed lines in the row and column dendograms, it will be explained in Section 2.7.2 (the last paragraph).

`Heatmap()` internally calls `kmeans()` with random start points, which results in, for some cases, generating different clusters from repeated runs. To get rid of this problem, `row_km_repeats` and `column_km_repeats` can be set to a number larger than 1 to run `kmeans()` multiple times and a final consensus k-means clustering is used. Please note the final number of clusters form consensus k-means might be smaller than the number set in `row_km` and `column_km`.

```
# of course, it will be a little bit slower
Heatmap(mat, name = "mat",
        row_km = 2, row_km_repeats = 100,
        column_km = 3, column_km_repeats = 100)
```

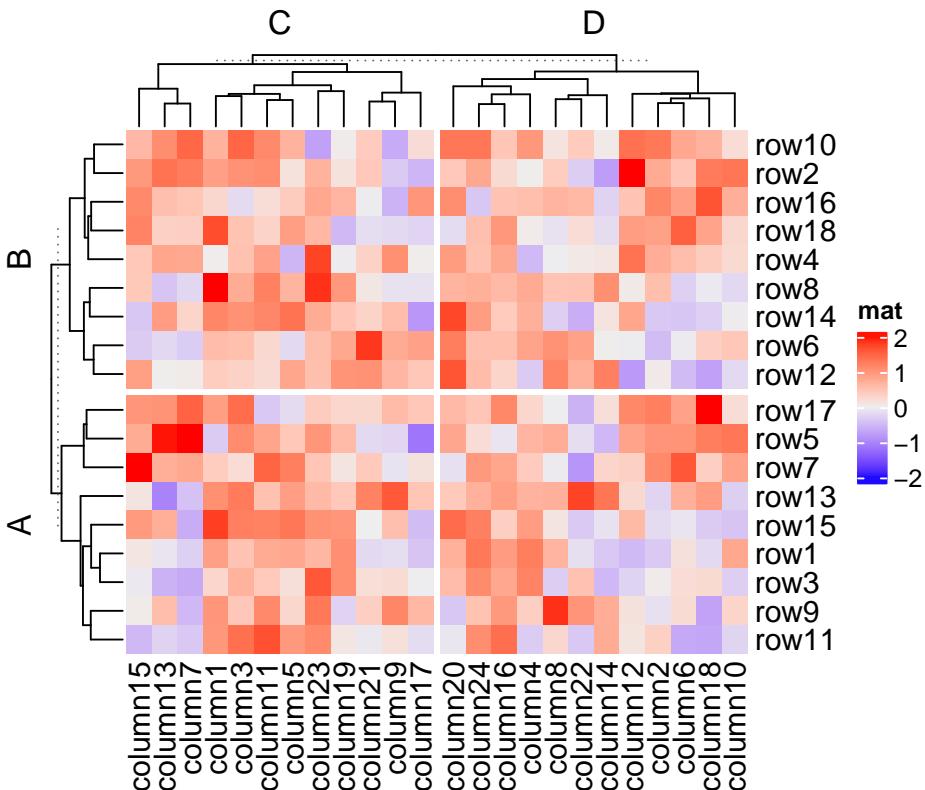
Note if you want the k-means clustering completely reproducible, you must explicitly set the same random seed:

```
set.seed(123)
Heatmap(..., row_km = ..., column_km = ...)
```

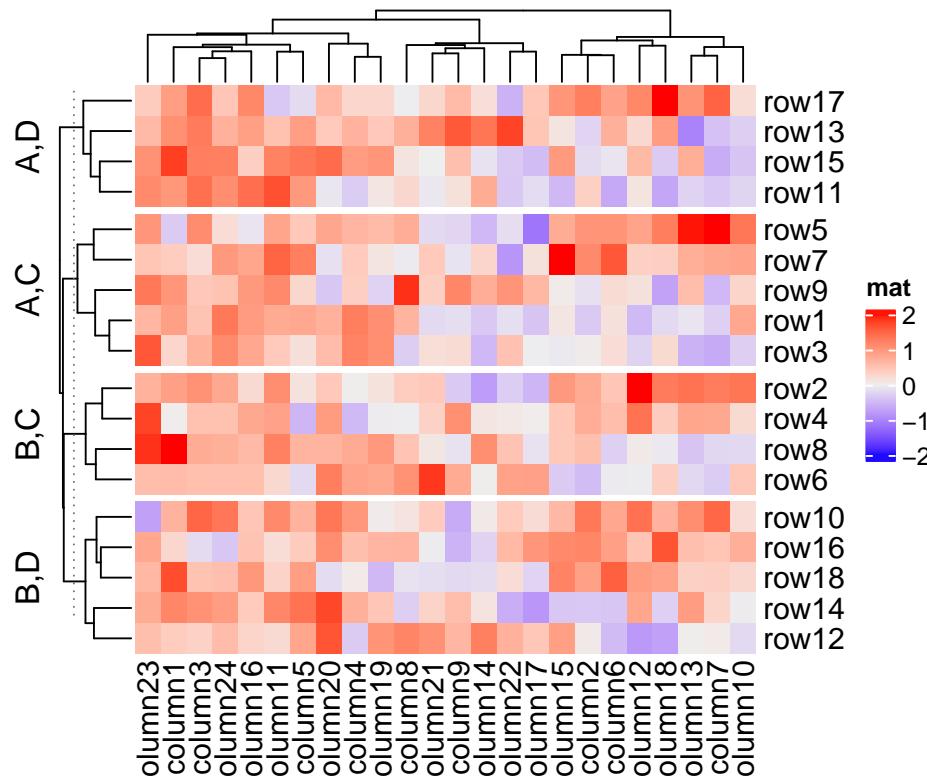
2.7.2 Split by categorical variables

More generally, `row_split` or `column_split` can be set to a categorical vector or a data frame where different combinations of levels split the rows/columns in the heatmap. How to control the order of the slices is introduced in Section 2.7.4.

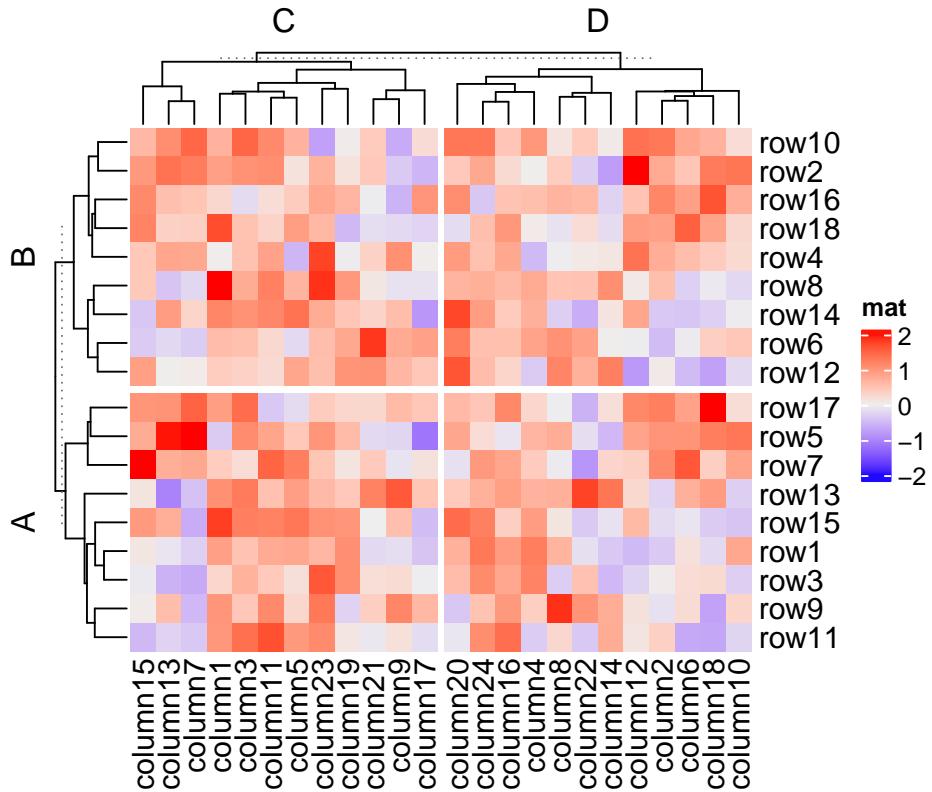
```
# split by a vector
Heatmap(mat, name = "mat",
        row_split = rep(c("A", "B"), 9), column_split = rep(c("C", "D"), 12))
```



```
# split by a data frame
Heatmap(mat, name = "mat",
        row_split = data.frame(rep(c("A", "B"), 9), rep(c("C", "D"), each = 9)))
```

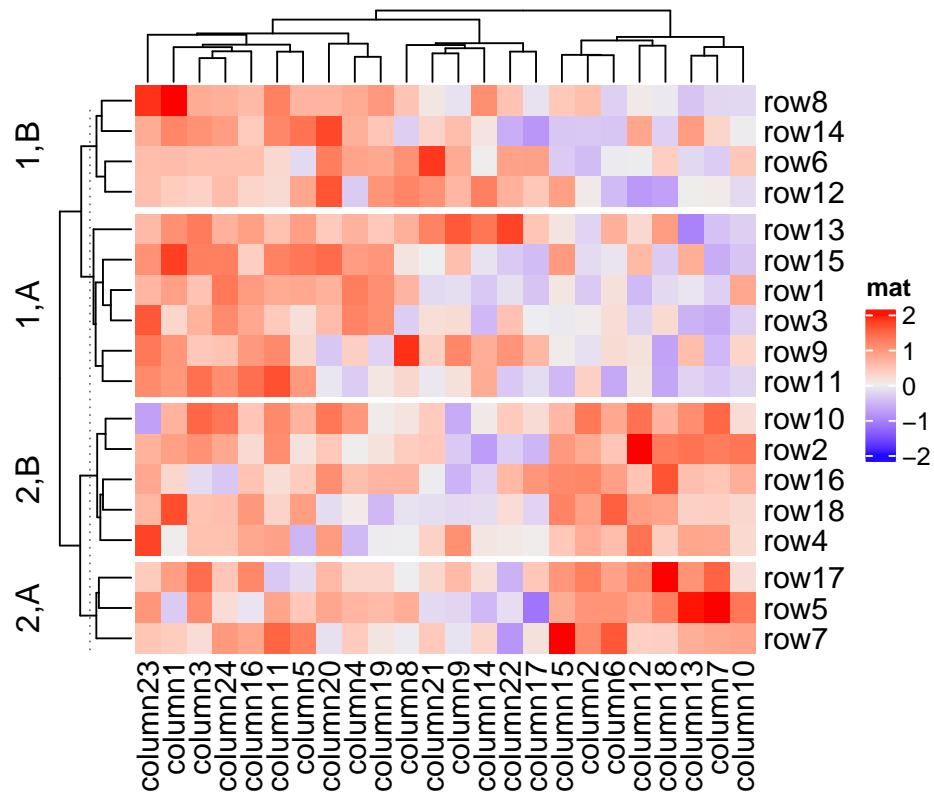


```
# split on both dimensions
Heatmap(mat, name = "mat", row_split = factor(rep(c("A", "B"), 9)),
        column_split = factor(rep(c("C", "D"), 12)))
```



Actually, k-means clustering just generates a vector of cluster classes and appends to `row_split` or `column_split`. `row_km/column_km` can be used mixed with `row_split` and `column_split`.

```
Heatmap(mat, name = "mat", row_split = rep(c("A", "B"), 9), row_km = 2)
```

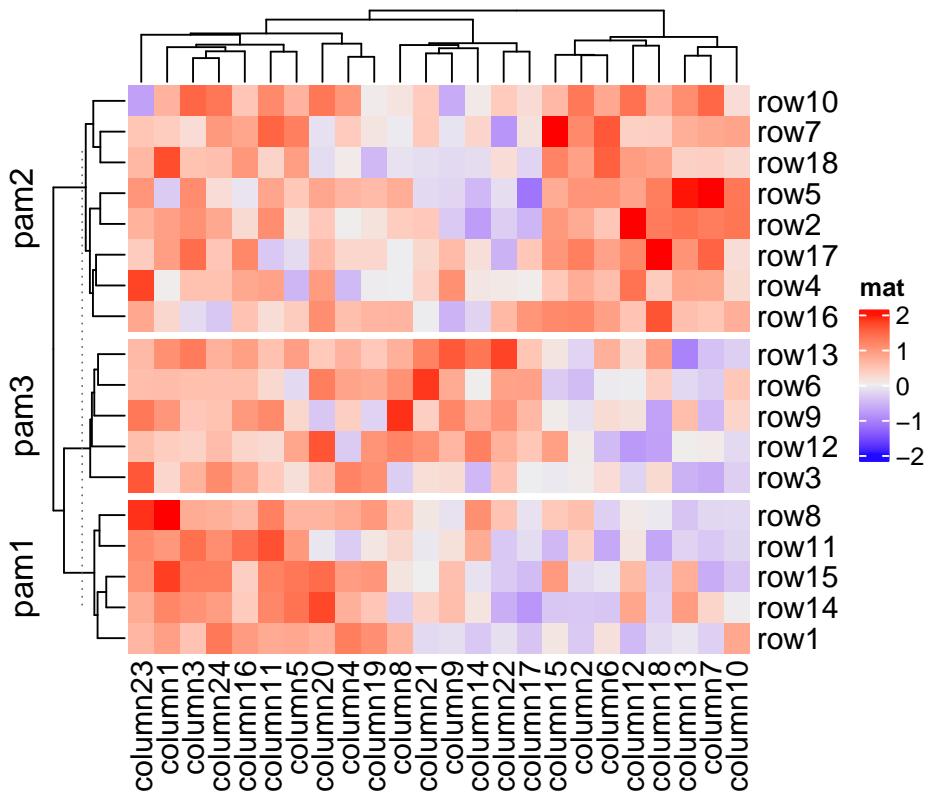


which is the same as:

```
# code only for demonstration
cl = kmeans(mat, centers = 2)$cluster
# classes from k-means are always put as the first column in `row_split`#
Heatmap(mat, name = "mat", row_split = cbind(cl, rep(c("A", "B"), 9)))
```

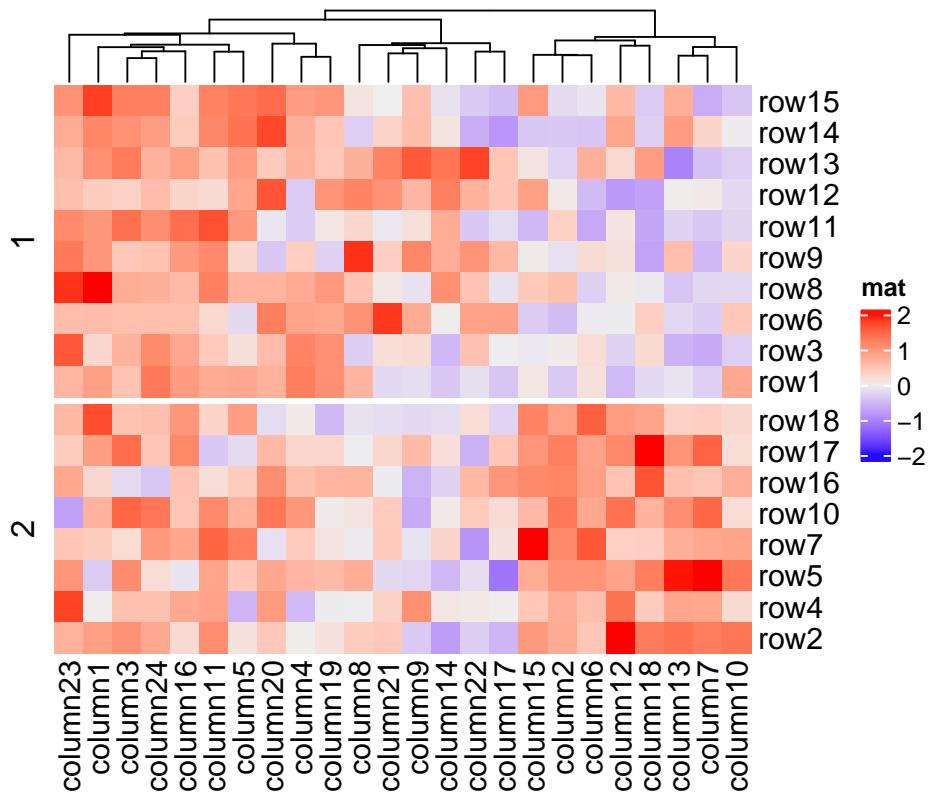
If you are not happy with the default k-means partition, it is easy to use other partition methods by just assigning the partition vector to `row_split/column_split`.

```
pa = cluster::pam(mat, k = 3)
Heatmap(mat, name = "mat", row_split = paste0("pam", pa$clustering))
```



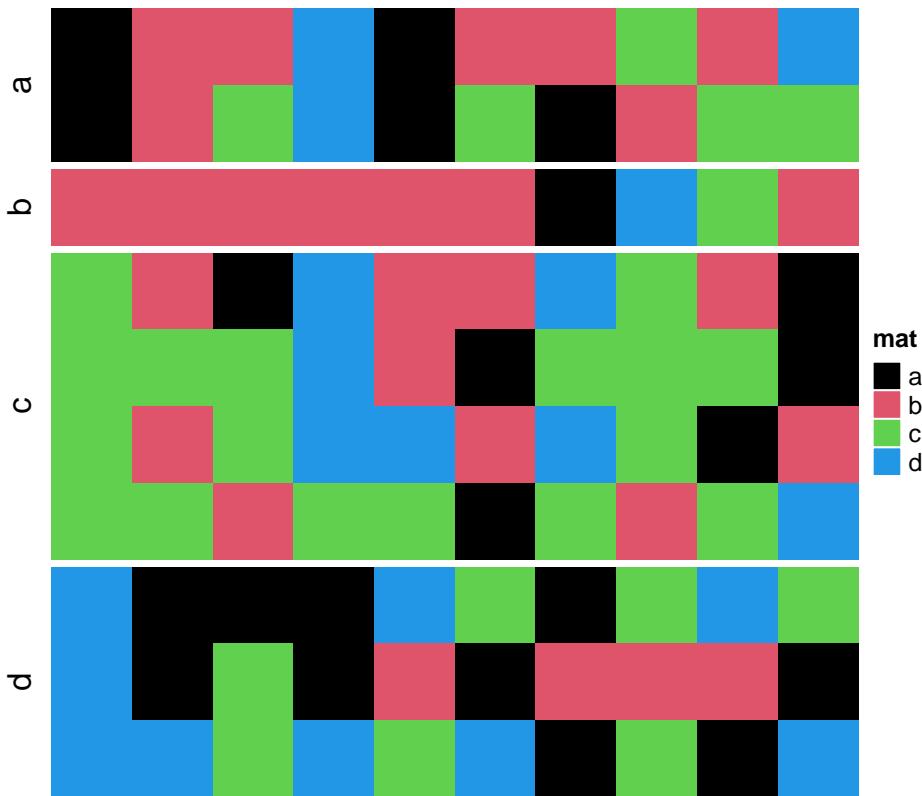
If `row_order` or `column_order` is set, in each row/column slice, it is still ordered.

```
# remember when `row_order` is set, row clustering is turned off
Heatmap(mat, name = "mat", row_order = 18:1, row_km = 2)
```



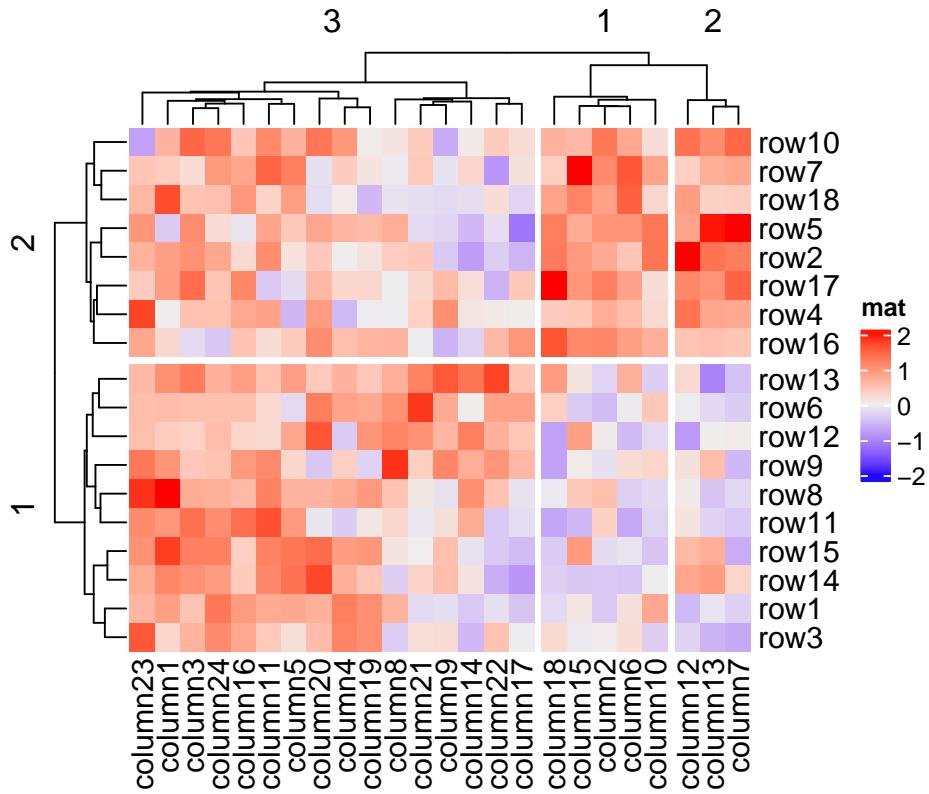
Character matrix can only be split by `row_split/column_split` argument.

```
# split by the first column in `discrete_mat`
Heatmap(discrete_mat, name = "mat", col = 1:4, row_split = discrete_mat[, 1])
```



If `row_km/column_km` is set or `row_split/column_split` is set as a vector or a data frame, hierarchical clustering is first applied to each slice which generates `k` dendrograms, then a parent dendrogram is generated based on the mean values of each slice. **The height of the parent dendrogram is adjusted by adding the maximal height of the dendrograms in all child slices and the parent dendrogram is added on top of the child dendrograms to form a single global dendrogram.** This is why you see dashed lines in the dendrograms in previous heatmaps. They are used to discriminate the parent dendrogram and the child dendrograms, and alert users they are calculated in different ways. These dashed lines can be removed by setting `show_parent_dend_line = FALSE` in `Heatmap()`, or set it as a global option: `ht_opt$show_parent_dend_line = FALSE`, but we are not suggested to do so.

```
Heatmap(mat, name = "mat", row_km = 2, column_km = 3,
        show_parent_dend_line = FALSE)
```

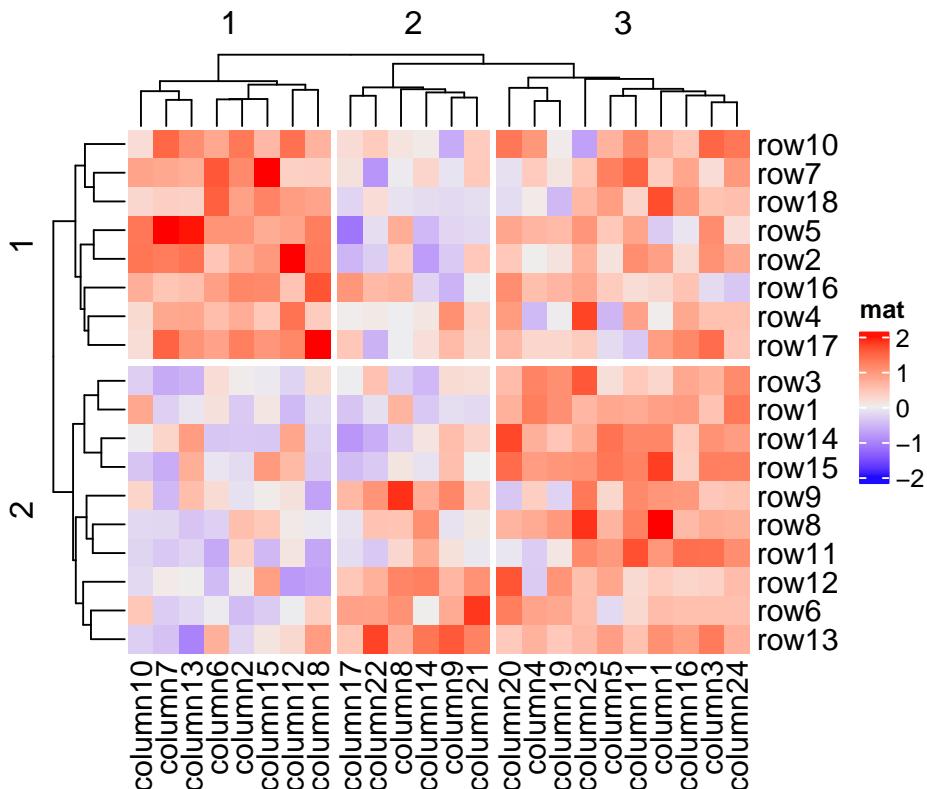


2.7.3 Split by dendrogram

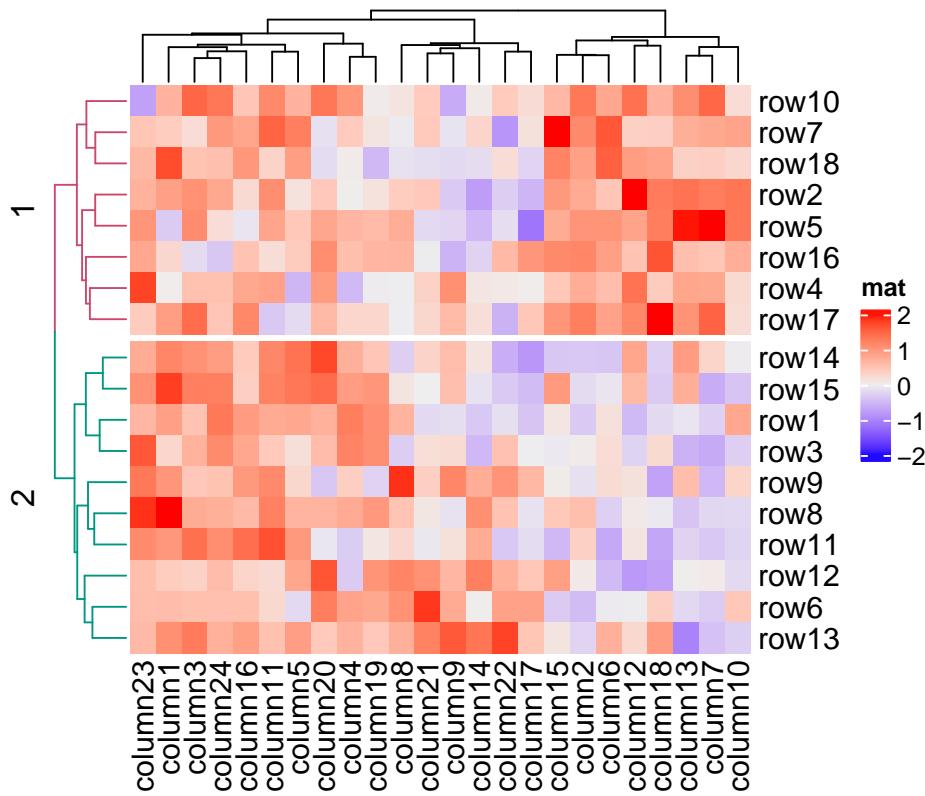
A second scenario for splitting is that users may still want to keep the global dendrogram which is generated from the complete matrix. In this case, `row_split/column_split` can be set to a single number which will apply `cutree()` on the row/column dendrogram. This works when `cluster_rows/cluster_columns` is set to TRUE or is assigned with a `hclust/dendrogram` object.

For this case, the dendrogram is still as same as the original one, except that the positions of dendrogram leaves are slightly adjusted by the gaps between slices. (There is no dashed lines, because here the dendrogram is calculated as a complete one and there is no parent dendrogram or child dendrograms.)

```
Heatmap(mat, name = "mat", row_split = 2, column_split = 3)
```



```
dend = as.dendrogram(hclust(dist(mat)))
dend = color_branches(dend, k = 2)
Heatmap(mat, name = "mat", cluster_rows = dend, row_split = 2)
```



If you want to combine splitting from `cutree()` and other categorical variables, you need to generate the partitions from `cutree()` in the first place, append to e.g. `row_split` as a data frame and then send it to `row_split` argument.

```
# code only for demonstration
split = data.frame(cutree(hclust(dist(mat))), k = 2), rep(c("A", "B"), 9))
Heatmap(mat, name = "mat", row_split = split)
```

2.7.4 Order of slices

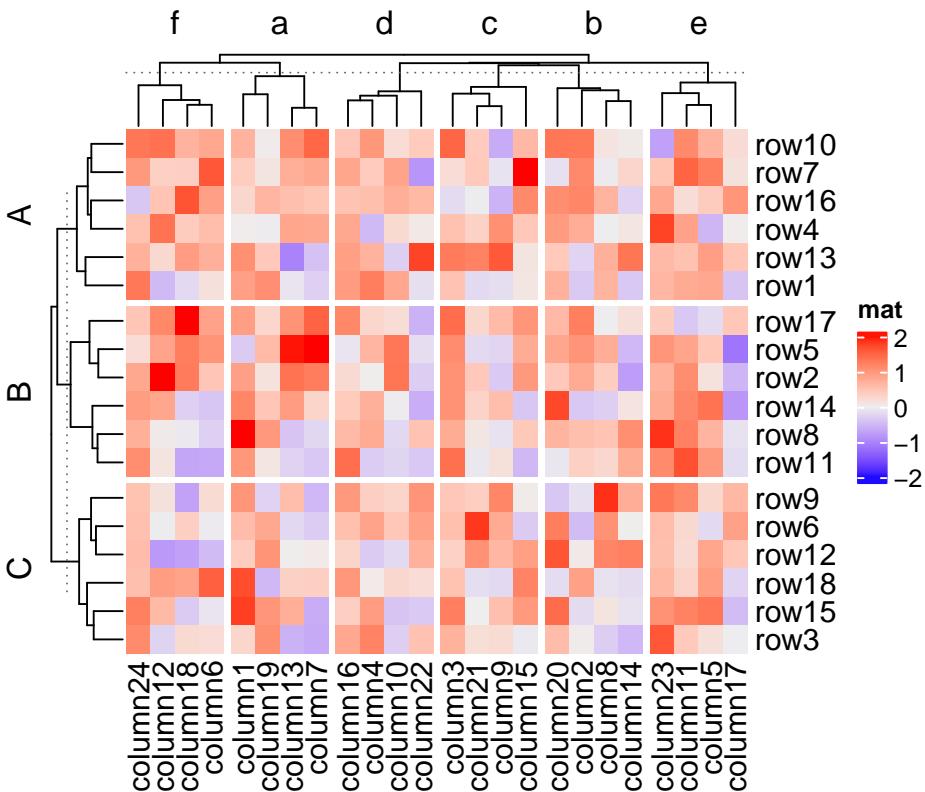
When `row_split/column_split` is set as a categorical variable (a vector or a data frame) or `row_km/column_km` is set, by default, there is an additional clustering applied to the mean of slices to show the hierarchy on the slice level. Under this scenario, you cannot precisely control the order of slices because it is controlled by the clustering of slices.

Nevertheless, you can set `cluster_row_slices` or `cluster_column_slices` to `FALSE` to turn off the clustering on slices, and now you can precisely control the order of slices.

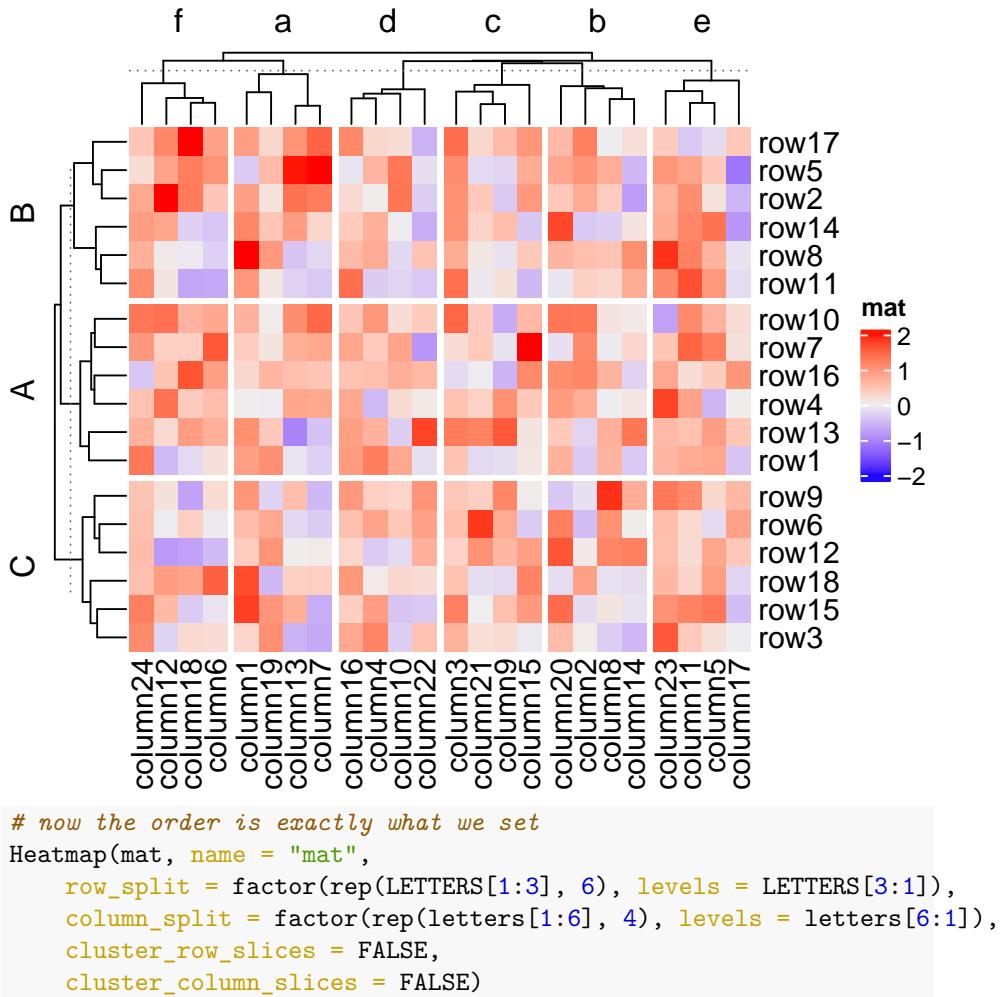
When there is no slice clustering, the order of each slice can be controlled by `levels` of each variable in `row_split/column_split` (in this case, each

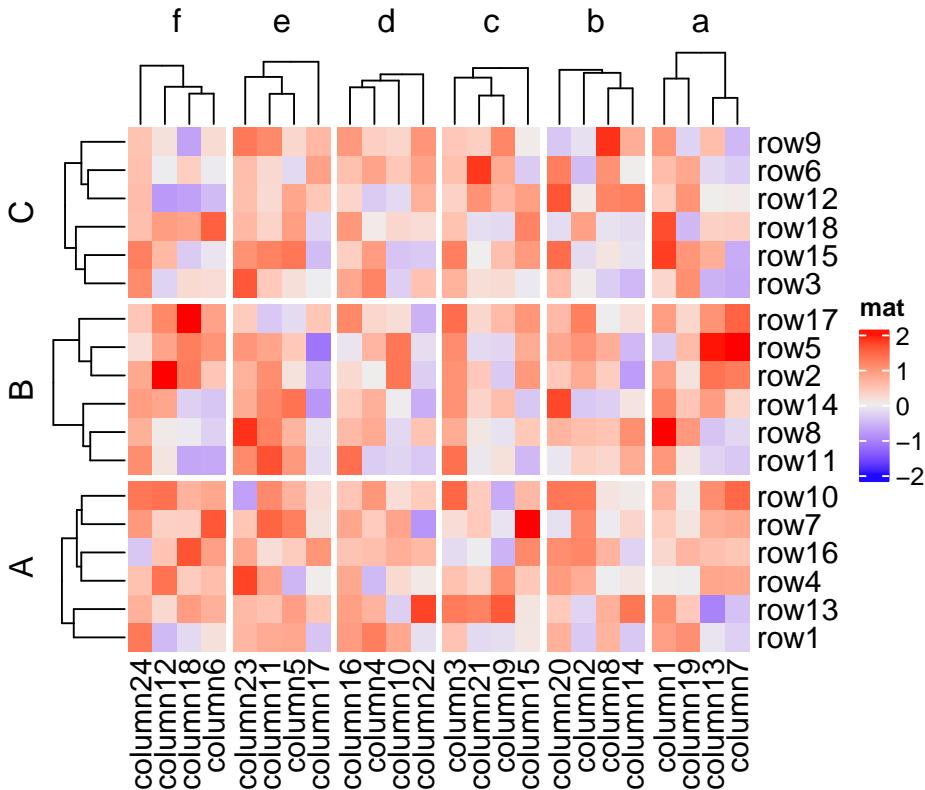
variable should be a factor). If all variables are characters, the default order is `unique(row_split)` or `unique(column_split)`. Compare following heatmaps:

```
Heatmap(mat, name = "mat",
        row_split = rep(LETTERS[1:3], 6), column_split = rep(letters[1:6], 4))
```



```
# clustering is similar as previous heatmap with branches
# in some nodes in the dendrogram flipped
Heatmap(mat, name = "mat",
        row_split = factor(rep(LETTERS[1:3], 6), levels = LETTERS[3:1]),
        column_split = factor(rep(letters[1:6], 4), levels = letters[6:1]))
```





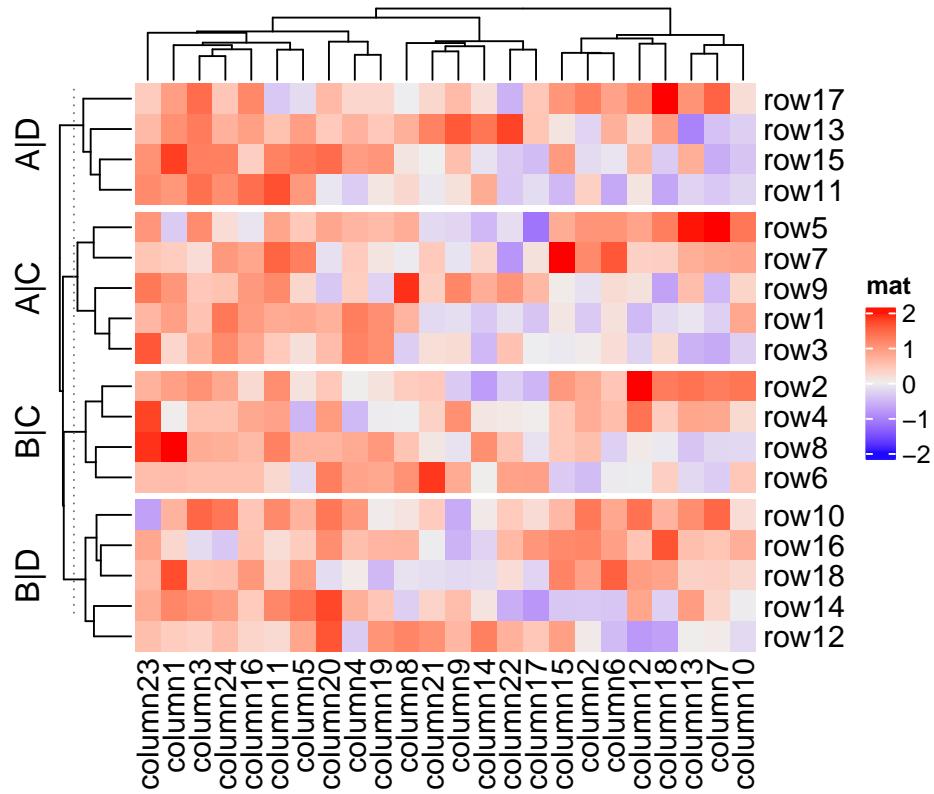
Also you can see in the third heatmap, when `cluster_row_slices` and `cluster_column_slices` are set to `FALSE`, there is no dendrogram on slice level.

2.7.5 Titles for splitting

When `row_split/column_split` is set as a single number, there is only one categorical variable, while when `row_km/column_km` is set and/or `row_split/column_split` is set as categorical variables, there will be multiple categorical variables. By default, the titles are in a form of "`level1,level2,...`" which corresponds to every combination of levels in all categorical variables. The titles for splitting can be controlled by "a template."

ComplexHeatmap supports three types of templates. The first one is by `sprintf()` where the `%s` is replaced by the corresponding level. In following example, since all combinations of `split` are `c("A", "C")`, `c("A", "D")`, `c("B", "C")` and `c("B", "D")`, if `row_title` is set to `%s|%s`, the four row titles will be `A|C`, `A|D`, `B|C`, `B|D`.

```
split = data.frame(rep(c("A", "B"), 9), rep(c("C", "D"), each = 9))
Heatmap(mat, name = "mat", row_split = split, row_title = "%s|%s")
```



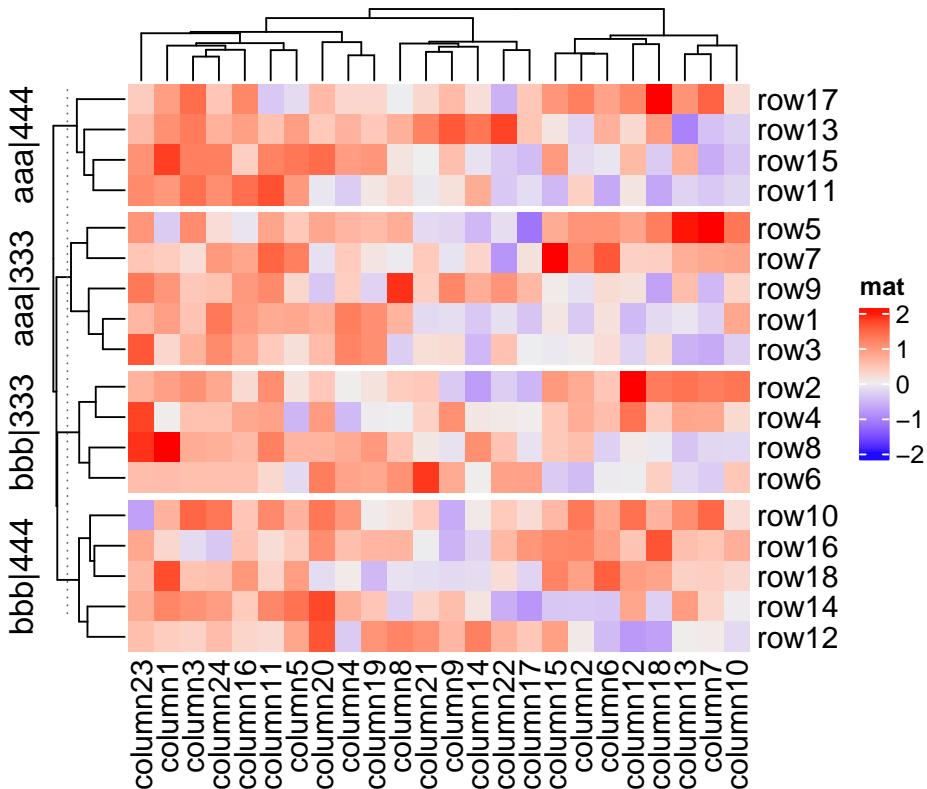
For the `sprintf()` template, you can only put the levels which are A,B,C,D in the title, and C,D is always after A,B (i.e. it always A,C and A,D). However, when making the heatmap, you might want to put more meaningful text instead of the internal levels. Once you know how to correspond the text to the level, you can add it by the following two other template methods.

In the following two template methods, special marks are used to mark the R code which is executable (it is called variable interpolation where the code is extracted and executed and the returned value is put back to the string). There are two types of template marks `@{}` and `{}`. The first one is from **GetoptLong** package which should already be installed when you install the **ComplexHeatmap** package and the second one is from **glue** package which you need to install first.

There is an internal variable `x` you should use when you use the latter two templates. `x` is just a simple vector which contains current category levels (e.g. `c("A", "C")`).

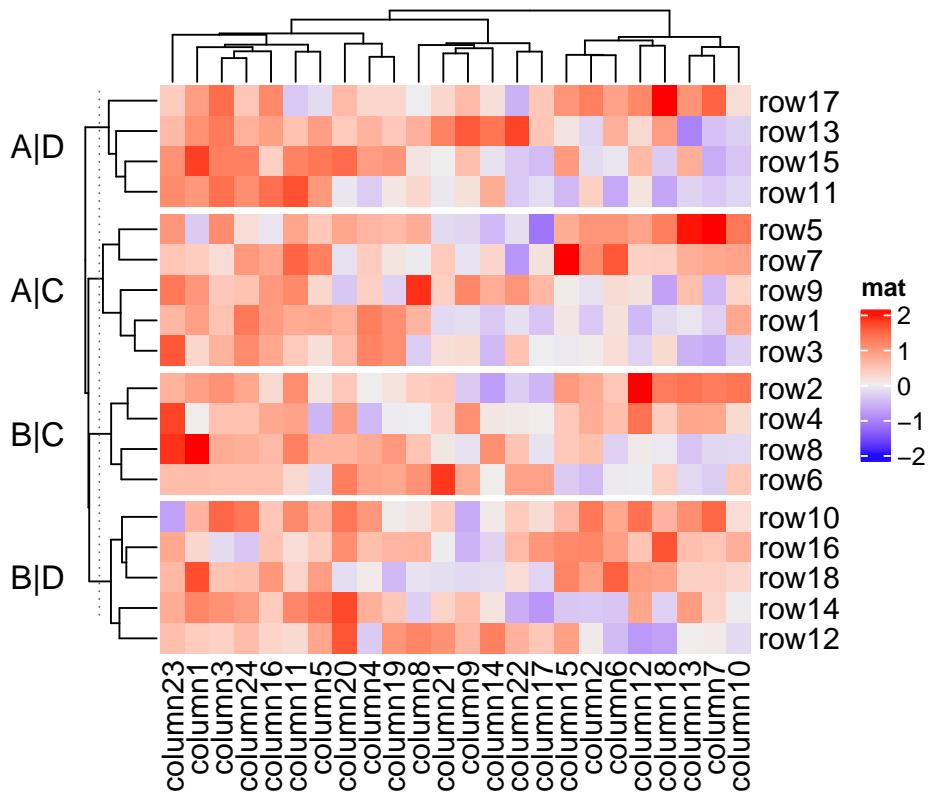
```
# We only run the code for the first heatmap
map = c("A" = "aaa", "B" = "bbb", "C" = "333", "D" = "444")
Heatmap(mat, name = "mat", row_split = split,
        row_title = "@{map[ x[1] ]}|@{map[ x[2] ]}")
```

```
Heatmap(mat, name = "mat", row_split = split,
       row_title = "{map[ x[1] ]}|{map[ x[2] ]}")
```



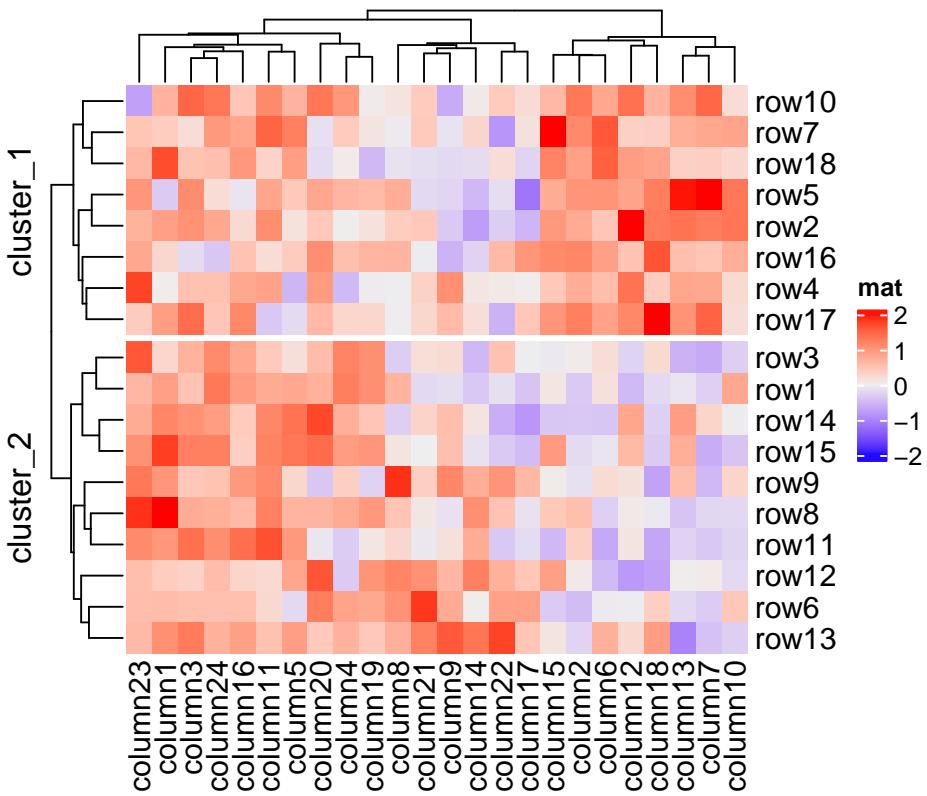
The row title is rotated by default, you can set `row_title_rot = 0` to make it horizontal:

```
Heatmap(mat, name = "mat", row_split = split, row_title = "%s|%s",
        row_title_rot = 0)
```



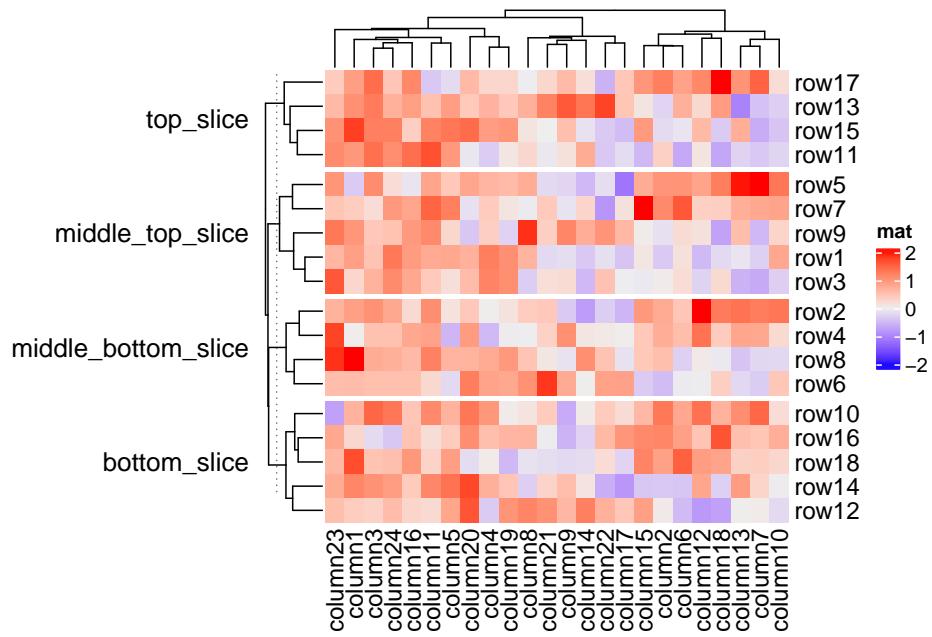
When `row_split/column_split` is set as a number, you can also use template to adjust the titles for slices.

```
Heatmap(mat, name = "mat", row_split = 2, row_title = "cluster_%s")
```



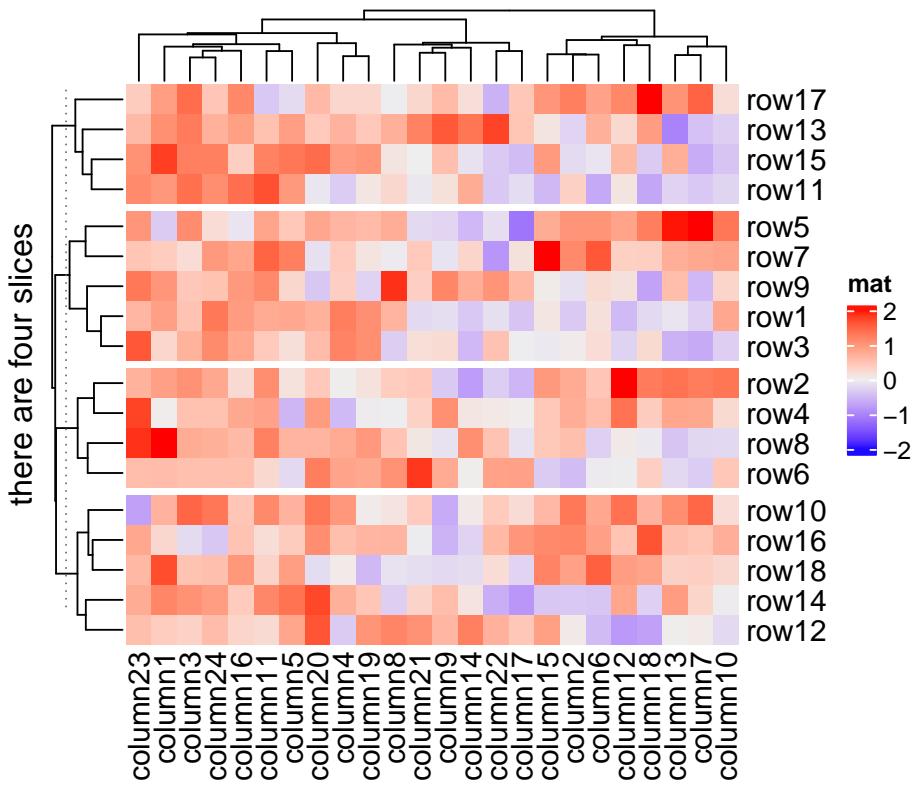
If you know the final number of row slices, you can directly set a vector of titles to `row_title`. Be careful the number of row slices is not always identical to `n_level_1 * n_level_2 * ...`

```
# we know there are four slices
Heatmap(mat, name = "mat", row_split = split,
        row_title = c("top_slice", "middle_top_slice", "middle_bottom_slice", "bottom_slice"),
        row_title_rot = 0)
```



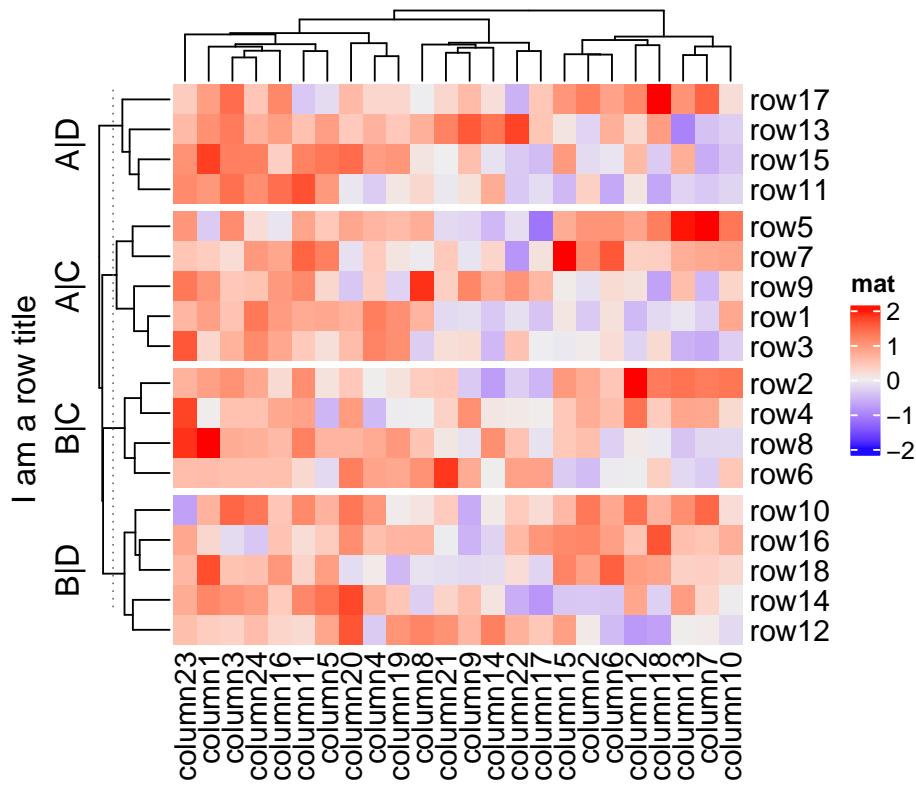
If the length of `row_title` is specified as a single string, it will be like a single title for all slices.

```
Heatmap(mat, name = "mat", row_split = split, row_title = "there are four slices")
```



If you still want titles for each slice, but also a global title, you can do as follows.

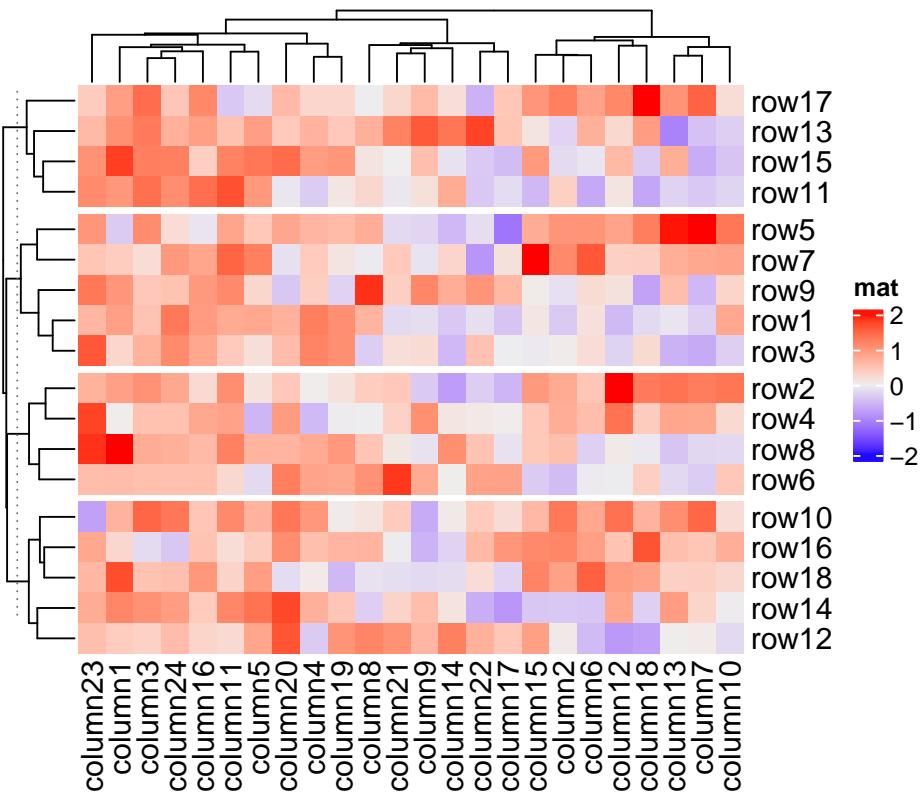
```
ht = Heatmap(mat, name = "mat", row_split = split, row_title = "%s|%s")
# This row_title is actually a heatmap-list-level row title
draw(ht, row_title = "I am a row title")
```



Actually the `row_title` used in `draw()` function is the row title of the heatmap list (although in the example there is only one heatmap). The `draw()` function and the heatmap list will be introduced in Chapter 4.

If `row_title` is set to `NULL`, no row title is drawn.

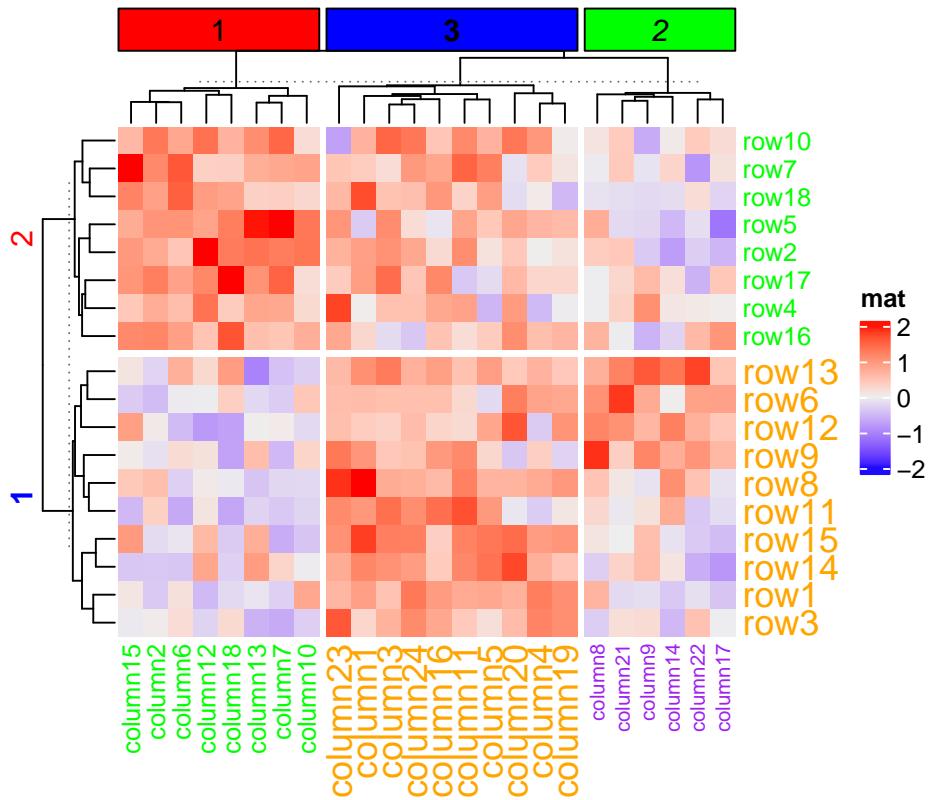
```
Heatmap(mat, name = "mat", row_split = split, row_title = NULL)
```



All these rules also work for column titles for slices.

2.7.6 Graphic parameters for splitting

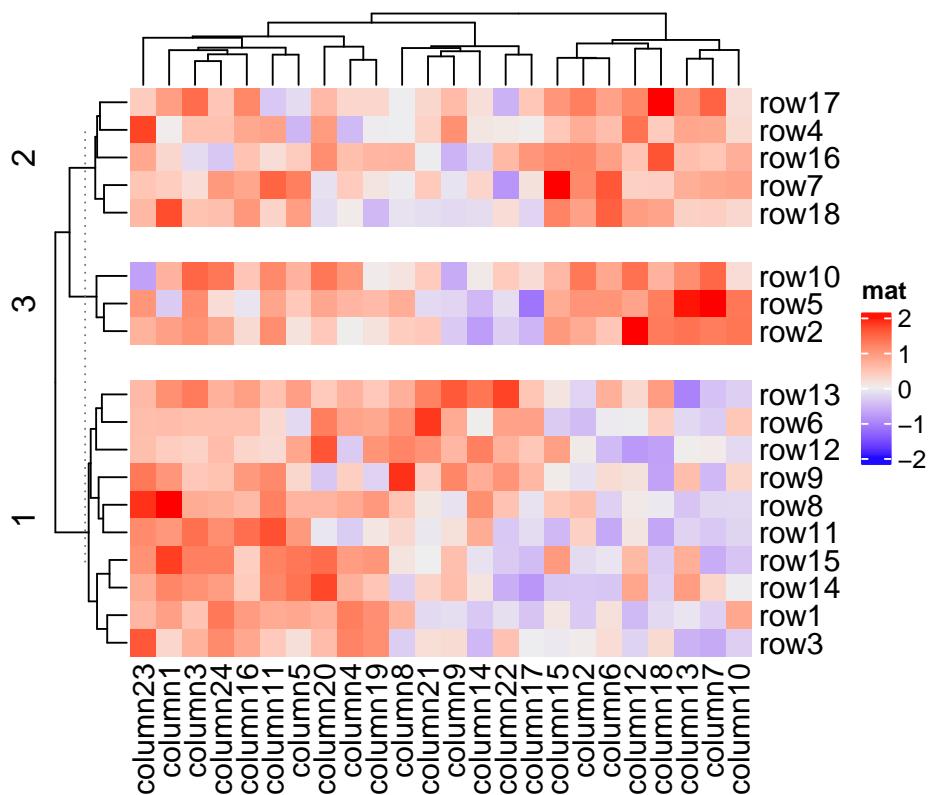
When splitting is applied on rows/columns, graphic parameters for row/column title and row/column names can be specified as same length as number of slices.



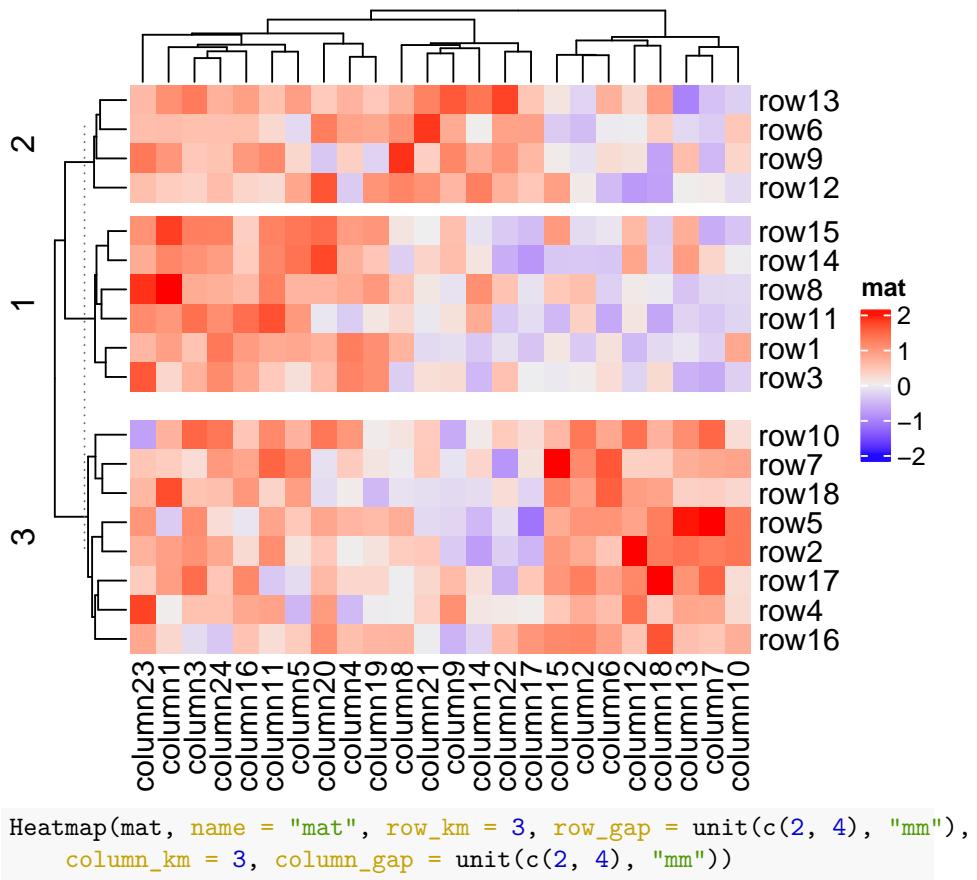
2.7.7 Gaps between slices

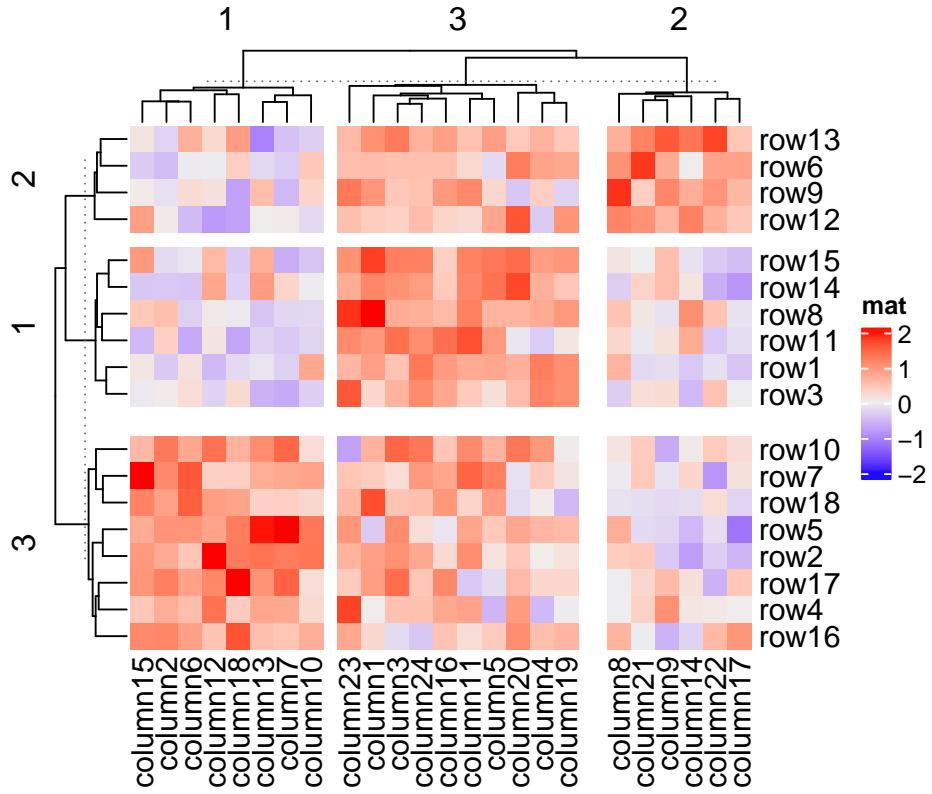
The space of gaps between row/column slices can be controlled by `row_gap/column_gap`. The value can be a single unit or a vector of units.

```
Heatmap(mat, name = "mat", row_km = 3, row_gap = unit(5, "mm"))
```



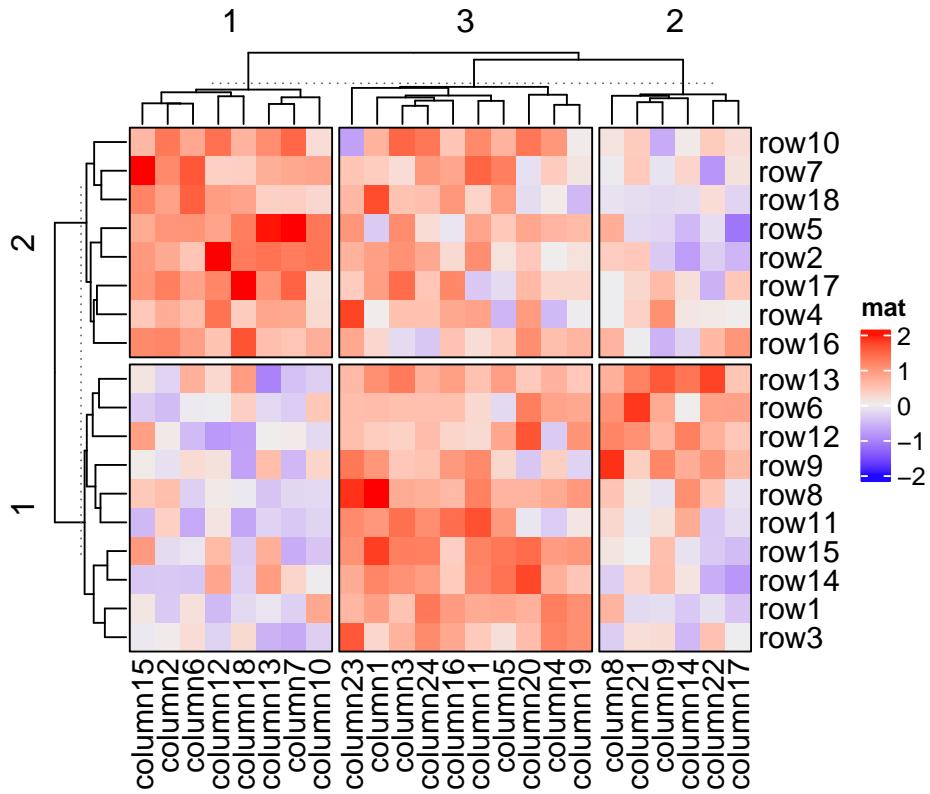
```
Heatmap(mat, name = "mat", row_km = 3, row_gap = unit(c(2, 4), "mm"))
```





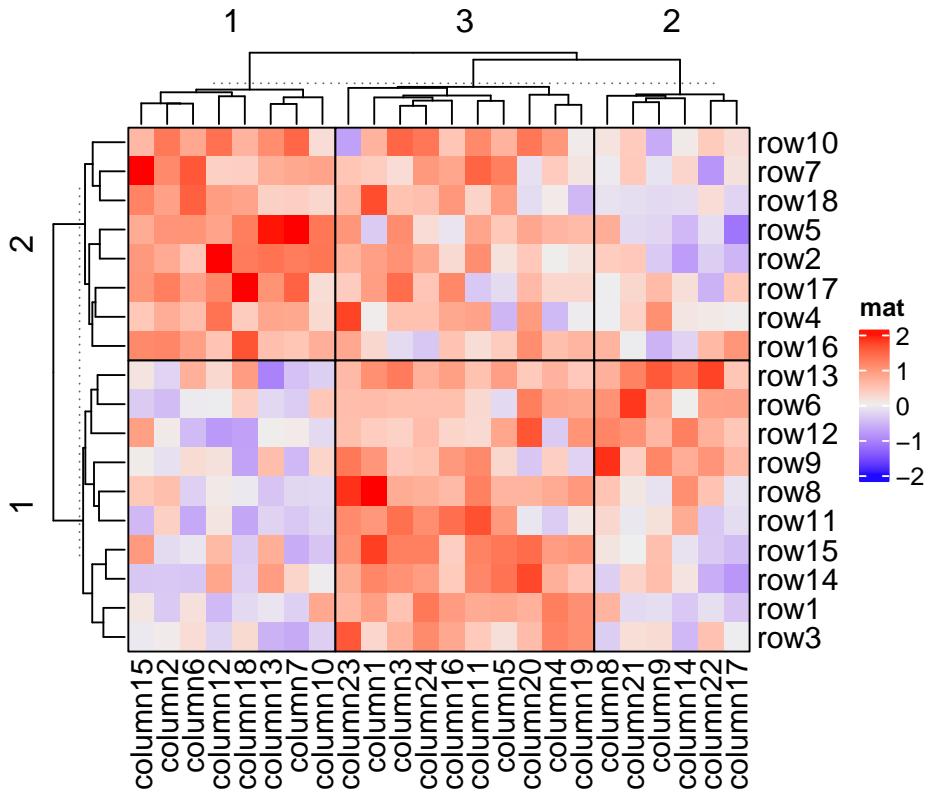
When heatmap border is added by setting `border = TRUE`, the border of every slice is added.

```
Heatmap(mat, name = "mat", row_km = 2, column_km = 3, border = TRUE)
```



If you set gap size to zero, the heatmap will look like it is partitioned by vertical and horizontal lines.

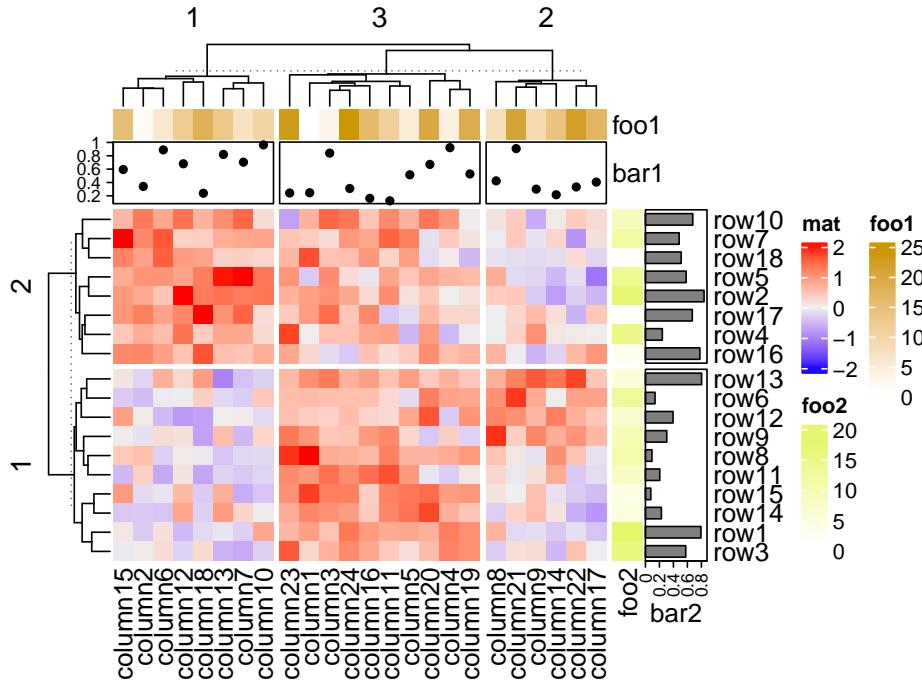
```
Heatmap(mat, name = "mat", row_km = 2, column_km = 3,
       row_gap = unit(0, "mm"), column_gap = unit(0, "mm"), border = TRUE)
```



2.7.8 Split heatmap annotations

When the heatmap is split, all the heatmap components are split accordingly. Following gives you a simple example and the heatmap annotation will be introduced in Chapter 3.

```
Heatmap(mat, name = "mat", row_km = 2, column_km = 3,
       top_annotation = HeatmapAnnotation(foo1 = 1:24, bar1 = anno_points(runif(24))),
       right_annotation = rowAnnotation(foo2 = 18:1, bar2 = anno_barplot(runif(18)))
     )
```



2.8 Heatmap as raster image

When we produce so-called “high quality figures,” normally we save the figures as vector graphics in the format of *e.g.* pdf or svg. The vector graphics basically store details of every single graphic elements, thus, if a heatmap made from a very huge matrix is saved as vector graphics, the final file size would be very big. On the other hand, when visualizing *e.g.* the pdf file on the screen, multiple grids from the heatmap actually only map to single pixels, due to the limited size of the screen. Thus, there need some ways to effectively reduce the original image and it is not necessary to store the complete matrix for the heatmap.

Rasterization is a way to convert the vector graphics into a matrix of colors. In this case, an image is represented as a matrix of RGB values, which is called a raster image. If the heatmap is larger than the size of the screen or the pixels that current graphics devices can support, we can convert the heatmap and reduce it, by saving it in a form of a color matrix with the same dimension as on the device.

Let’s assume a matrix has n_r rows and n_c columns. When it is drawn on a certain graphics device, *e.g.* an on-screen device, the corresponding heatmap body has p_r and p_c pixels (or points) for the rows and columns, respectively. When $n_r > p_r$ and/or $n_c > p_c$, multiple values in the matrix are mapped to single pixels. Here we need to reduce n_r and/or n_c if they are larger than p_r and/or p_c .

To make it simple, I assume both $n_r > p_r$ and $n_c > p_c$. The principle is basically the same for the scenarios where only one dimension of the matrix is larger than the device.

From **ComplexHeatmap** version 2.5.4, there are following three implementations for image rasterization. Note the implementation is a little bit different from the earlier versions (of course, better than the earlier versions).

1. First an image (in a specific format, *e.g.* png or jpeg) with $(p_r \cdot a) \times (p_c \cdot a)$ resolution is saved into a temporary file (*e.g.*, by `png()` or `jpeg()`) where a is a zooming factor, next it is read back as a `raster` object by *e.g.* `png::readPNG()` or `jpeg::readJPEG()`, and later the raster object is filled into the heatmap body by `grid::grid.raster()`. So we can say, the rasterization is done by the raster image devices (`png()` or `jpeg()`).

This type of rasterization is automatically turned on (**if magick package is not installed**) when the number of rows or columns exceeds 2000 (You will see a message. It won't happen silently). It can also be manually controlled by setting the `use_raster` argument:

```
Heatmap(..., use_raster = TRUE)
```

The zooming factor is controlled by `raster_quality` argument. A value larger than 1 generates files with larger size.

```
Heatmap(..., use_raster = TRUE, raster_quality = 5)
```

2. Simply reduce the original matrix to $p_r \times p_c$ where now each single values can correspond to single pixels. In the reduction, a user-defined function is applied to summarize the sub-matrices.

This can be set by `raster_resize_mat` argument:

```
# the default summary function is mean()
Heatmap(..., use_raster = TRUE, raster_resize_mat = TRUE)
# use max() as the summary function
Heatmap(..., use_raster = TRUE, raster_resize_mat = max)
# randomly pick one
Heatmap(..., use_raster = TRUE, raster_resize_mat = function(x) sample(x, 1))
```

3. A temporary image with complete resolution $n_r \times n_c$ is first generated, here `magick::image_resize()` is used to reduce the image to size $p_r \times p_c$. Finally the reduced image is read as a `raster` object and filled into the heatmap body. **magick** provides a lot of methods for “resizing”/“scaling” the image, which is called the “filtering methods” under the term of **magick**. All filtering methods can be obtained by `magick::filter_types()`.

This type of rasterization can be turned on by setting `raster_by_magick = TRUE` and choosing a proper `raster_magick_filter`.

```
Heatmap(..., use_raster = TRUE, raster_by_magick = TRUE)
Heatmap(..., use_raster = TRUE, raster_by_magick = TRUE, raster_magick_filter = ...)
```

The type of temporary image is controlled by `raster_device` argument. All supported “raster devices” are: `png`, `CairoPNG`, `agg_png`, `jpeg`, `tiff`, `CairoJPEG` and `CairoTIFF`. `CairoPNG` is taken as the default raster device (If `Cairo` package is installed) because the previously default device `png` occasionally produces white vertical and horizontal lines in the raster image.

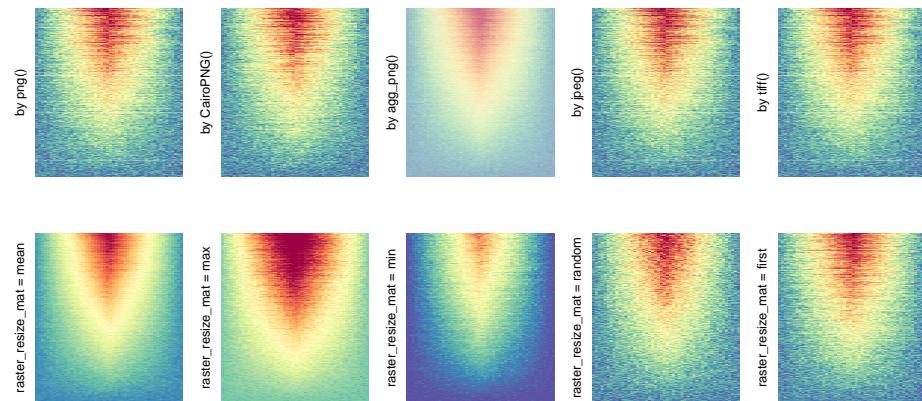
In `ComplexHeatmap`, `use_raster` is by default turned on if the number of rows or columns is more than 2000 in the matrix.

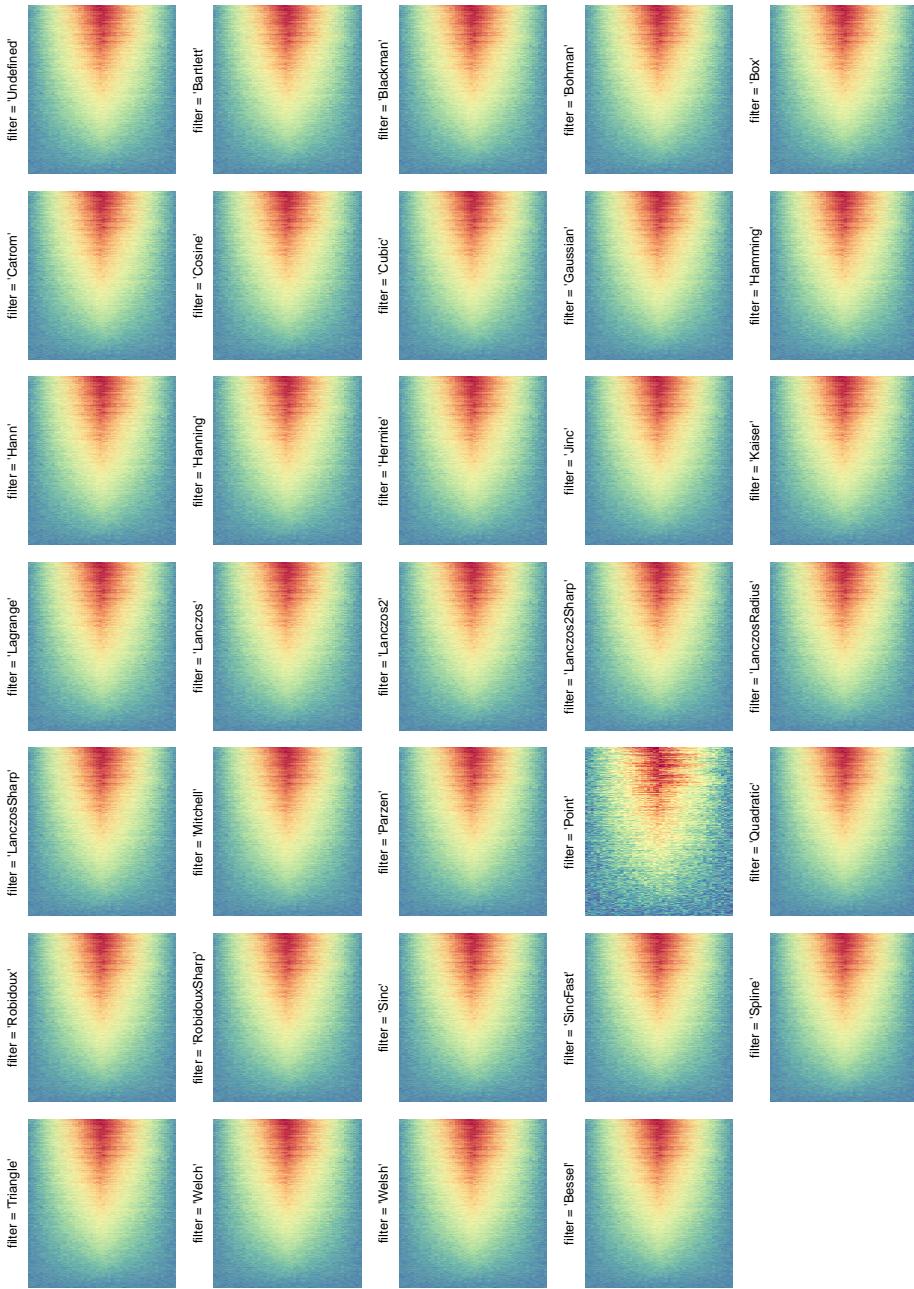
In the following parts of this section, we compare the visual difference between different image rasterization methods. The following example is from Guillaume Devailly’s simulated data but with small adaptations. This example shows an enrichment pattern to the top center of the plot.

In the following examples, We won’t show the code for making heatmaps because there are too many heatmaps and the specific settings is already written as the row title of each heatmap. We set the same color mapping for all heatmaps, so that you can see how different rasterizations change the original patterns.

For the comparison, we generated many heatmaps. They can be categorized into three groups, as corresponded to the three rasterization methods mentioned previously.

- by `png()`/`CairoPNG()`/`agg_png()`/`jpeg()`/`tiff()`: Rasterization method 1. First row in the following heatmaps.
- `raster_resize_mat = *:` Rasterization method 2, with different summary methods. Second row in the following heatmaps.
- `filter = *:` Rasterization method 3, with different filtering method. The string `filter` should be `raster_magick_filter`. It is truncated so that the row title won’t be cut by the plot regions.





More examples on image rasteration can be found in this blog post: [Rasterization in ComplexHeatmap](#).

According to the examples that have been shown, we would say rasterization by

magick package performs better, thus, by default, in **ComplexHeatmap**, the rasterization is done by **magick** (with "Lanczos" as the default filter method) and if **magick** is not installed, it uses CairoPNG() and a friendly message is printed to suggest users to install **magick**.

Following example compares the PDF file size with raster images by different raster devices.

```
library(GetoptLong)
set.seed(123)
mat2 = matrix(rnorm(10000*100), ncol = 100)
pdf(qq("heatmap_no_rasteration.pdf"), width = 8, height = 8)
ht = Heatmap(mat2, cluster_rows = FALSE, cluster_columns = FALSE, use_raster = FALSE)
draw(ht)
dev.off()

# Here I removed CairoTIFF because it is not working on my laptop
all_devices = c("png", "CairoPNG", "agg_png", "jpeg", "CairoJPEG", "tiff")
for(device in all_devices) {
  pdf(qq("heatmap_@{device}.pdf"), width = 8, height = 8)
  ht = Heatmap(mat2, cluster_rows = FALSE, cluster_columns = FALSE, use_raster = TRUE
               raster_device = device)
  draw(ht)
  dev.off()
}

all_files = qq("heatmap_@{all_devices}.pdf", collapse = FALSE)
all_files = c("heatmap_no_rasteration.pdf", all_files)
fs = file.size(all_files)
names(fs) = all_files
sapply(fs, function(x) paste(round(x/1024), "KB"))

## heatmap_no_rasteration.pdf          heatmap_png.pdf
##                  "6356 KB"           "175 KB"
##      heatmap_CairoPNG.pdf          heatmap_agg_png.pdf
##                  "177 KB"           "308 KB"
##      heatmap_jpeg.pdf            heatmap_CairoJPEG.pdf
##                  "667 KB"           "677 KB"
##      heatmap_tiff.pdf
##                  "175 KB"
```

We can see png-family methods and tiff generate smaller file size.

2.9 Customize the heatmap body

The heatmap body can be self-defined to add more types of graphics. By default the heatmap body is composed by a matrix of small rectangles (it might be

called grids in other parts of this documentation, but let's call it “*cells*” here) with different filled colors. However, it is also possible to add more graphics or symbols as additional layers on the heatmap. There are two arguments `cell_fun` and `layer_fun` which both should be user-defined functions.

In later chapter, we will introduce OncoPrint (Chapter 7), UpSet plot (Chapter 8) and 3D heatmap (Chapter 12). They are all implemented with `cell_fun`/`layer_fun`.

2.9.1 `cell_fun`

`cell_fun` draws in each cell repeatedly, which is internally executed in two nested `for` loops, while `layer_fun` is the vectorized version of `cell_fun`. `cell_fun` is easier to understand but `layer_fun` is much faster to execute and more customizable.

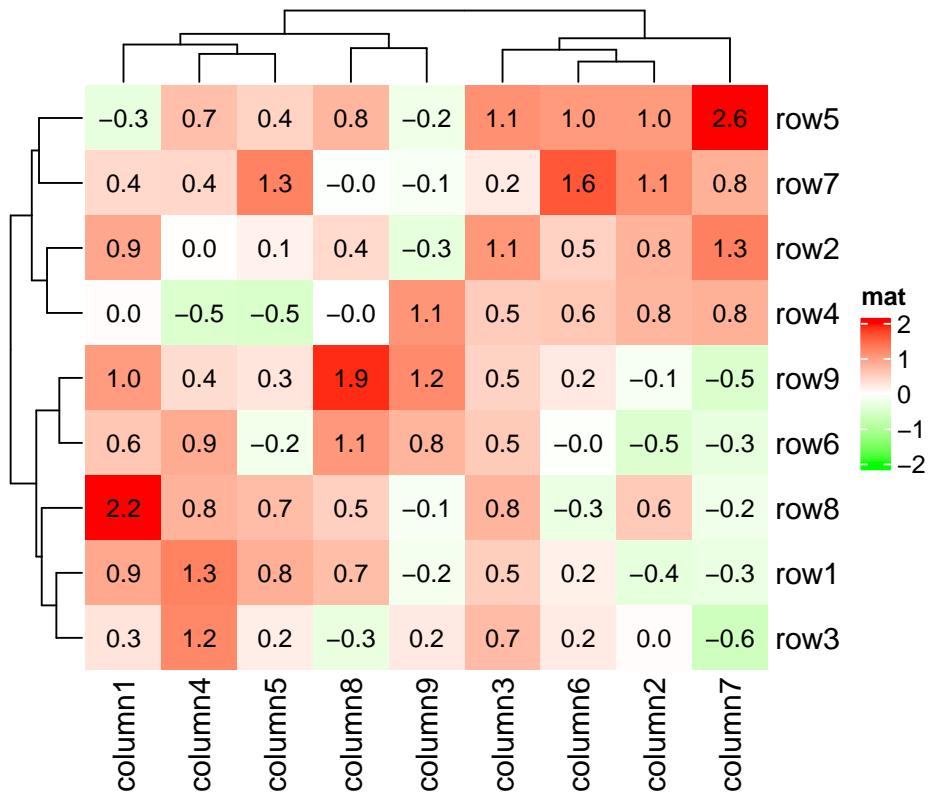
`cell_fun` expects a function with 7 arguments (the argument names can be different from following, but the order must be the same), which are:

- `j`: column index in the matrix. Column index corresponds to the x-direction in the viewport, that's why `j` is put as the first argument.
- `i`: row index in the matrix.
- `x`: x coordinate of middle point of the cell which is measured in the viewport of the heatmap body.
- `y`: y coordinate of middle point of the cell which is measured in the viewport of the heatmap body.
- `width`: width of the cell. The value is `unit(1/ncol(sub_mat), "npc")` where `sub_mat` correspond to the sub-matrix by row splitting and column splitting.
- `height`: height of the cell. The value is `unit(1/nrow(sub_mat), "npc")`.
- `fill`: color of the cell.

The values for the seven arguments are automatically sent to the function when executed in each cell.

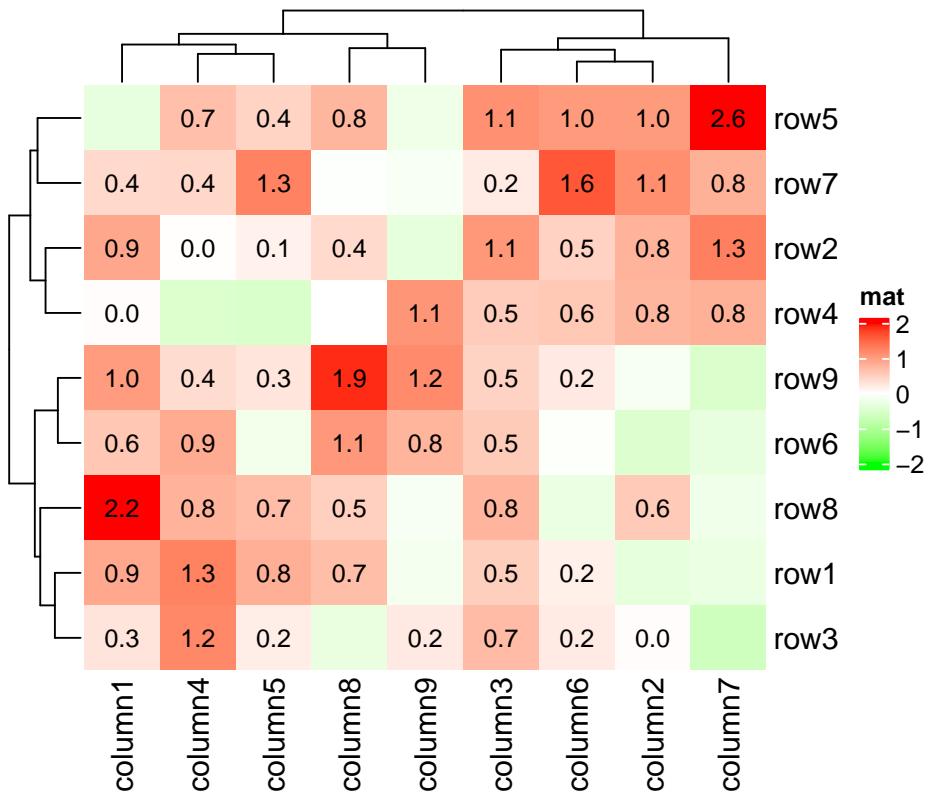
The most common use is to add values in the matrix onto the heatmap:

```
small_mat = mat[1:9, 1:9]
col_fun = colorRamp2(c(-2, 0, 2), c("green", "white", "red"))
Heatmap(small_mat, name = "mat", col = col_fun,
        cell_fun = function(j, i, x, y, width, height, fill) {
            grid.text(sprintf("%.1f", small_mat[i, j]), x, y, gp = gpar(fontsize = 10))
        })
```



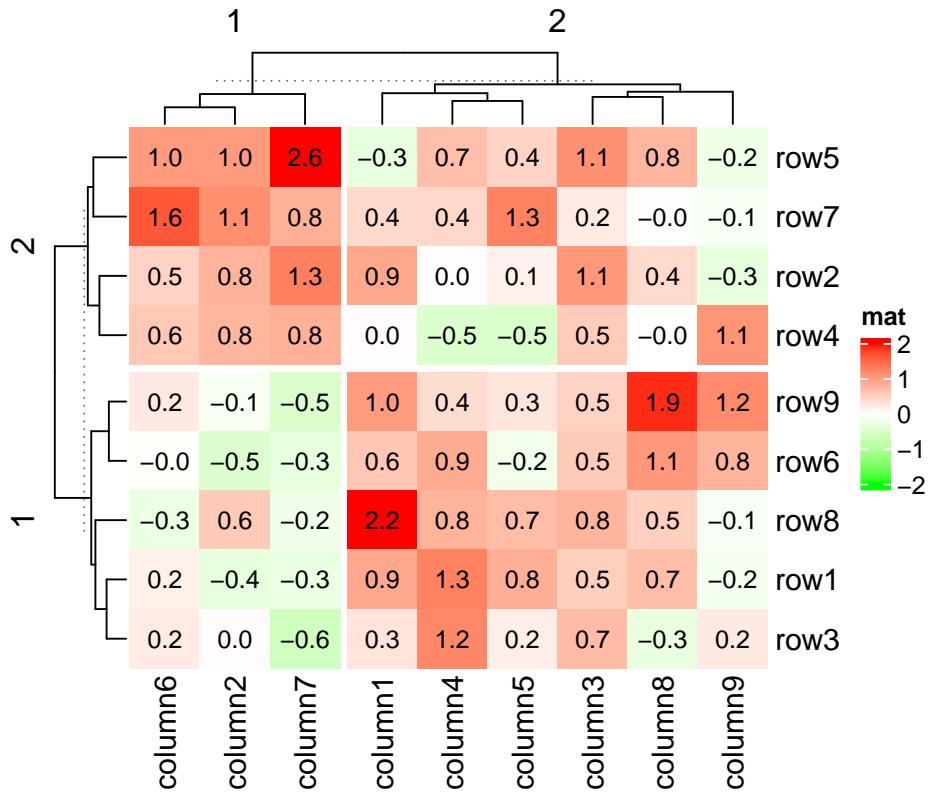
and we can also choose only to add text for the cells with positive values:

```
Heatmap(small_mat, name = "mat", col = col_fun,
        cell_fun = function(j, i, x, y, width, height, fill) {
          if(small_mat[i, j] > 0)
            grid.text(sprintf("%.1f", small_mat[i, j]), x, y, gp = gpar(fontsize = 10))
        })
```



You can split the heatmap without doing anything extra to `cell_fun`:

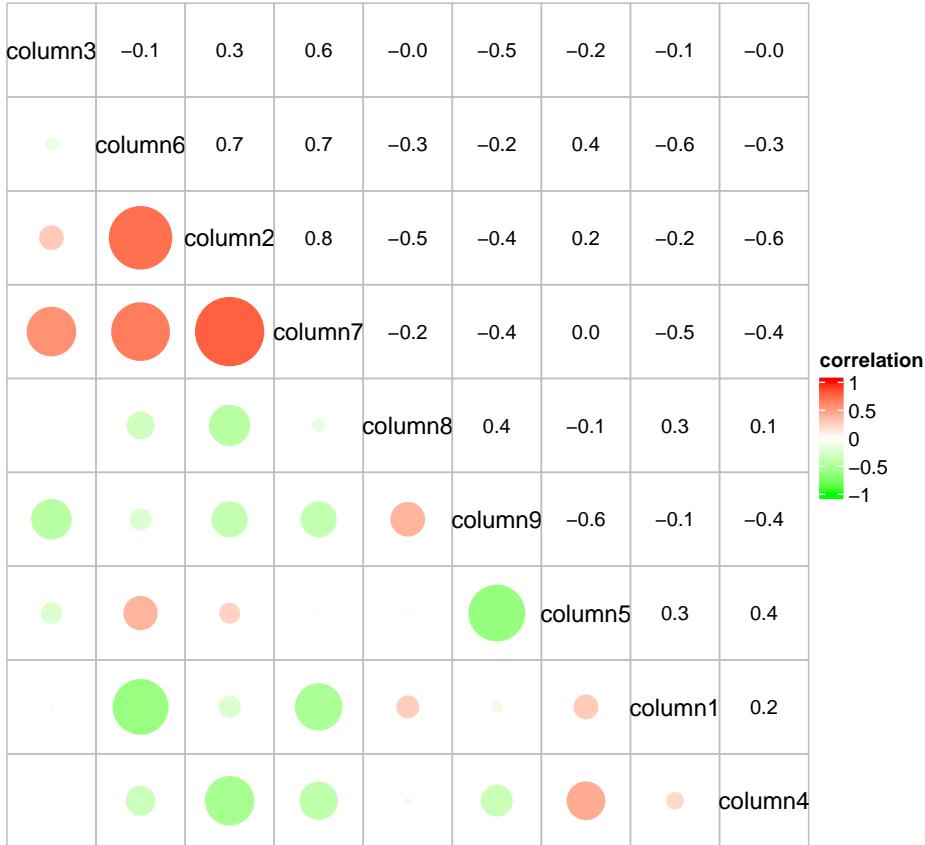
```
Heatmap(small_mat, name = "mat", col = col_fun,
        row_km = 2, column_km = 2,
        cell_fun = function(j, i, x, y, width, height, fill) {
          grid.text(sprintf("%.1f", small_mat[i, j]), x, y, gp = gpar(fontsize = 10))
})
```



In following example, we make a heatmap which shows correlation matrix similar as the `corrplot` package:

```
cor_mat = cor(small_mat)
od = hclust(dist(cor_mat))$order
cor_mat = cor_mat[od, od]
nm = rownames(cor_mat)
col_fun = circlize::colorRamp2(c(-1, 0, 1), c("green", "white", "red"))
# `col = col_fun` here is used to generate the legend
Heatmap(cor_mat, name = "correlation", col = col_fun, rect_gp = gpar(type = "none"),
       cell_fun = function(j, i, x, y, width, height, fill) {
         grid.rect(x = x, y = y, width = width, height = height,
                   gp = gpar(col = "grey", fill = NA))
         if(i == j) {
           grid.text(nm[i], x = x, y = y)
         } else if(i > j) {
           grid.circle(x = x, y = y, r = abs(cor_mat[i, j])/2 * min(unit.c(width, height)),
                       gp = gpar(fill = col_fun(cor_mat[i, j]), col = NA))
         } else {
           grid.text(sprintf("%.1f", cor_mat[i, j]), x, y, gp = gpar(fontsize = 10))
         }
       })
```

```
}, cluster_rows = FALSE, cluster_columns = FALSE,
show_row_names = FALSE, show_column_names = FALSE)
```



As you may see in previous plot, when setting the non-standard parameter `rect_gp = gpar(type = "none")`, the clustering is performed but nothing is drawn on the heatmap body.

One last example is to visualize a GO game. The input data takes records of moves in the game.

```
str = "B[cp];W[pq];B[dc];W[qd];B[eq];W[od];B[de];W[jc];B[qk];W[qn]
;B[qh];W[ck];B[ci];W[cn];B[hc];W[je];B[jq];W[df];B[ee];W[cf]
;B[ei];W[bc];B[ce];W[be];B[bd];W[cd];B[bf];W[ad];B[bg];W[cc]
;B[eb];W[db];B[ec];W[lq];B[nq];W[jp];B[iq];W[kq];B[pp];W[op]
;B[po];W[oq];B[rp];W[q1];B[oo];W[no];B[pl];W[pm];B[np];W[qq]
;B[om];W[ol];B[pk];W[qp];B[on];W[rm];B[mo];W[nr];B[r1];W[rk]
;B[qm];W[dp];B[dq];W[q1];B[or];W[mp];B[nn];W[mq];B[qm];W[bp]
;B[co];W[q1];B[no];W[pr];B[qm];W[dd];B[pn];W[ed];B[bo];W[eg]
```

```
;B[ef] ;W[dg] ;B[ge] ;W[gh] ;B[gf] ;W[gg] ;B[ek] ;W[ig] ;B[fd] ;W[en]
;B[bn] ;W[ip] ;B[dm] ;W[ff] ;B[cb] ;W[fe] ;B[hp] ;W[ho] ;B[hq] ;W[e1]
;B[d1] ;W[fk] ;B[ej] ;W[fp] ;B[go] ;W[hn] ;B[fo] ;W[em] ;B[dn] ;W[eo]
;B[gp] ;W[ib] ;B[gc] ;W[pg] ;B[qg] ;W[ng] ;B[qc] ;W[re] ;B[pf] ;W[of]
;B[rc] ;W[ob] ;B[ph] ;W[qa] ;B[rn] ;W[mi] ;B[og] ;W[oe] ;B[qe] ;W[rd]
;B[rf] ;W[pd] ;B[gm] ;W[gl] ;B[fm] ;W[fl] ;B[lj] ;W[mj] ;B[lk] ;W[ro]
;B[h1] ;W[hk] ;B[ik] ;W[dk] ;B[bi] ;W[di] ;B[dj] ;W[dh] ;B[hj] ;W[gj]
;B[li] ;W[lh] ;B[kh] ;W[lg] ;B[jn] ;W[do] ;B[c1] ;W[ij] ;B[gk] ;W[b1]
;B[cm] ;W[hk] ;B[jk] ;W[lo] ;B[hi] ;W[hm] ;B[gk] ;W[bm] ;B[cn] ;W[hk]
;B[i1] ;W[cq] ;B[bq] ;W[ii] ;B[sm] ;W[jo] ;B[kn] ;W[fq] ;B[ep] ;W[cj]
;B[bk] ;W[er] ;B[cr] ;W[gr] ;B[gk] ;W[fj] ;B[ko] ;W[kp] ;B[hr] ;W[jr]
;B[nh] ;W[mh] ;B[mk] ;W[bb] ;B[da] ;W[jh] ;B[ic] ;W[id] ;B[hb] ;W[jb]
;B[oj] ;W[fn] ;B[fs] ;W[fr] ;B[gs] ;W[es] ;B[hs] ;W[gn] ;B[kr] ;W[is]
;B[dr] ;W[fi] ;B[bj] ;W[hd] ;B[gd] ;W[ln] ;B[lm] ;W[oi] ;B[oh] ;W[ni]
;B[pi] ;W[ki] ;B[kj] ;W[ji] ;B[so] ;W[rq] ;B[if] ;W[jf] ;B[hh] ;W[hf]
;B[he] ;W[ie] ;B[hg] ;W[ba] ;B[ca] ;W[sp] ;B[im] ;W[sn] ;B[rm] ;W[pe]
;B[qf] ;W[if] ;B[hk] ;W[nj] ;B[nk] ;W[lr] ;B[mn] ;W[aaf] ;B[ag] ;W[ch]
;B[bh] ;W[lp] ;B[ia] ;W[ja] ;B[ha] ;W[sf] ;B[sg] ;W[se] ;B[eh] ;W[fh]
;B[in] ;W[ih] ;B[ae] ;W[so] ;B[af]"
```

We convert it into a matrix:

```
str = gsub("\\\\n", "", str)
step = strsplit(str, ";")[[1]]
type = gsub("(B|W).*", "\\\\1", step)
row = gsub("(B|W)\\\\[(.)\\\\]", "\\\\2", step)
column = gsub("(B|W)\\\\[(.)\\\\]", "\\\\2", step)

go_mat = matrix(nrow = 19, ncol = 19)
rownames(go_mat) = letters[1:19]
colnames(go_mat) = letters[1:19]
for(i in seq_along(row)) {
  go_mat[row[i], column[i]] = type[i]
}
go_mat[1:4, 1:4]

##  a   b   c   d
## a NA  NA  NA  "W"
## b "W" "W" "W" "B"
## c "B" "B" "W" "W"
## d "B" "W" "B" "W"
```

Black and white stones are put based on the values in the matrix:

```
ht = Heatmap(go_mat, name = "go", rect_gp = gpar(type = "none"),
  cell_fun = function(j, i, x, y, w, h, col) {
    grid.rect(x, y, w, h, gp = gpar(fill = "#dcb35c", col = NA))
```

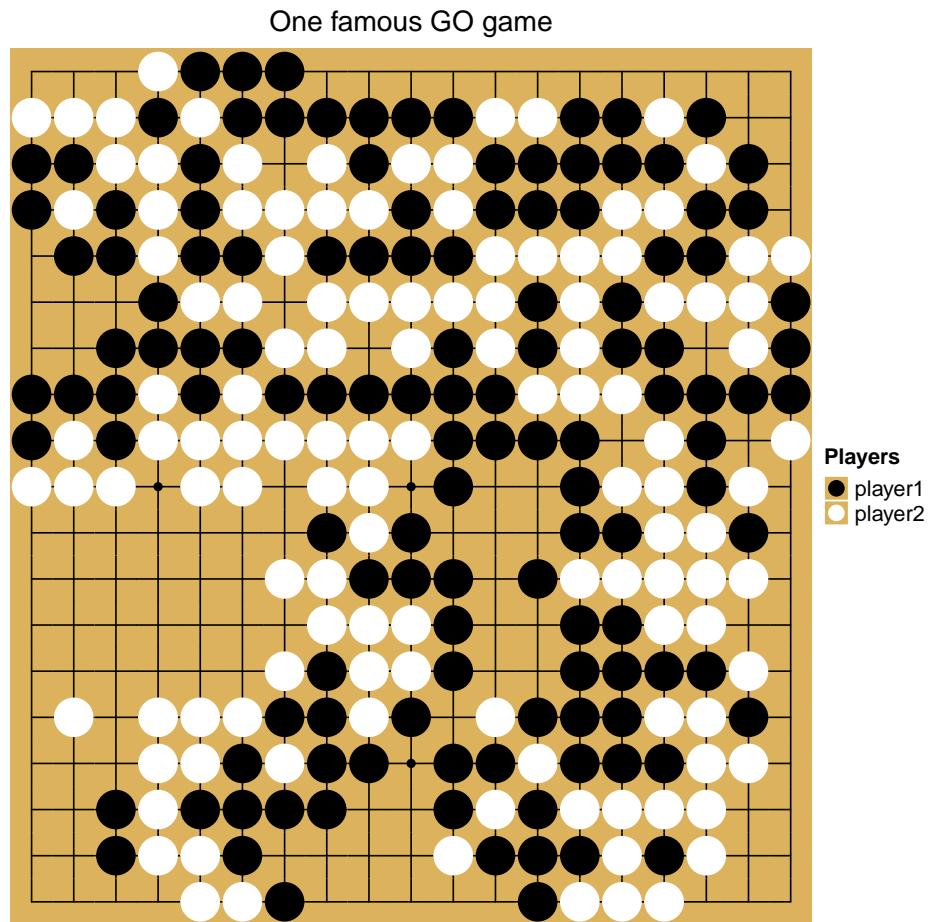
```

if(i == 1) {
    grid.segments(x, y-h*0.5, x, y)
} else if(i == nrow(go_mat)) {
    grid.segments(x, y, x, y+h*0.5)
} else {
    grid.segments(x, y-h*0.5, x, y+h*0.5)
}
if(j == 1) {
    grid.segments(x, y, x+w*0.5, y)
} else if(j == ncol(go_mat)) {
    grid.segments(x-w*0.5, y, x, y)
} else {
    grid.segments(x-w*0.5, y, x+w*0.5, y)
}

if(i %in% c(4, 10, 16) & j %in% c(4, 10, 16)) {
    grid.points(x, y, pch = 16, size = unit(2, "mm"))
}

r = min(unit.c(w, h))*0.45
if(is.na(go_mat[i, j])) {
} else if(go_mat[i, j] == "W") {
    grid.circle(x, y, r, gp = gpar(fill = "white", col = "white"))
} else if(go_mat[i, j] == "B") {
    grid.circle(x, y, r, gp = gpar(fill = "black", col = "black"))
}
},
col = c("B" = "black", "W" = "white"),
show_row_names = FALSE, show_column_names = FALSE,
column_title = "One famous GO game",
show_heatmap_legend = FALSE
)
lgd = Legend(title = "Players", labels = c("player1", "player2"),
    type = "points", legend_gp = gpar(col = c("black", "white")),
    size = unit(0.45, "snpc"), background = "#dcb35c")
draw(ht, heatmap_legend_list = lgd)

```



In the Go game example, we actually self-defined a legend. This will be introduced in Section 5 in more details.

2.9.2 layer_fun

If you tried out the previous code for the Go game, you might see the generation of the plot is a little bit slow that fragments in the plot come one after other other. This is because `cell_fun` is applied in every cell separately. In this section, we will introduce `layer_fun` which is a vectorized version of `cell_fun`.

`cell_fun` adds graphics cell by cell, while `layer_fun` adds graphics in a block-wise manner. Similar as `cell_fun`, `layer_fun` also needs seven arguments, but they are all in vector form (`layer_fun` can also have a eighth and ninth arguments which is introduced later in this section):

```
# code only for demonstration
Heatmap(..., layer_fun = function(j, i, x, y, w, h, fill) {...})
```

```
# or you can capitalize the arguments to mark they are vectors,
# the names of the argument do not matter
Heatmap(..., layer_fun = function(J, I, X, Y, W, H, F) {...})
```

`j` and `i` still contain the column and row indices corresponding to the original matrix, but since now `layer_fun` is applied to a heatmap slice which contains a block of cells, `j` and `i` are vectors for all the cells in the current heatmap slice. Similarly, `x`, `y`, `w`, `h` and `fill` are all vectors corresponding to all cells in the current heatmap slice.

Since `j` and `i` now are vectors, to get corresponding values in the matrix, we cannot use the form as `mat[j, i]` because it gives you a sub-matrix with `length(i)` rows and `length(j)` columns. Instead we can use `pindex()` function from `ComplexHeatmap` which is like pairwise indexing for a matrix. See follow example:

```
mfoo = matrix(1:9, nr = 3)  
mfoo[1:2, c(1, 3)]
```

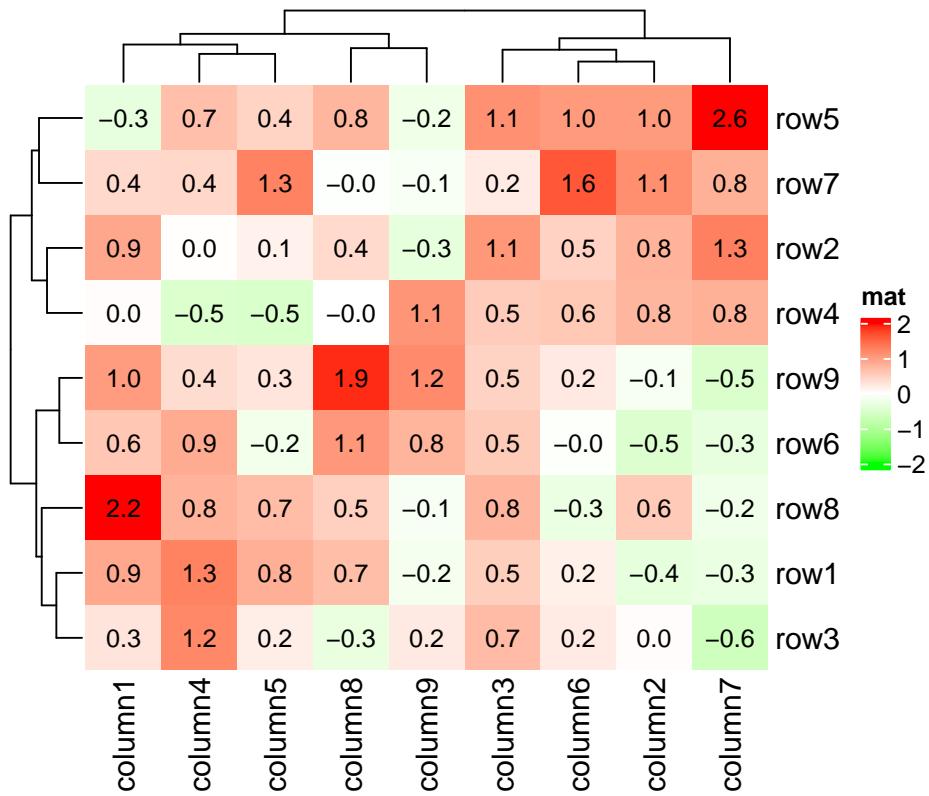
```

##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
# but we actually want mfoo[1, 1] and mfoo[2, 3]
pindex(mfoo, 1:2, c(1, 3))

```

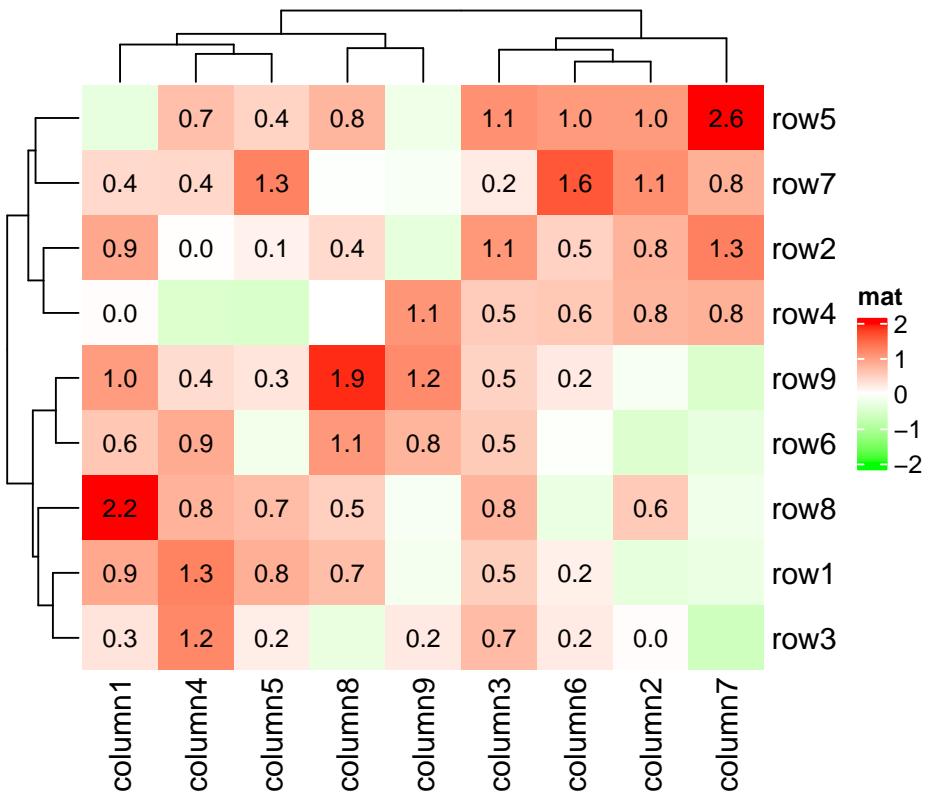
```
## [1] 1 8
```

Next example shows the `layer_fun` version of adding text on heatmap. It's basically the same as the `cell_fun` version.



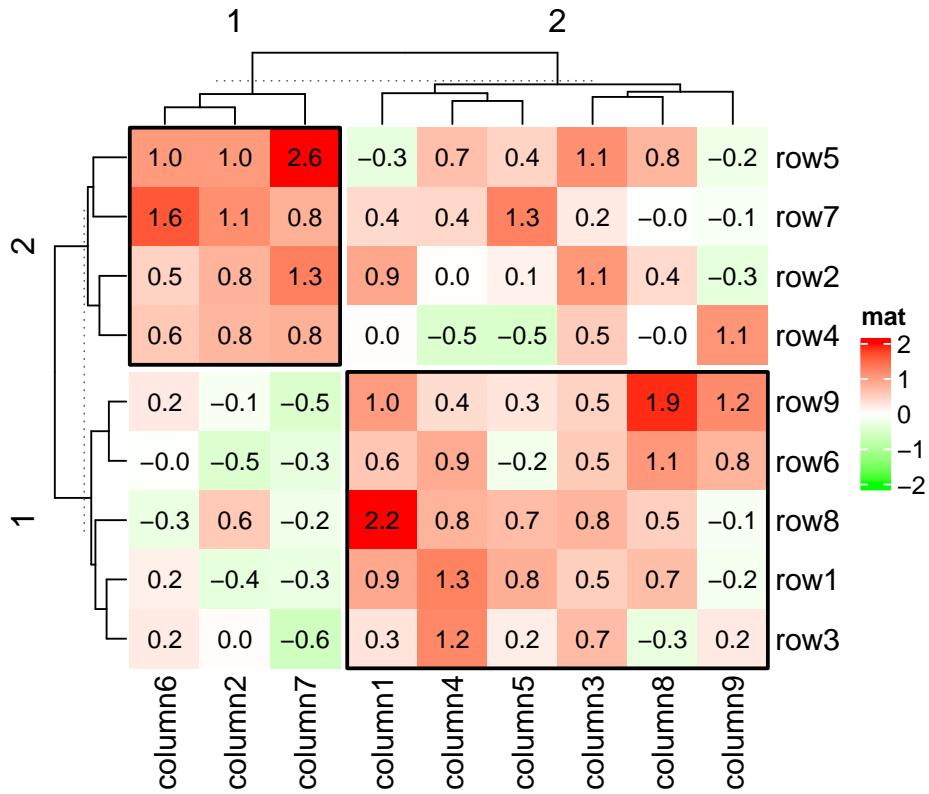
And only add text to cells with positive values:

```
Heatmap(small_mat, name = "mat", col = col_fun,
        layer_fun = function(j, i, x, y, width, height, fill) {
          v = pindex(small_mat, i, j)
          l = v > 0
          grid.text(sprintf("%.1f", v[1]), x[1], y[1], gp = gpar(fontsize = 10))
})
```



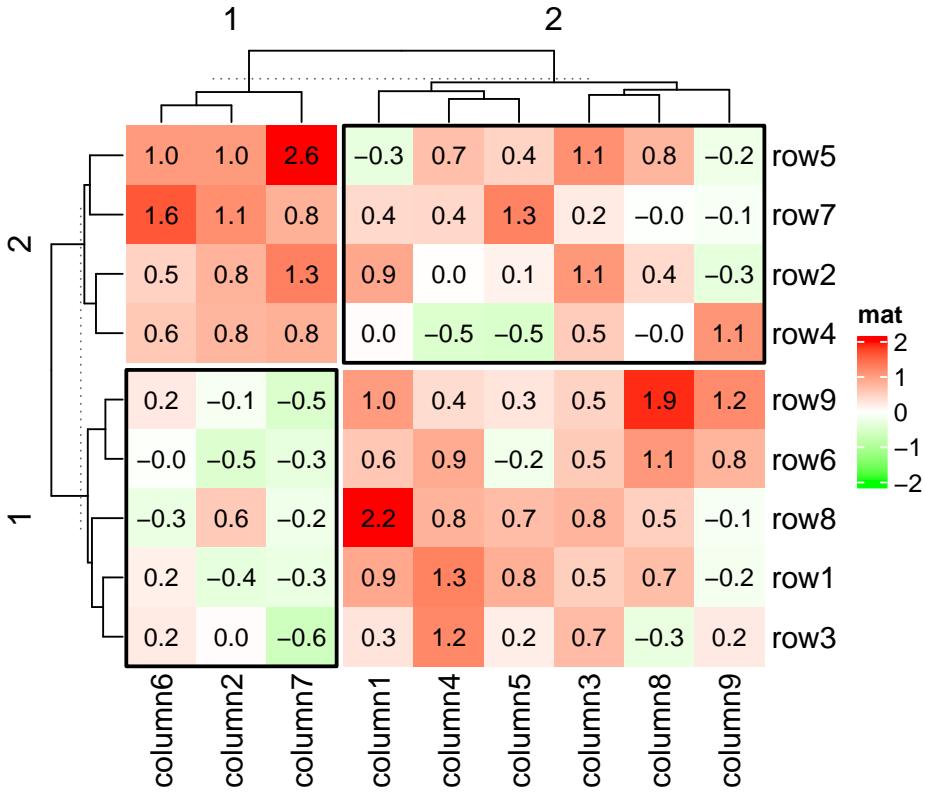
When the heatmap is split, `layer_fun` is applied in every slice.

```
Heatmap(small_mat, name = "mat", col = col_fun,
        row_km = 2, column_km = 2,
        layer_fun = function(j, i, x, y, width, height, fill) {
          v = pindex(small_mat, i, j)
          grid.text(sprintf("%.1f", v), x, y, gp = gpar(fontsize = 10))
          if(sum(v > 0)/length(v) > 0.75) {
            grid.rect(gp = gpar(lwd = 2, fill = "transparent"))
          }
      })
```



`layer_fun` can also have two more arguments which are the index for the current row slice and column slice. E.g. we want to add borders for the top right and bottom left slices.

```
Heatmap(small_mat, name = "mat", col = col_fun,
        row_km = 2, column_km = 2,
        layer_fun = function(j, i, x, y, width, height, fill, slice_r, slice_c) {
          v = pindex(small_mat, i, j)
          grid.text(sprintf("%.1f", v), x, y, gp = gpar(fontsize = 10))
          if(slice_r != slice_c) {
            grid.rect(gp = gpar(lwd = 2, fill = "transparent"))
          }
})
```



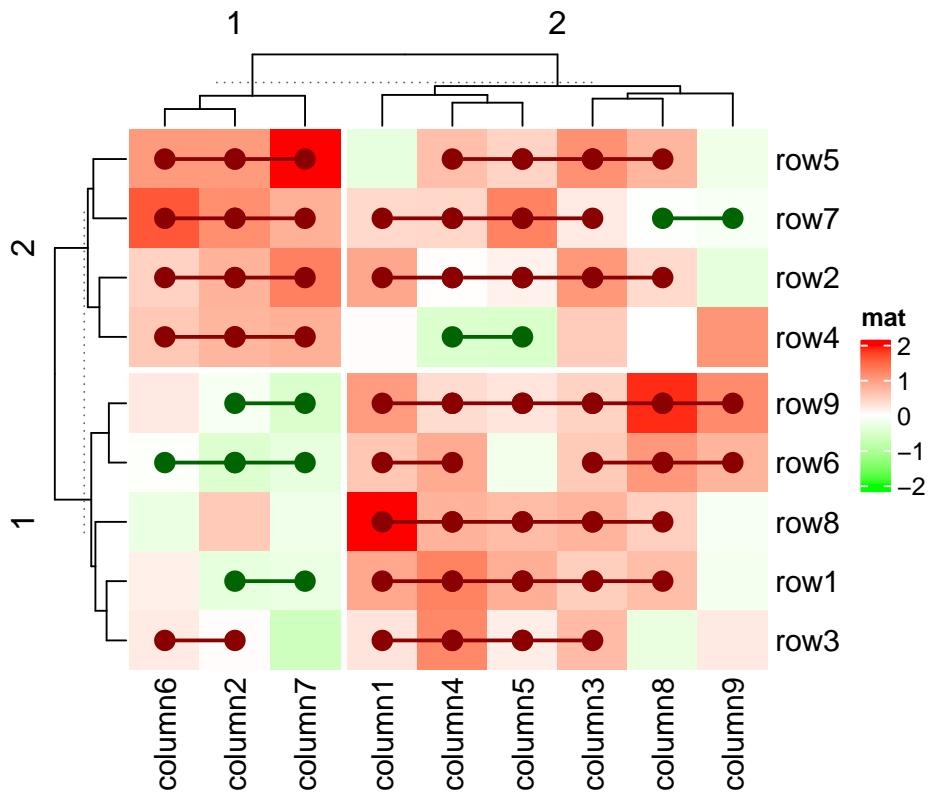
The advantage of `layer_fun` is that it is not only fast to add graphics, but also it provides more possibilities to customize the heatmap, e.g. you can interact between cells in a heatmap slice. Consider following visualization: For each row in the heatmap, if values in the neighbouring two columns have the same sign, we add a red line or a green line depending on the sign of the two values. Since now graphics in a cell depend on other cells, it is only possible to implement it via `layer_fun`. (Don't be frightened by following code. They are explained after the code.)

```
Heatmap(small_mat, name = "mat", col = col_fun,
        row_km = 2, column_km = 2,
        layer_fun = function(j, i, x, y, w, h, fill) {
            # restore_matrix() is explained after this chunk of code
            ind_mat = restore_matrix(j, i, x, y)
            for(ir in seq_len(nrow(ind_mat))) {
                # start from the second column
                for(ic in seq_len(ncol(ind_mat))[-1]) {
                    ind1 = ind_mat[ir, ic-1] # previous column
                    ind2 = ind_mat[ir, ic]    # current column
                    v1 = small_mat[i[ind1], j[ind1]]
                    v2 = small_mat[i[ind2], j[ind2]]
```

```

        if(v1 * v2 > 0) { # if they have the same sign
          col = ifelse(v1 > 0, "darkred", "darkgreen")
          grid.segments(x[ind1], y[ind1], x[ind2], y[ind2],
                        gp = gpar(col = col, lwd = 2))
          grid.points(x[c(ind1, ind2)], y[c(ind1, ind2)],
                      pch = 16, gp = gpar(col = col), size = unit(4, "mm"))
        }
      }
    }
)

```



The values that are sent to `layer_fun` are all vectors (for the vectorization of the `grid` graphics functions), however, the heatmap slice where `layer_fun` is applied to, is still represented by a matrix, thus, it would be very convinient if all the arguments in `layer_fun` can be converted to the sub-matrix for the current slice. Here, as shown in above example, `restore_matrix()` does the job. `restore_matrix()` directly accepts the first four argument in `layer_fun` and returns an index matrix, where rows and columns correspond to the rows

and columns in the current slice, from top to bottom and from left to right. The values in the matrix are the natural order of e.g. vector `j` in current slice.

If you run following code:

```
Heatmap(small_mat, name = "mat", col = col_fun,
        row_km = 2, column_km = 2,
        layer_fun = function(j, i, x, y, w, h, fill) {
          ind_mat = restore_matrix(j, i, x, y)
          print(ind_mat)
        }
      )
```

The first output which is for the top-left slice:

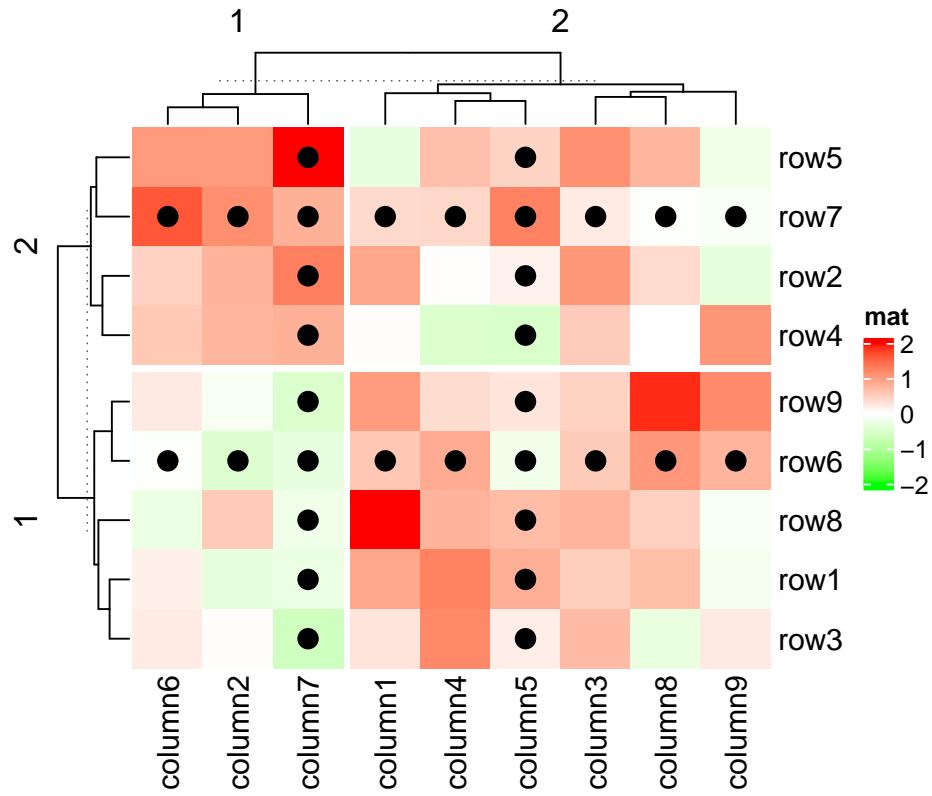
```
[,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15
```

As you see, this is a three-row and five-column index matrix where the first row corresponds to the top row in the slice. The values in the matrix correspond to the natural index (i.e. 1, 2, ...) in `j`, `i`, `x`, `y`, ... in `layer_fun`. Now, if we want to add values on the second column in the top-left slice, the code which is put inside `layer_fun` would look like:

```
for(ind in ind_mat[, 2]) {
  grid.text(small_mat[i[ind], j[ind]], x[ind], y[ind], ...)
}
```

Now it is easier to understand the second example: we want to add points to the second row and the third column in every slice:

```
Heatmap(small_mat, name = "mat", col = col_fun,
        row_km = 2, column_km = 2,
        layer_fun = function(j, i, x, y, w, h, fill) {
          ind_mat = restore_matrix(j, i, x, y)
          ind = unique(c(ind_mat[2, ], ind_mat[, 3]))
          grid.points(x[ind], y[ind], pch = 16, size = unit(4, "mm"))
        }
      )
```

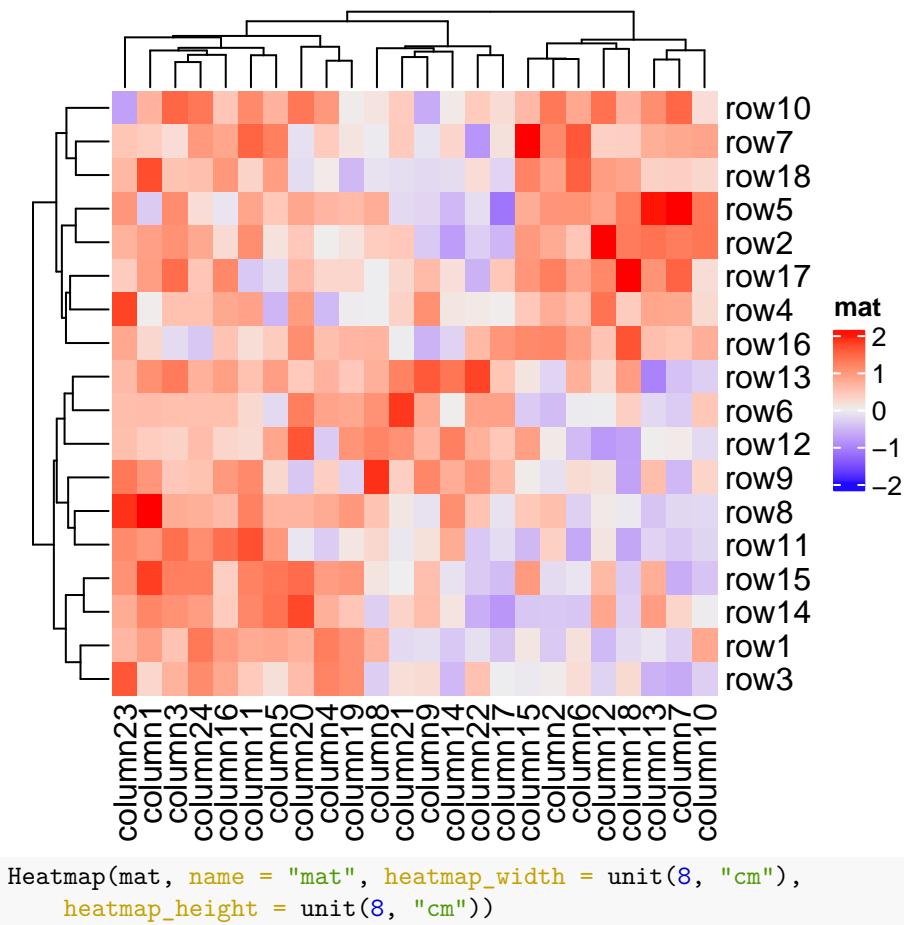


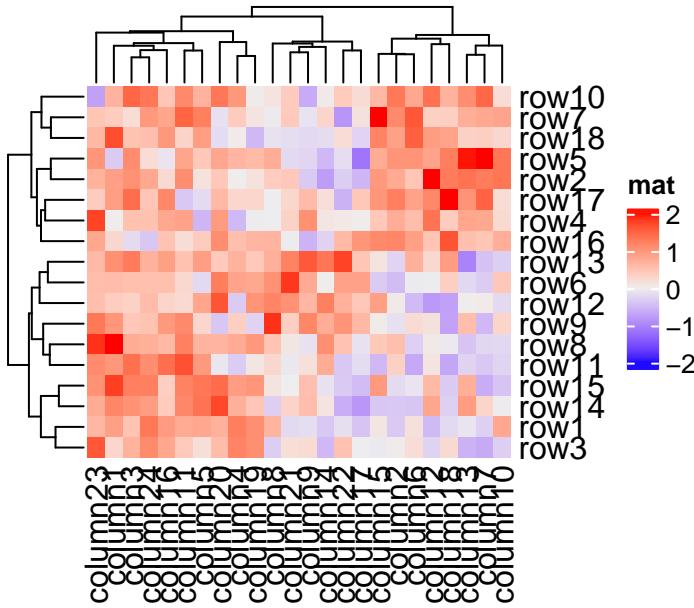
2.10 Size of the heatmap

`width`, `heatmap_width`, `height` and `heatmap_height` control the size of the heatmap. By default, all heatmap components have fixed width or height, e.g. the width of row dendrogram is 1cm. The width or the height of the heatmap body fill the rest area of the final plotting region, which means, if you draw it in an interactive graphic window and you change the size of the window by dragging it, the size of the heatmap body is automatically adjusted.

`heatmap_width` and `heatmap_height` control the width/height of the complete heatmap including all heatmap components (excluding the legends) while `width` and `height` only control the width/height of the heatmap body. All these four arguments can be set as absolute units.

```
Heatmap(mat, name = "mat", width = unit(8, "cm"), height = unit(8, "cm"))
```





These four arguments are more important when adjust the size in a list of heatmaps (see Section 4.2).

When the size of the heatmap is set as absolute units, it is possible that the size of the figure is larger than the size of the heatmap, which gives blank areas around the heatmap. The size of the heatmap can be retrieved by `ComplexHeatmap:::width()` and `ComplexHeatmap:::height()` functions.

```
ht = Heatmap(mat, name = "mat", width = unit(8, "cm"), height = unit(8, "cm"))
ht = draw(ht) # you must call draw() to reassign the heatmap variable
ComplexHeatmap:::width(ht)

## [1] 118.851591171994mm
ComplexHeatmap:::height(ht)

## [1] 114.717557838661mm
ht = Heatmap(mat, name = "mat", heatmap_width = unit(8, "cm"),
            heatmap_height = unit(8, "cm"))
ht = draw(ht)
ComplexHeatmap:::width(ht)

## [1] 94.8877245053272mm
ComplexHeatmap:::height(ht)

## [1] 83.8660578386606mm
```

Check this blog post ([Set cell width/height in the heatmap](#)) if you want to save

the heatmap in a figure file which has exactly the same size as the heatmap.

2.11 Plot the heatmap

`Heatmap()` function actually is only a constructor, which means it only puts all the data and configurations into the object in the `Heatmap` class. The clustering will only be performed when the `draw()` method is called. Under interactive mode (e.g. the interactive R terminal where you can type your R code line by line), directly calling `Heatmap()` without assigning to any object prints the object and the print method (or the S4 `show()` method) for the `Heatmap` class object calls `draw()` internally. So if you type `Heatmap(...)` in your R terminal, it looks like it is a plotting function like `plot()`, you need to be aware of that it is actually not true and in the following cases you might see nothing plotted.

- you put `Heatmap(...)` inside a function,
- you put `Heatmap(...)` in a code chunk like `for` or `if-else`
- you put `Heatmap(...)` in an Rscript and you run it under command line.

The reason is in above three cases, the `show()` method WILL NOT be called and thus `draw()` method is not executed either. So, to make the plot, you need to call `draw()` explicitly: `draw(Heatmap(...))` or:

```
# code only for demonstration
ht = Heatmap(...)
draw(ht)
```

The `draw()` function actually is applied to a list of heatmaps in `HeatmapList` class. The `draw()` method for the single `Heatmap` class constructs a `HeatmapList` with only one heatmap and call `draw()` method of the `HeatmapList` class. The `draw()` function accpets a lot of more arguments which e.g. control the legends. It will be discussed in Chapter 4.

```
draw(ht, heatmap_legend_side, padding, ...)
```

2.12 Extract orders and dendograms

The row/column orders of the heatmap can be obtained by `row_order()`/`column_order()` functions. You can directly apply to the heatmap object returned by `Heatmap()` or to the object returned by `draw()`. In following, we take `row_order()` as example.

```
small_mat = mat[1:9, 1:9]
ht1 = Heatmap(small_mat)
row_order(ht1)

## [1] 5 7 2 4 9 6 8 1 3
```

```
ht2 = draw(ht1)
row_order(ht2)

## [1] 5 7 2 4 9 6 8 1 3
```

As explained in previous section, `Heatmap()` function does not perform clustering, thus, when directly apply `row_order()` on `ht1`, clustering will be first performed. Later when making the heatmap by `draw(ht1)`, the clustering will be applied again. This might be a problem that if you set k-means clustering in the heatmap. Since the clustering is applied twice, k-means might give you different clusterings, which means, you might have different results from `row_order()` and you might have different heatmap.

In following chunk of code, `o1`, `o2` and `o3` might be different because each time, k-means clustering is performed.

```
# code only for demonstration
ht1 = Heatmap(small_mat, row_km = 2)
o1 = row_order(ht1)
o2 = row_order(ht1)
ht2 = draw(ht1)
o3 = row_order(ht2)
o4 = row_order(ht2)
```

`draw()` function returns the heatmap (or more precisely, the heatmap list) which has already been reordered, and applying `row_order()` just extracts the row order from the object, which ensures the row order is exactly the same as the one shown in the heatmap. In above code, `o3` is always identical to `o4`.

So, the preferable way to get row/column orders is as follows.

```
# code only for demonstration
ht = Heatmap(small_mat)
ht = draw(ht)
row_order(ht)
column_order(ht)
```

If rows/columns are split, row order or column order will be a list.

```
ht = Heatmap(small_mat, row_km = 2, column_km = 3)
ht = draw(ht)
row_order(ht)

## $`2`
## [1] 5 7 2 4
##
## $`1`
## [1] 9 6 8 1 3
```

```
column_order(ht)

## $`1`
## [1] 6 2 7
##
## $`3`
## [1] 1 3 4 5
##
## $`2`
## [1] 8 9
```

Similarly, the `row_dend()`/`column_dend()` functions return the dendograms. It returns a single dendrogram or a list of dendograms depending on whether the heatmap is split.

```
ht = Heatmap(small_mat, row_km = 2)
ht = draw(ht)
row_dend(ht)

## $`2`
## 'dendrogram' with 2 branches and 4 members total, at height 2.946428
##
## $`1`
## 'dendrogram' with 2 branches and 5 members total, at height 2.681351
column_dend(ht)

## 'dendrogram' with 2 branches and 9 members total, at height 4.574114
```

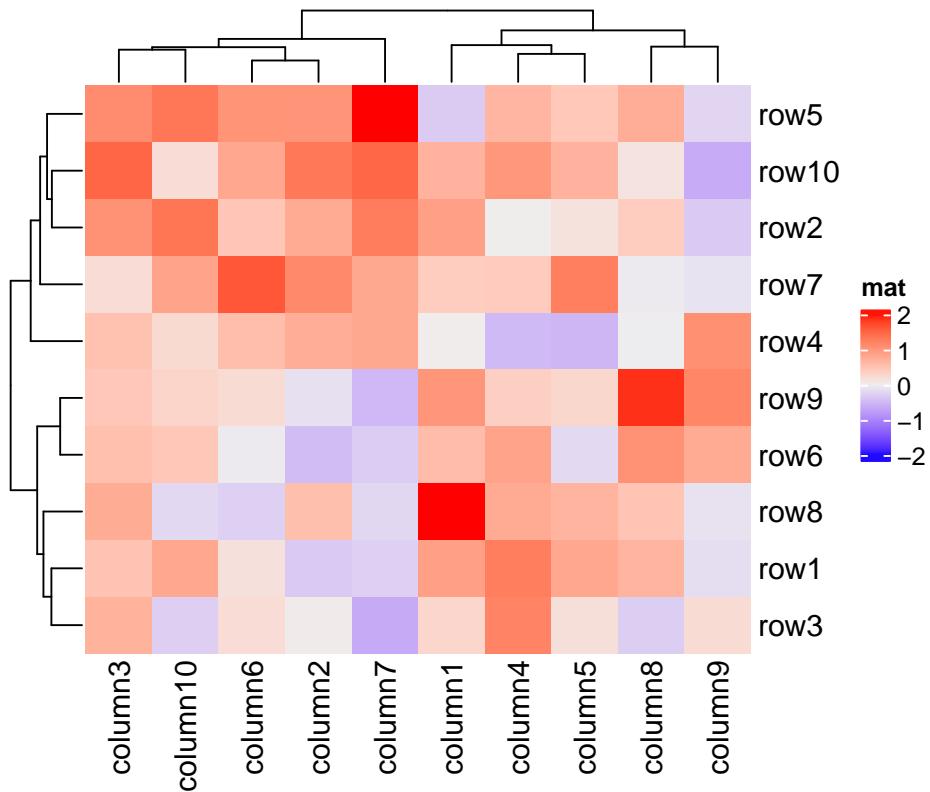
`row_order()`, `column_order()`, `row_dend()` and `column_dend()` also work for a list of heatmaps, it will be introduced in Section 4.12.

2.13 Subset a heatmap

Since heatmap is a representation of a matrix, there is also a subset method for the `Heatmap` class.

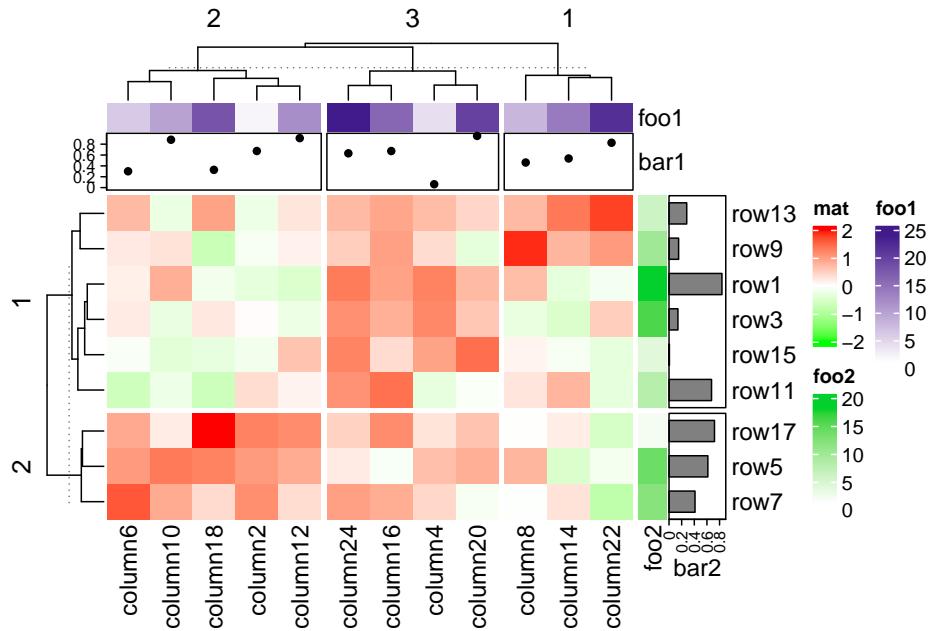
```
ht = Heatmap(mat, name = "mat")
dim(ht)

## [1] 18 24
ht[1:10, 1:10]
```



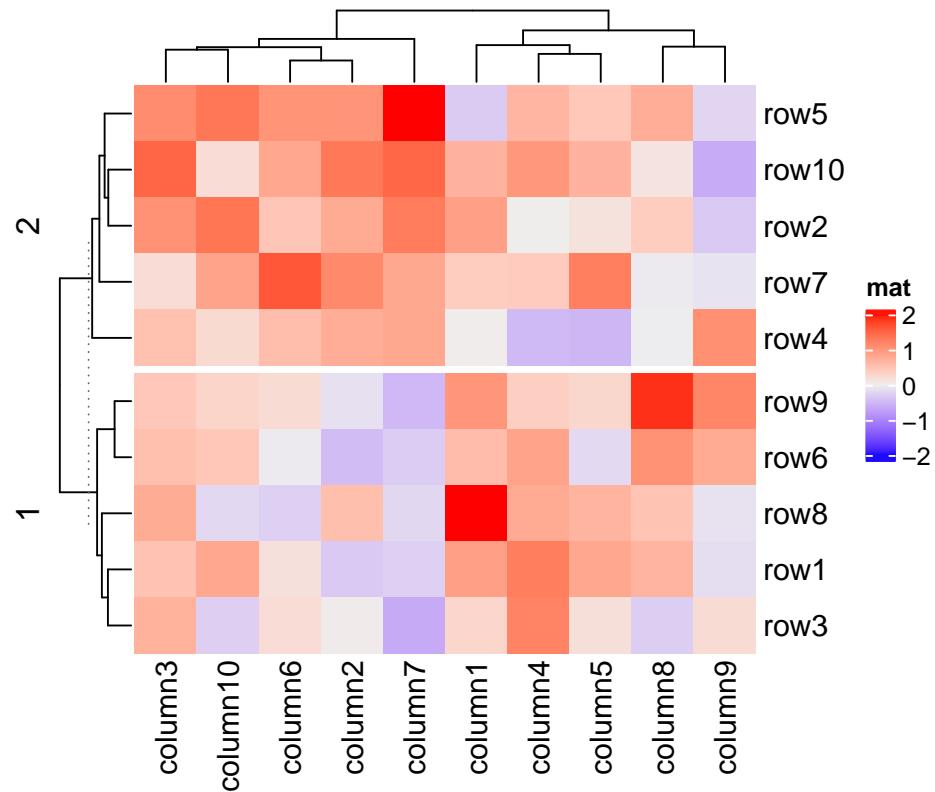
The annotations are subsetted accordingly as well.

```
ht = Heatmap(mat, name = "mat", row_km = 2, column_km = 3,
             col = colorRamp2(c(-2, 0, 2), c("green", "white", "red")),
             top_annotation = HeatmapAnnotation(foo1 = 1:24, bar1 = anno_points(runif(24))),
             right_annotation = rowAnnotation(foo2 = 18:1, bar2 = anno_barplot(runif(18)))
)
ht[1:9*2 - 1, 1:12*2] # odd rows, even columns
```



The heatmap components are subsetted if they are vector-like. Some configurations in the heatmap keep the same when subsetting, e.g. if `row_km` is set in the original heatmap, the configuration of k-means is kept and it is performed in the sub-heatmap. So in following example, k-means clustering is only performed when making heatmap for `ht2`.

```
ht = Heatmap(mat, name = "mat", row_km = 2)
ht2 = ht[1:10, 1:10]
ht2
```



The implementation of subsetting heatmaps is very experimental. It is not always working, e.g. if `cell_fun` is defined and uses an external matrix, or clustering objects are assigned to `cluster_rows` or `cluster_columns`.

There are also subset methods for the `HeatmapAnnotation` class (Section 3.22) and the `HeatmapList` class (Section 4.10), but both are very experimental as well.

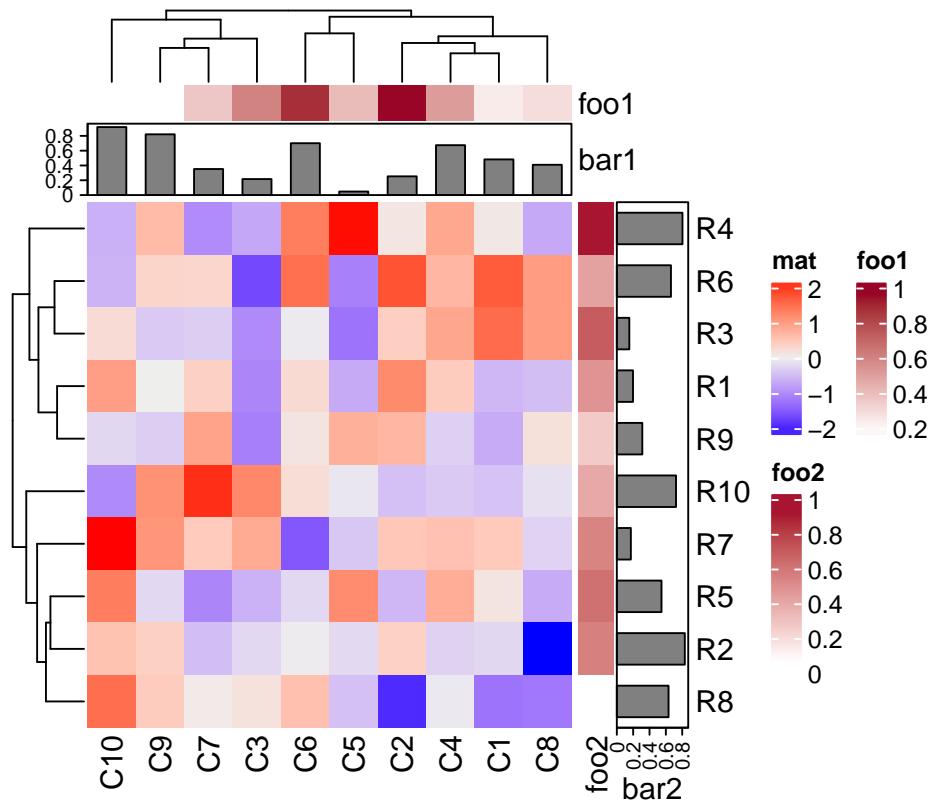
Chapter 3

Heatmap Annotations

Heatmap annotations are important components of a heatmap that it shows additional information that associates with rows or columns in the heatmap. **ComplexHeatmap** package provides very flexible supports for setting annotations and defining new annotation graphics. The annotations can be put on the four sides of the heatmap, by `top_annotation`, `bottom_annotation`, `left_annotation` and `right_annotation` arguments.

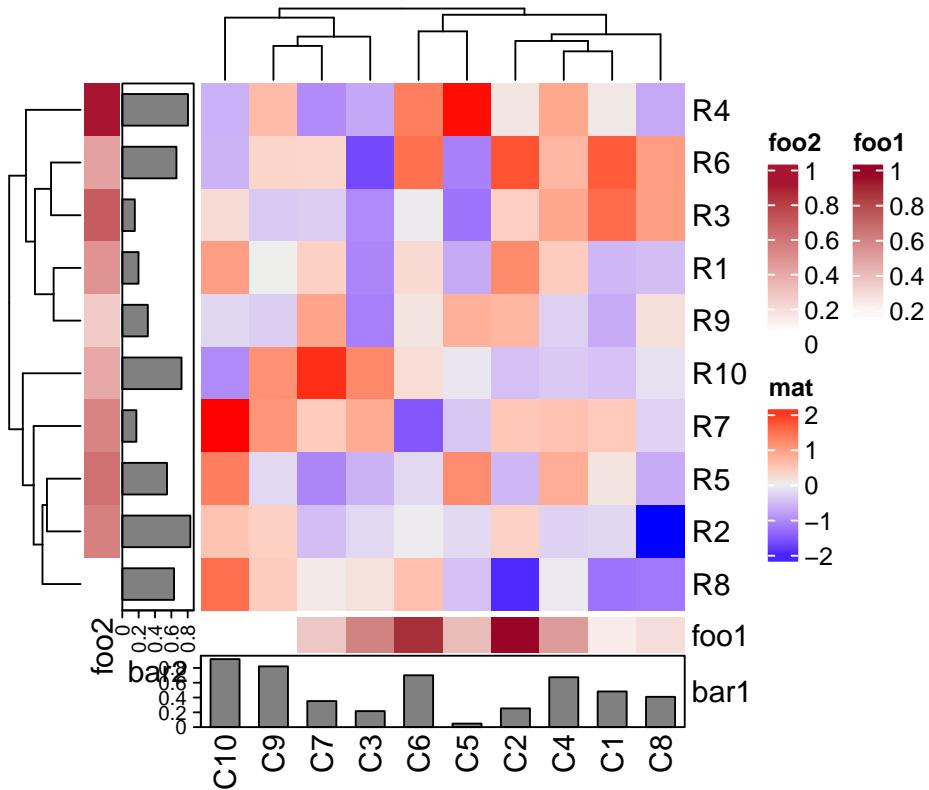
The value for the four arguments should be in the `HeatmapAnnotation` class and should be constructed by `HeatmapAnnotation()` function, or by `rowAnnotation()` function if it is a row annotation. (`rowAnnotation()` is just a helper function which is identical to `HeatmapAnnotation(..., which = "row")`). A simple usage of heatmap annotations is as follows.

```
set.seed(123)
mat = matrix(rnorm(100), 10)
rownames(mat) = paste0("R", 1:10)
colnames(mat) = paste0("C", 1:10)
column_ha = HeatmapAnnotation(foo1 = runif(10), bar1 = anno_barplot(runif(10)))
row_ha = rowAnnotation(foo2 = runif(10), bar2 = anno_barplot(runif(10)))
Heatmap(mat, name = "mat", top_annotation = column_ha, right_annotation = row_ha)
```



Or assign as bottom annotation and left annotation.

```
Heatmap(mat, name = "mat", bottom_annotation = column_ha, left_annotation = row_ha)
```



In above examples, `column_ha` and `row_ha` both have two annotations where `foo1` and `foo2` are numeric vectors and `bar1` and `bar2` are barplots. The vector-like annotation is called “*simple annotation*” here and the barplot annotation is called “*complex annotation*”. You can already see the annotations must be defined as name-value pairs (e.g. `foo = ...`).

Heatmap annotations can also be independent of the heatmaps. They can be concatenated to the heatmap list by `+` if it is horizontal, or `%v%` if it is vertical. Chapter 4 will discuss how to concatenate heatmaps and annotations.

```
# code only for demonstration
Heatmap(...) + rowAnnotation() + ...
Heatmap(...) %v% HeatmapAnnotation(...) %v% ...
```

`HeatmapAnnotation()` returns a `HeatmapAnnotation` class object. The object is usually composed of several annotations. In the following sections of this chapter, we first introduce settings for individual annotation, and later we show how to put them together.

You can see the information of the `column_ha` and `row_ha` objects by directly enter the object names:

```

column_ha

## A HeatmapAnnotation object with 2 annotations
##   name: heatmap_annotation_0
##   position: column
##   items: 10
##   width: 1npc
##   height: 15.3514598035146mm
##   this object is subsettable
##   5.9228888888889mm extension on the left
##   9.4709mm extension on the right
##
##   name   annotation_type color_mapping height
##   foo1  continuous vector    random      5mm
##   bar1  anno_barplot()        10mm

row_ha

## A HeatmapAnnotation object with 2 annotations
##   name: heatmap_annotation_1
##   position: row
##   items: 10
##   width: 15.3514598035146mm
##   height: 1npc
##   this object is subsettable
##   9.96242222222222mm extension on the bottom
##
##   name   annotation_type color_mapping width
##   foo2  continuous vector    random      5mm
##   bar2  anno_barplot()        10mm

```

In the following examples in this chapter, we will only show the graphics for the annotations with no heatmap, unless it is necessary. If you want to try it with a heatmap, you just assign the `HeatmapAnnotation` object which we always name as `ha` to `top_annotation`, `bottom_annotation`, `left_annotation` or `right_annotation` arguments.

Settings are basically the same for column annotations and row annotations. If there is nothing special, we only show the column annotation as examples. If you want to try row annotation, just add `which = "row"` to `HeatmapAnnotation()` or directly change to `rowAnnotation()` function.

3.1 Simple annotation

A so-called “*simple annotation*” is the most used style of annotations which is heatmap-like or grid-like graphics where colors are used to map to the annotation

values. To generate a simple annotation, you just simply put the annotation vector in `HeatmapAnnotation()` with a certain name.

```
ha = HeatmapAnnotation(foo = 1:10)
```



Or a discrete annotation:

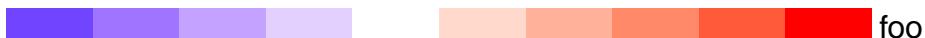
```
ha = HeatmapAnnotation(bar = sample(letters[1:3], 10, replace = TRUE))
```



You can use any strings as annotation names except those pre-defined arguments in `HeatmapAnnotation()`.

If colors are not specified, colors are randomly generated. To set colors for annotations, `col` needs to be set as a named list where the names should be the same as annotation names. For continuous values, the color mapping should be a color mapping function generated by `circlize::colorRamp2()`.

```
library(circlize)
col_fun = colorRamp2(c(0, 5, 10), c("blue", "white", "red"))
ha = HeatmapAnnotation(foo = 1:10, col = list(foo = col_fun))
```



And for discrete annotations, the color should be a named vector where names correspond to the levels in the annotation.

```
ha = HeatmapAnnotation(bar = sample(letters[1:3], 10, replace = TRUE),
                      col = list(bar = c("a" = "red", "b" = "green", "c" = "blue")))
```



If you specify more than one vectors, there will be multiple annotations (`foo` and `bar` in following example). Also you can see how `col` is set when `foo` and `bar` are all put into a single `HeatmapAnnotation()`. Maybe now you can understand the names in the color list is actually used to map to the annotation names. Values in `col` will be used to construct legends for simple annotations.

```
ha = HeatmapAnnotation(
  foo = 1:10,
  bar = sample(letters[1:3], 10, replace = TRUE),
  col = list(foo = col_fun,
             bar = c("a" = "red", "b" = "green", "c" = "blue"))
```

```
)  
)
```



The color for NA value is controlled by `na_col` argument.

```
ha = HeatmapAnnotation(  
  foo = c(1:4, NA, 6:10),  
  bar = c(NA, sample(letters[1:3], 9, replace = TRUE)),  
  col = list(foo = col_fun,  
            bar = c("a" = "red", "b" = "green", "c" = "blue"))  
,  
  na_col = "black"  
)
```



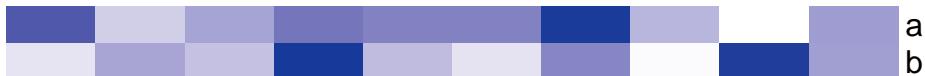
`gp` mainly controls the graphics parameters for the borders of the grids.

```
ha = HeatmapAnnotation(  
  foo = 1:10,  
  bar = sample(letters[1:3], 10, replace = TRUE),  
  col = list(foo = col_fun,  
            bar = c("a" = "red", "b" = "green", "c" = "blue"))  
,  
  gp = gpar(col = "black")  
)
```



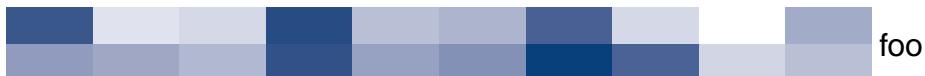
The simple annotation can also be a matrix (numeric or character) that all the columns in the matrix share a same color mapping schema. **Note columns in the matrix correspond to the rows in the column annotation** (you can imagine a vector is a one-column matrix). Also the column names of the matrix are used as the annotation names.

```
ha = HeatmapAnnotation(foo = cbind(a = runif(10), b = runif(10)))
```



If the matrix has no column name, the name of the annotation is still used, but drawn in the middle of the annotation.

```
ha = HeatmapAnnotation(foo = cbind(runif(10), runif(10)))
```



As simple annotations can be in different modes (e.g. numeric, or character), they can be combined as a data frame and send to df argument. Imagine in your project, you might already have an annotation table, you can directly set it by df.

```
anno_df = data.frame(
  foo = 1:10,
  bar = sample(letters[1:3], 10, replace = TRUE)
)
ha = HeatmapAnnotation(df = anno_df,
  col = list(foo = col_fun,
             bar = c("a" = "red", "b" = "green", "c" = "blue"))
)
```



Single annotations and data frame can be mixed, but single annotations are inserted after the data frame annotation. In following example, colors for foo2 is not specified, random colors will be used.

```
ha = HeatmapAnnotation(df = anno_df,
  foo2 = rnorm(10),
  col = list(foo = col_fun,
             bar = c("a" = "red", "b" = "green", "c" = "blue"))
)
```



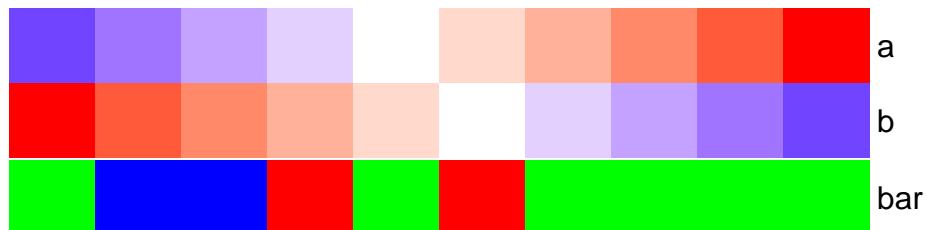
`border` controls the border of every single annotation.

```
ha = HeatmapAnnotation(
  foo = cbind(1:10, 10:1),
  bar = sample(letters[1:3], 10, replace = TRUE),
  col = list(foo = col_fun,
             bar = c("a" = "red", "b" = "green", "c" = "blue"))
),
border = TRUE
)
```



The height of the simple annotation is controlled by `simple_anno_size` argument. Since all single annotations have same height, the value of `simple_anno_size` is a single unit value. Note there are arguments like `width`, `height`, `annotation_width` and `annotation_height`, but they are used to adjust the width/height for the complete heatmap annotations (which are always mix of several annotations). The adjustment of these four arguments will be introduced in Section 3.21.

```
ha = HeatmapAnnotation(
  foo = cbind(a = 1:10, b = 10:1),
  bar = sample(letters[1:3], 10, replace = TRUE),
  col = list(foo = col_fun,
             bar = c("a" = "red", "b" = "green", "c" = "blue"))
),
simple_anno_size = unit(1, "cm")
)
```



When you have multiple heatmaps and it is better to keep the size of simple annotations on all heatmaps with the same size. `ht_opt$simple_anno_size` can be set to control the simple annotation size globally (It will be introduced in Section 4.13).

3.2 Simple annotation as an annotation function

`HeatmapAnnotation()` supports “*complex annotation*” by setting the annotation as a function. The annotation function defines how to draw the graphics at a certain position corresponding to the column or row in the heatmap. There are quite a lot of annotation functions predefined in **ComplexHeatmap** package. In the end of this chapter, we will introduce how to construct your own annotation function by the `AnnotationFunction` class.

For all the annotation functions in forms of `anno_*`(`), if it is specified in HeatmapAnnotation() or rowAnnotation(), you don't need to do anything explicitly on anno_*() to tell whether it should be drawn on rows or columns. anno_*() automatically detects whether it is a row annotation environment or a column annotation environment.`

The simple annotation shown in previous section is internally constructed by `anno_simple()` annotation function. Directly using `anno_simple()` will not automatically generate legends for the final plot, but, it can provide more flexibility for more annotation graphics (note in Chapter 5 we will show, although `anno_simple()` cannot automatically generate the legends, the legends can be controlled and added to the final plot manually).

For an example in previous section:

```
# code only for demonstration
ha = HeatmapAnnotation(foo = 1:10)
```

is actually identical to:

```
# code only for demonstration
ha = HeatmapAnnotation(foo = anno_simple(1:10))
```

`anno_simple()` makes heatmap-like annotations (or the simple annotations). Basically if users only make heatmap-like annotations, they do not need to directly use `anno_simple()`, but this function allows to add more symbols on the annotation grids.

`anno_simple()` allows to add “points” or single-letter symbols on top of the annotation grids. `pch`, `pt_gp` and `pt_size` control the settings of the points. The value of `pch` can be a vector with possible NA values.

```
ha = HeatmapAnnotation(foo = anno_simple(1:10, pch = 1,
                                         pt_gp = gpar(col = "red"), pt_size = unit(1:10, "mm")))
```



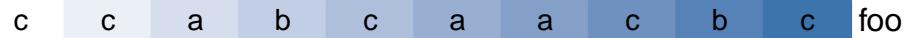
Set `pch` as a vector:

```
ha = HeatmapAnnotation(foo = anno_simple(1:10, pch = 1:10))
```



Set pch as a vector of letters:

```
ha = HeatmapAnnotation(foo = anno_simple(1:10,
                                         pch = sample(letters[1:3], 10, replace = TRUE)))
```



Set pch as a vector with NA values (nothing is drawn for NA pch values):

```
ha = HeatmapAnnotation(foo = anno_simple(1:10, pch = c(1:4, NA, 6:8, NA, 10, 11)))
```



pch also works if the value for `anno_simple()` is a matrix. The length of pch should be as same as the number of matrix rows or columns or even the length of the matrix (the length of the matrix is the length of all data points in the matrix).

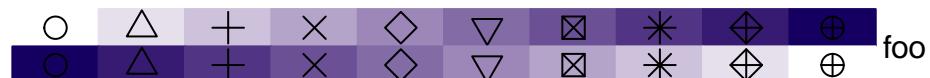
Length of pch corresponds to matrix columns:

```
ha = HeatmapAnnotation(foo = anno_simple(cbind(1:10, 10:1), pch = 1:2))
```



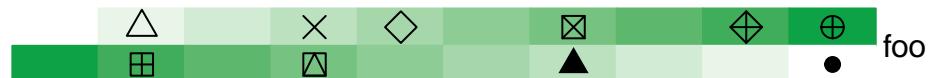
Length of pch corresponds to matrix rows:

```
ha = HeatmapAnnotation(foo = anno_simple(cbind(1:10, 10:1), pch = 1:10))
```



pch is a matrix:

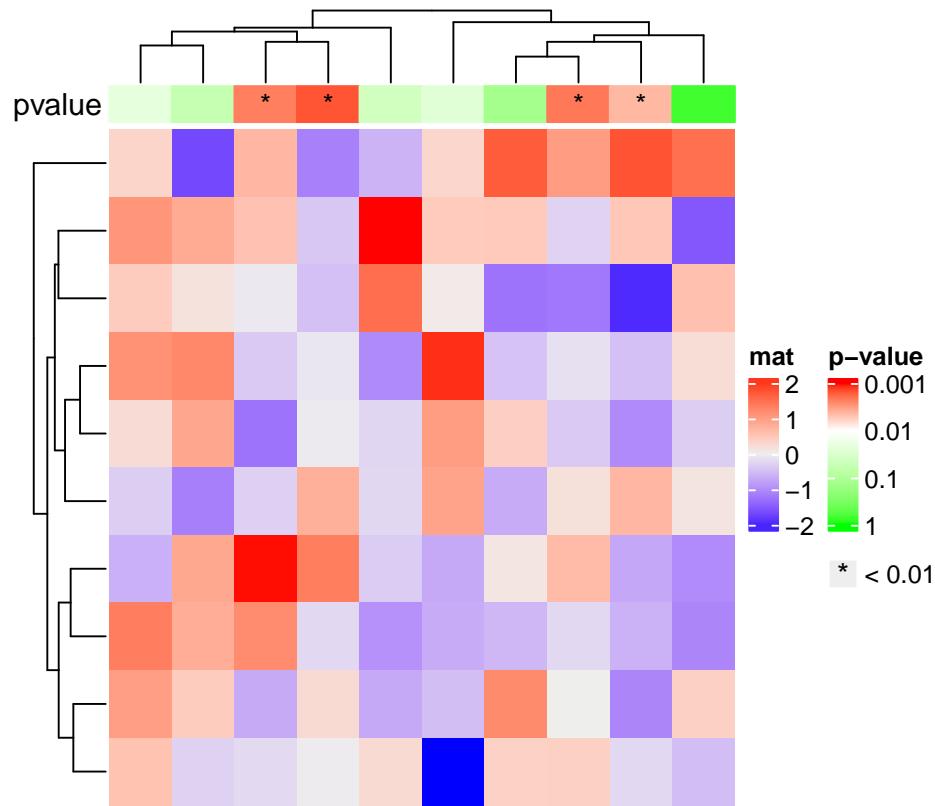
```
pch = matrix(1:20, nc = 2)
pch[sample(length(pch), 10)] = NA
ha = HeatmapAnnotation(foo = anno_simple(cbind(1:10, 10:1), pch = pch))
```



Till now, you might wonder how to set the legends of the symbols you've added to the simple annotations. Here we will only show you a simple example and this functionality will be discussed in Chapter 5. In following example, we assume

the simple annotations are kind of p-values and we add * for p-values less than 0.01.

```
set.seed(123)
pvalue = 10^-runif(10, min = 0, max = 3)
is_sig = pvalue < 0.01
pch = rep("*", 10)
pch[!is_sig] = NA
# color mapping for -log10(pvalue)
pvalue_col_fun = colorRamp2(c(0, 2, 3), c("green", "white", "red"))
ha = HeatmapAnnotation(
    pvalue = anno_simple(-log10(pvalue), col = pvalue_col_fun, pch = pch),
    annotation_name_side = "left")
ht = Heatmap(matrix(rnorm(100), 10), name = "mat", top_annotation = ha)
# now we generate two legends, one for the p-value
# see how we define the legend for pvalue
lgd_pvalue = Legend(title = "p-value", col_fun = pvalue_col_fun, at = c(0, 1, 2, 3),
    labels = c("1", "0.1", "0.01", "0.001"))
# and one for the significant p-values
lgd_sig = Legend(pch = "*", type = "points", labels = "< 0.01")
# these two self-defined legends are added to the plot by `annotation_legend_list`
draw(ht, annotation_legend_list = list(lgd_pvalue, lgd_sig))
```

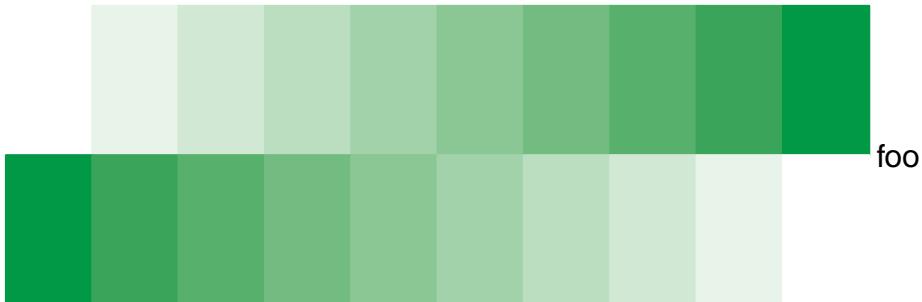


The height of the simple annotation can be controlled by `height` argument or `simple_anno_size` inside `anno_simple()`. `simple_anno_size` controls the size for single-row annotation and `height/width` controls the total height/width of the simple annotations. If `height/width` is set, `simple_anno_size` is ignored.

```
ha = HeatmapAnnotation(foo = anno_simple(1:10, height = unit(2, "cm")))
```



```
ha = HeatmapAnnotation(foo = anno_simple(cbind(1:10, 10:1),  
    simple_anno_size = unit(2, "cm")))
```



For all the annotation functions we introduce later, the height or the width for individual annotations should all be set inside the `anno_*`() functions.

```
# code only for demonstration
anno_*(..., width = ...)
anno_*(..., height = ...)
```

Again, the `width`, `height`, `annotation_width` and `annotation_height` arguments in `HeatmapAnnotation()` are used to adjust the size of multiple annotations.

3.3 Empty annotation

`anno_empty()` is a place holder where nothing is drawn. Later user-defined graphics can be added by `decorate_annotation()` function.

```
ha = HeatmapAnnotation(foo = anno_empty(border = TRUE))
```

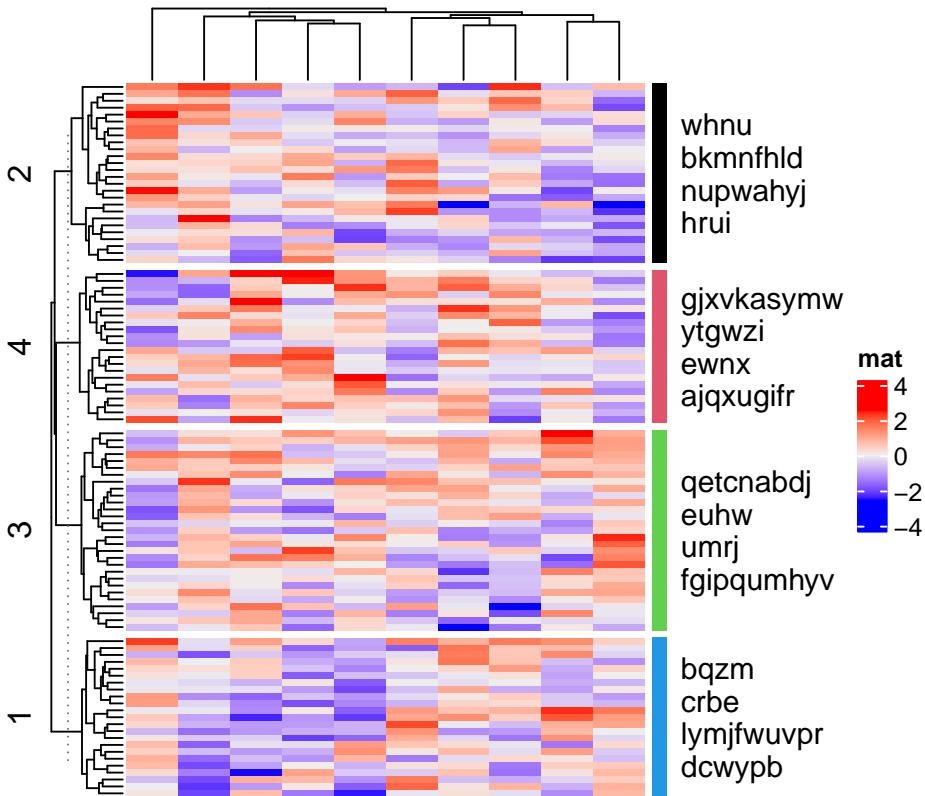


In Chapter 6, we will introduce the use of the decoration functions, but here we give a quick example. In gene expression expression analysis, there are scenarios that we split the heatmaps into several groups and we want to highlight some key genes in each group. In this case, we simply add the gene names on the right side of the heatmap without aligning them to their corresponding rows. (`anno_mark()` can align the labels correctly to their corresponding rows, but in the example we show here, it is not necessary).

In following example, since rows are split into four slices, the empty annotation is also split into four slices. Basically what we do is in each empty annotation slice, we add a colored segment and text.

```
random_text = function(n) {
  sapply(1:n, function(i) {
```

```
        paste0(sample(letters, sample(4:10, 1)), collapse = ""))
    })
}
text_list = list(
  text1 = random_text(4),
  text2 = random_text(4),
  text3 = random_text(4),
  text4 = random_text(4)
)
# note how we set the width of this empty annotation
ha = rowAnnotation(foo = anno_empty(border = FALSE,
  width = max_text_width(unlist(text_list)) + unit(4, "mm")))
Heatmap(matrix(rnorm(1000), nrow = 100), name = "mat", row_km = 4, right_annotation = ha)
for(i in 1:4) {
  decorate_annotation("foo", slice = i, {
    grid.rect(x = 0, width = unit(2, "mm"), gp = gpar(fill = i, col = NA), just = "left")
    grid.text(paste(text_list[[i]]), collapse = "\n"), x = unit(4, "mm"), just = "left")
  })
}
```



Note the previous plot can also be made by `anno_block()` or `anno_text_box()` (Section 3.19).

A second use of the empty annotation is to add complex annotation graphics where the empty annotation pretends to be a virtual plotting region. You can construct an annotation function by `AnnotationFunction` class for complex annotation graphics, which allows subsetting and splitting, but still, it can be a secondary choice to directly draw inside the empty annotation, which is easier and faster for implementing (but less flexible and does not allow splitting).

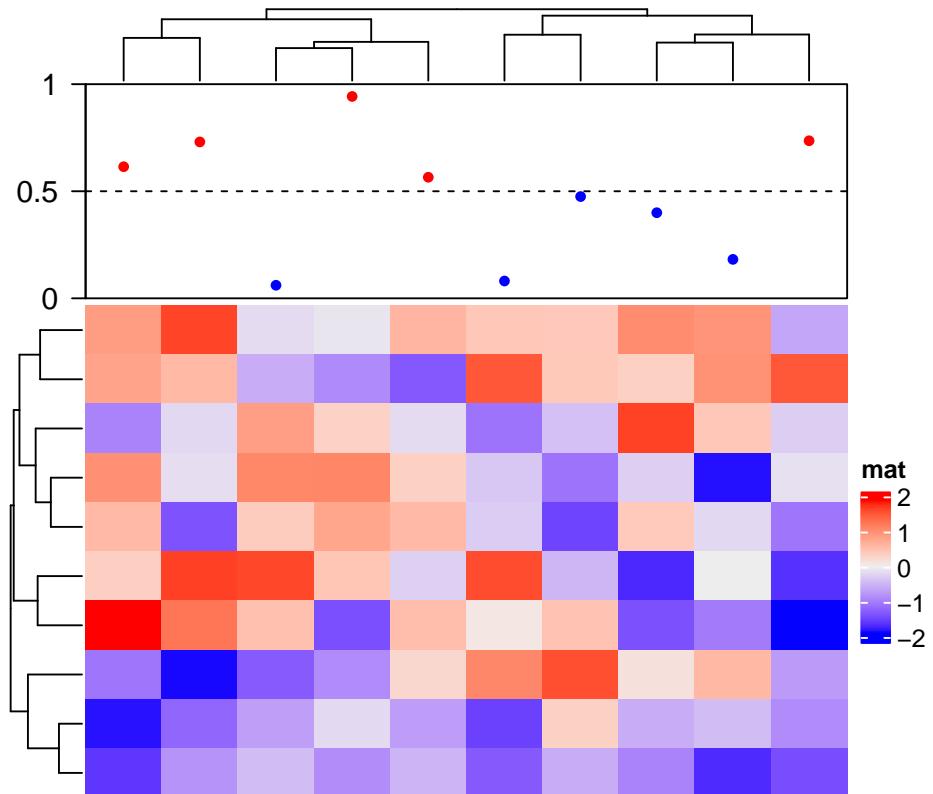
In following we show how to add a “complex version” of points annotation. The only thing that needs to be careful is the location on x-axis (y-axis if it is a row annotation) should correspond to the column index after column reordering.

```
# Note this example is only for demonstration of `anno_empty()`.  
# Actually it can be made easily by `anno_points()`.  
ha = HeatmapAnnotation(foo = anno_empty(border = TRUE, height = unit(3, "cm")))  
ht = Heatmap(matrix(rnorm(100), nrow = 10), name = "mat", top_annotation = ha)  
ht = draw(ht)  
co = column_order(ht)  
value = runif(10)
```

```

decorate_annotation("foo", {
  # value on x-axis is always 1:ncol(mat)
  x = 1:10
  # while values on y-axis is the value after column reordering
  value = value[co]
  pushViewport(viewport(xscale = c(0.5, 10.5), yscale = c(0, 1)))
  grid.lines(c(0.5, 10.5), c(0.5, 0.5), gp = gpar(lty = 2),
    default.units = "native")
  grid.points(x, value, pch = 16, size = unit(2, "mm"),
    gp = gpar(col = ifelse(value > 0.5, "red", "blue")), default.units = "native")
  grid.yaxis(at = c(0, 0.5, 1))
  popViewport()
})

```



3.4 Block annotation

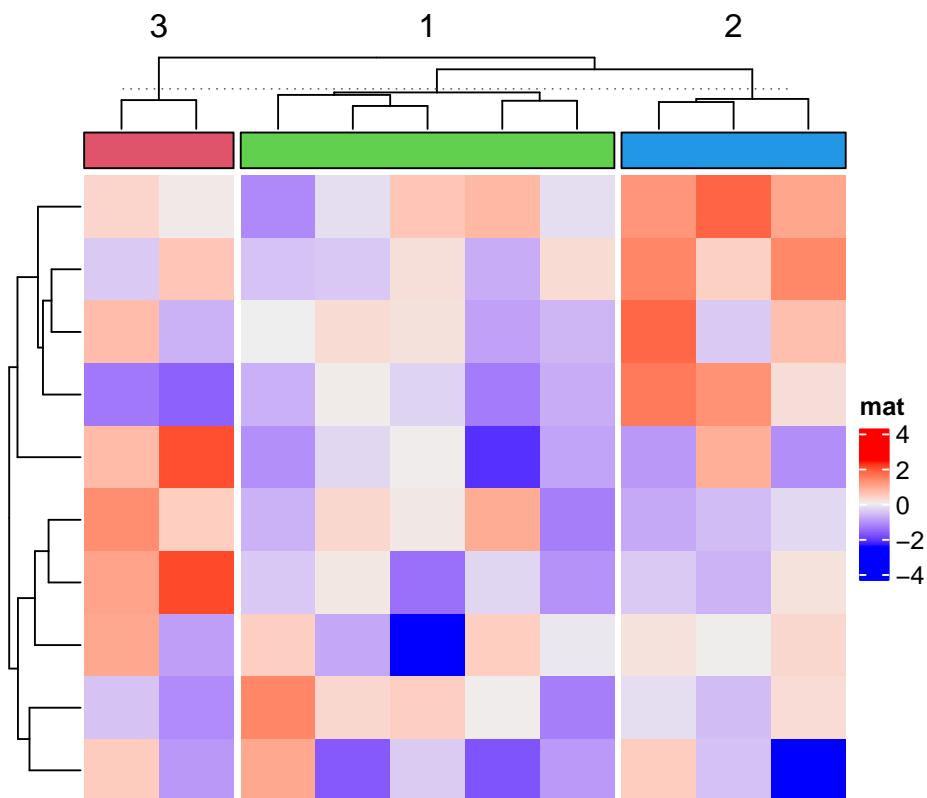
There are two uses for block annotation. 1. simply as rectangles (with labels inside) to mark heatmap slices, 2. as plotting regions to associate subsets of

rows or columns in the heatmap.

3.4.1 Block for putting labels

In this case, the block annotation is more like a color block which identifies groups when the rows or columns of the heatmap are split.

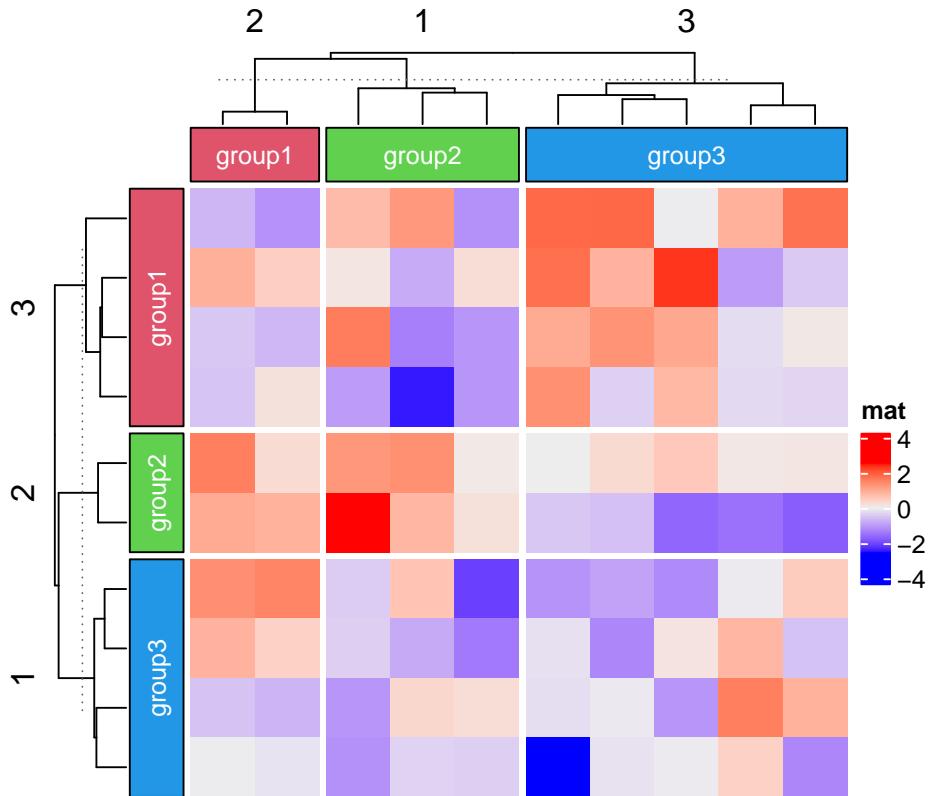
```
Heatmap(matrix(rnorm(100), 10), name = "mat",
        top_annotation = HeatmapAnnotation(foo = anno_block(gp = gpar(fill = 2:4))),
        column_km = 3)
```



Labels can be added to each block.

```
Heatmap(matrix(rnorm(100), 10), name = "mat",
        top_annotation = HeatmapAnnotation(foo = anno_block(gp = gpar(fill = 2:4),
                                                       labels = c("group1", "group2", "group3"),
                                                       labels_gp = gpar(col = "white", fontsize = 10)),
        column_km = 3,
        left_annotation = rowAnnotation(foo = anno_block(gp = gpar(fill = 2:4),
                                                       labels = c("group1", "group2", "group3")),
```

```
labels_gp = gpar(col = "white", fontsize = 10)),
row_km = 3)
```



Note the length of `labels` or graphics parameters should have the same length as the number of slices.

`anno_block()` function draws rectangles for row/column slices where one rectangle only corresponds to one single slice. Then what if we want to draw the rectangles over several slices to show they belong to certain groups?

Currently, it is difficult to directly support it in `anno_block()`, however, there is workaround for it. Actually, to draw rectangles across several slices, we need to know two things: 1. the positions of the slices in the plot, and 2. space to draw the rectangles. Luckily, the positions can be obtained by directly go to the corresponding viewport and the space can be allocated by `anno_empty()` function.

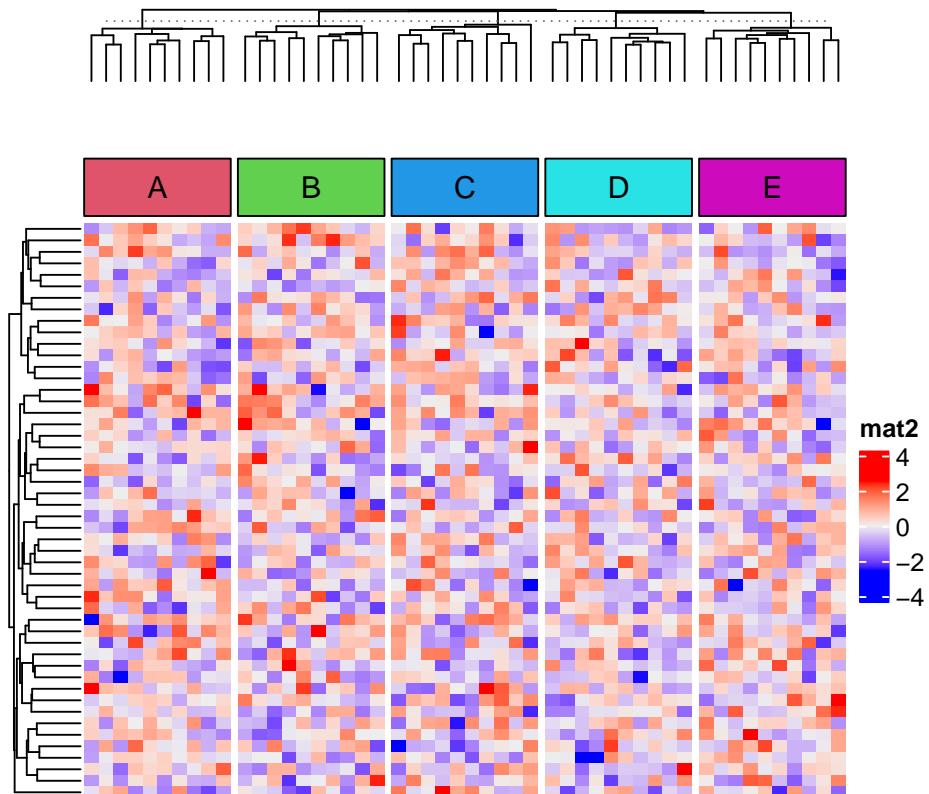
In the following code, we use `anno_empty()` to create an empty annotation:

```
set.seed(123)
mat2 = matrix(rnorm(50*50), nrow = 50)
```

```

split = rep(1:5, each = 10)
ha = HeatmapAnnotation(
  empty = anno_empty(border = FALSE),
  foo = anno_block(gp = gpar(fill = 2:6), labels = LETTERS[1:5])
)
Heatmap(mat2, name = "mat2", column_split = split, top_annotation = ha,
        column_title = NULL)

```



Let's say, we want to put the first three column slices as a group and the last two slices as the second group.

The positions of the first and the third slices for annotation "empty" can be obtained by:

```

seekViewport("annotation_empty_1")
loc1 = deviceLoc(x = unit(0, "npc"), y = unit(0, "npc"))
seekViewport("annotation_empty_3")
loc2 = deviceLoc(x = unit(1, "npc"), y = unit(1, "npc"))
loc2

```

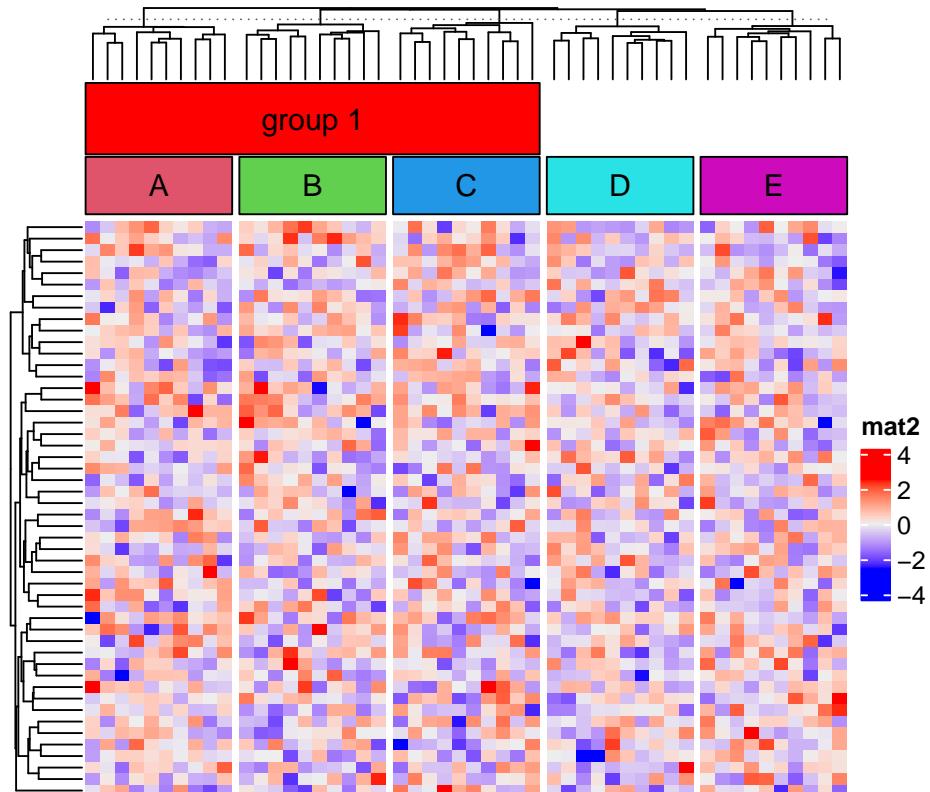
```
## $x
## [1] 4.07403126835173inches
##
## $y
## [1] 6.51051067246731inches
```

The viewport name "annotation_empty_1" correspond to the first slice for annotation `empty`, and we take the left bottom of the first "empty" annotation slice and the top right of the third slice, saved in `loc1` and `loc2` variables.

Here what is important is the use of `grid::deviceLoc()` function. It directly converts a location measured in a certain viewport to the position in the graphics device.

In the end, we go to the "global" viewport because the size of "global" viewport is the size of the graphics device, and draw the rectangle and add label.

```
seekViewport("global")
grid.rect(loc1$x, loc1$y, width = loc2$x - loc1$x, height = loc2$y - loc1$y,
          just = c("left", "bottom"), gp = gpar(fill = "red"))
grid.text("group 1", x = (loc1$x + loc2$x)*0.5, y = (loc1$y + loc2$y)*0.5)
```



The viewport names for the annotations are in a fixed format, which is `annotation_{annotation_name}_{slice_index}`. The full set of viewport names can be obtained by `list_components()` function.

```
list_components()
```

```
## [1] "ROOT"                      "global"
## [3] "global_layout"              "global-heatmaplist"
## [5] "main_heatmap_list"          "heatmap_mat2"
## [7] "mat2_heatmap_body_wrap"     "mat2_heatmap_body_1_1"
## [9] "mat2_heatmap_body_1_2"      "mat2_heatmap_body_1_3"
## [11] "mat2_heatmap_body_1_4"      "mat2_heatmap_body_1_5"
## [13] "mat2_dend_row_1"           "mat2_dend_column_1"
## [15] "mat2_dend_column_2"        "mat2_dend_column_3"
## [17] "mat2_dend_column_4"        "mat2_dend_column_5"
## [19] "annotation_empty_1"        "annotation_foo_1"
## [21] "annotation_empty_2"        "annotation_foo_2"
## [23] "annotation_empty_3"        "annotation_foo_3"
## [25] "annotation_empty_4"        "annotation_foo_4"
## [27] "annotation_empty_5"        "annotation_foo_5"
## [29] "global-heatmap_legend_right" "heatmap_legend"
```

If more than one group-level rectangles are to be added, we can wrap the code into a simple function `group_block_anno()`:

```
ha = HeatmapAnnotation(
  empty = anno_empty(border = FALSE, height = unit(8, "mm")),
  foo = anno_block(gp = gpar(fill = 2:6), labels = LETTERS[1:5])
)
Heatmap(mat2, name = "mat2", column_split = split, top_annotation = ha,
        column_title = NULL)

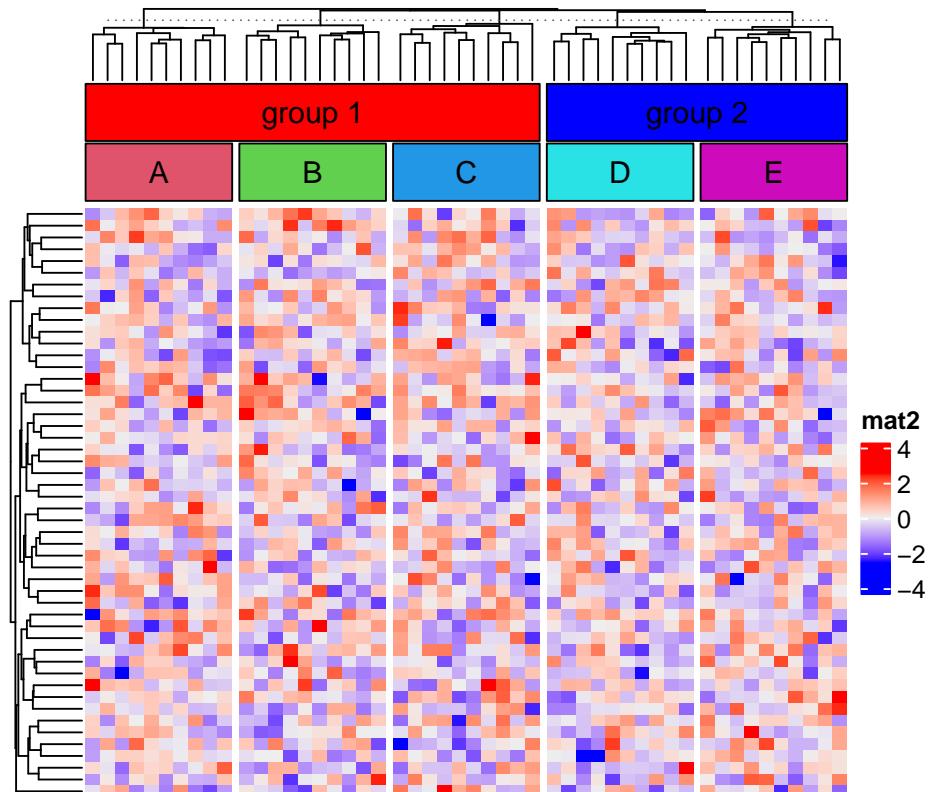
library(GetoptLong) # for the function qq()
group_block_anno = function(group, empty_anno, gp = gpar(),
                            label = NULL, label_gp = gpar()) {

  seekViewport(qq("annotation_{empty_anno}_{min(group)}"))
  loc1 = deviceLoc(x = unit(0, "npc"), y = unit(0, "npc"))
  seekViewport(qq("annotation_{empty_anno}_{max(group)}"))
  loc2 = deviceLoc(x = unit(1, "npc"), y = unit(1, "npc"))

  seekViewport("global")
  grid.rect(loc1$x, loc1$y, width = loc2$x - loc1$x, height = loc2$y - loc1$y,
            just = c("left", "bottom"), gp = gp)
  if(!is.null(label)) {
    grid.text(label, x = (loc1$x + loc2$x)*0.5, y = (loc1$y + loc2$y)*0.5, gp = label_gp)
  }
}
```

```
}
```

```
group_block_anno(1:3, "empty", gp = gpar(fill = "red"), label = "group 1")
group_block_anno(4:5, "empty", gp = gpar(fill = "blue"), label = "group 2")
```



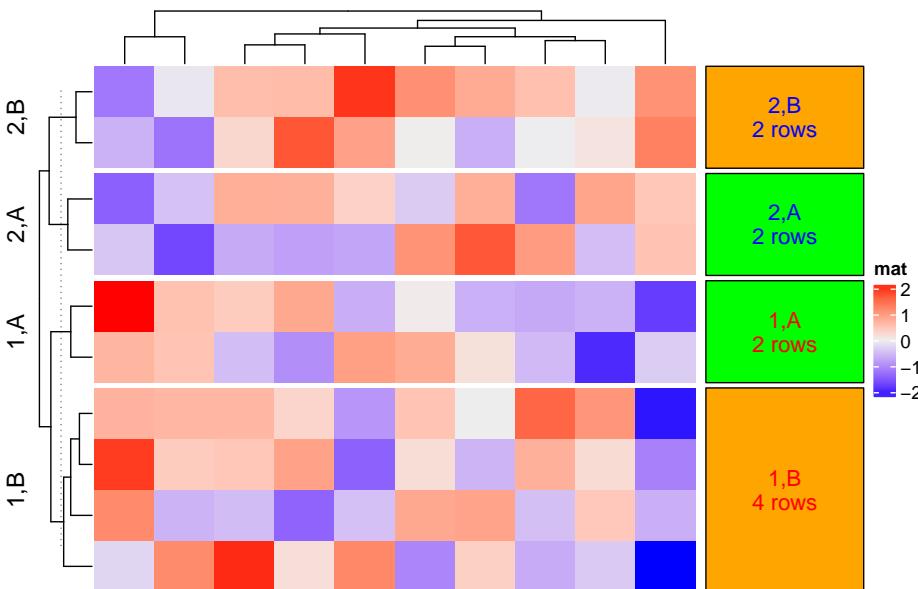
3.4.2 Blocks as plotting regions

When heatmap is split, each block in block annotation can be thought as a virtual plotting region. `anno_block()` allows an argument `panel_fun` which accepts a self-defined function that draws graphics in each slice. It must have two arguments:

1. row/column indices for the current slice (let's call it `index`),
2. a vector of levels from the split variable that correspond to current slice (let's call it `level`). When e.g. `row_km` is only set or `row_split` is only set to one categorical variable, then `level` is a vector of length one. If there are multiple categorical variables set with `row_km` and `row_split`, `level` is a vector of which the length is the same as the number of categorical variables.

When `panel_fun` is set, all other graphics parameters in `anno_block()` are ignored. See the following example:

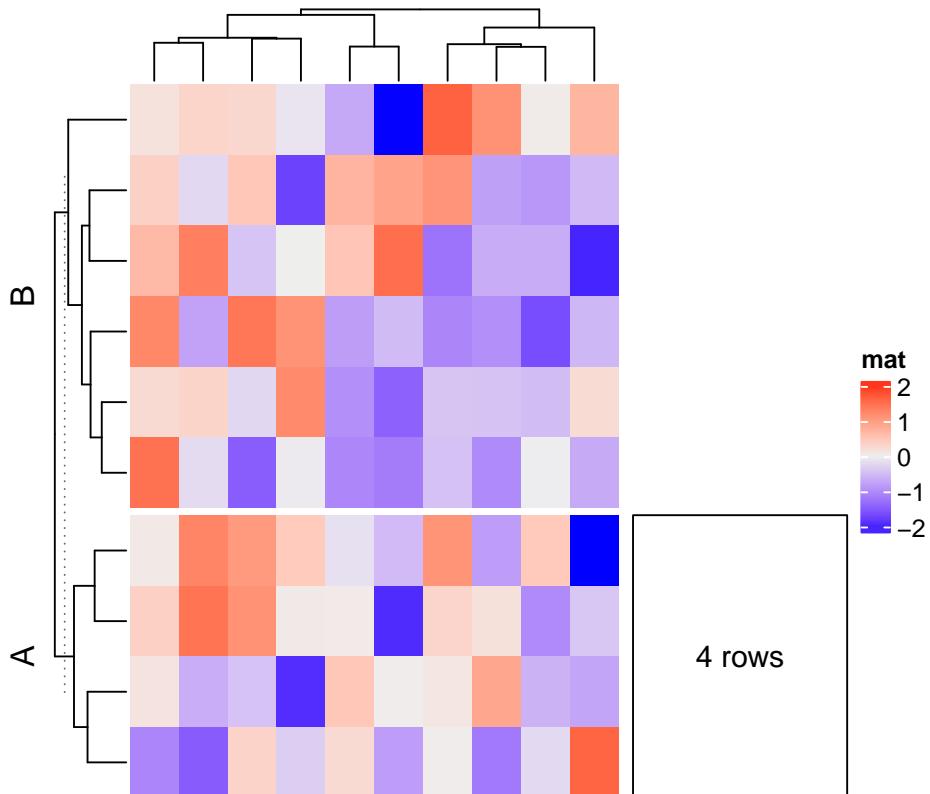
```
col = c("1" = "red", "2" = "blue", "A" = "green", "B" = "orange")
Heatmap(matrix(rnorm(100), 10), name = "mat", row_km = 2,
        row_split = sample(c("A", "B"), 10, replace = TRUE)) +
  rowAnnotation(foo = anno_block(
    panel_fun = function(index, levels) {
      grid.rect(gp = gpar(fill = col[levels[2]], col = "black"))
      txt = paste(levels, collapse = ",")
      txt = paste0(txt, "\n", length(index), " rows")
      grid.text(txt, 0.5, 0.5, rot = 0,
                gp = gpar(col = col[levels[1]]))
    },
    width = unit(3, "cm")
  ))
)
```



To make it more general, `anno_block()` accepts an argument `align_to` which defines a list of indices that blocks will be corresponded to, but you need to make sure the indices are continuously adjacent on heatmaps.

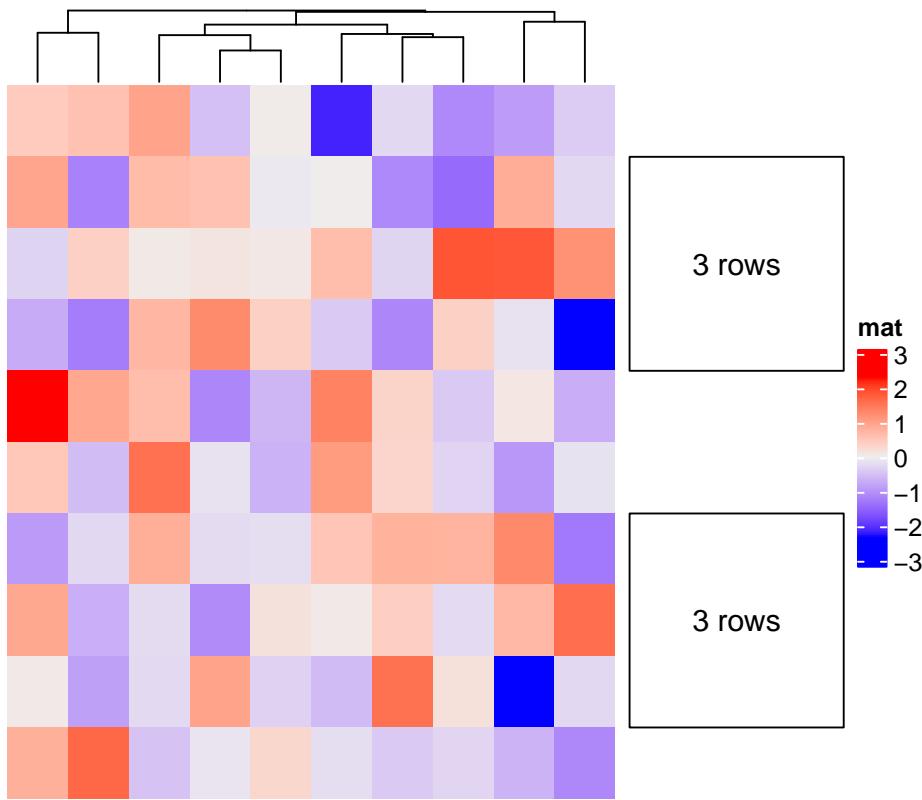
```
split = sample(c("A", "B"), 10, replace = TRUE)
align_to = list("A" = which(split == "A"))
panel_fun = function(index, nm) {
  grid.rect()
  grid.text(paste0(length(index), " rows"), 0.5, 0.5)
}
Heatmap(matrix(rnorm(100), 10), name = "mat", row_split = split) +
```

```
rowAnnotation(foo = anno_block(
  align_to = align_to,
  panel_fun = panel_fun,
  width = unit(3, "cm")
))
```



`anno_block()` normally works with `row_split`, but it is not necessary, see the following example:

```
align_to = list("A" = 2:4, "B" = 7:9)
Heatmap(matrix(rnorm(100), 10), name = "mat", cluster_rows = FALSE) +
rowAnnotation(foo = anno_block(
  align_to = align_to,
  panel_fun = panel_fun,
  width = unit(3, "cm")
))
```



3.5 Image annotation

Images can be added as annotations. `anno_image()` supports image in `png`, `svg`, `pdf`, `eps`, `jpeg/jpg`, `tiff` formats. How they are imported as annotations are as follows:

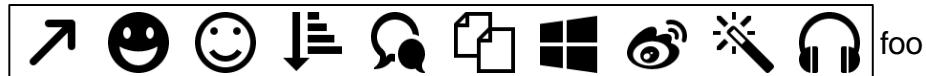
- `png`, `jpeg/jpg` and `tiff` images are imported by `png:::readPNG()`, `jpeg:::readJPEG()` and `tiff:::readTIFF()`, and drawn by `grid:::grid.raster()`.
- `svg` images are firstly reformatted by `rsvg::rsvg_svg()` and then imported by `grImport2:::readPicture()` and drawn by `grImport2:::grid.picture()`.
- `pdf` and `eps` images are imported by `grImport:::PostScriptTrace()` and `grImport:::readPicture()`, later drawn by `grImport:::grid.picture()`.

The free icons for following examples are from <https://github.com/Keyamoon/IcoMoon-Free>. A vector of image paths are set as the first argument of `anno_image()`.

```
image_png = sample(dir("IcoMoon-Free-master/PNG/64px", full.names = TRUE), 10)
image_svg = sample(dir("IcoMoon-Free-master/SVG/", full.names = TRUE), 10)
```

```
image_eps = sample(dir("IcoMoon-Free-master/EPS/", full.names = TRUE), 10)
image_pdf = sample(dir("IcoMoon-Free-master/PDF/", full.names = TRUE), 10)

# we only draw the image annotation for PNG images, while the others are the same
ha = HeatmapAnnotation(foo = anno_image(image_png))
```



Different image formats can be mixed in the input vector.

```
# code only for demonstration
ha = HeatmapAnnotation(foo = anno_image(c(image_png[1:3], image_svg[1:3],
  image_eps[1:3], image_pdf[1:3])))
```

Border and background colors (if the images have transparent background) can be set by `gp`.

```
ha = HeatmapAnnotation(foo = anno_image(image_png,
  gp = gpar(fill = 1:10, col = "black")))
```



`border` controls the border of the whole annotation.

```
ha = HeatmapAnnotation(foo = anno_image(image_png, border = "red"))
```



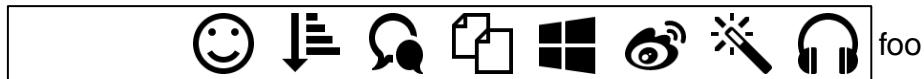
Padding or space around the images is set by `space`.

```
ha = HeatmapAnnotation(foo = anno_image(image_png, space = unit(3, "mm")))
```



If only some of the images need to be drawn, the other elements in the `image` vector can be set to '' or NA.

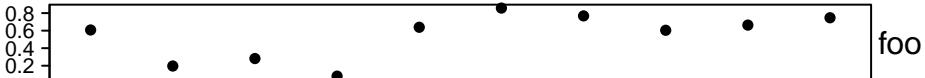
```
image_png[1:2] = ""
ha = HeatmapAnnotation(foo = anno_image(image_png))
```



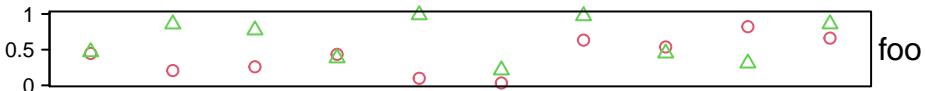
3.6 Points annotation

Points annotation implemented by `anno_points()` shows distribution of a list of data points. The data points object `x` can be a single vector or a matrix. If it is a matrix, the graphics settings such as `pch`, `size` and `gp` can correspond to matrix columns. Note again, if `x` is a matrix, rows in `x` correspond to columns in the heatmap matrix.

```
ha = HeatmapAnnotation(foo = anno_points(runif(10)))
```



```
ha = HeatmapAnnotation(foo = anno_points(matrix(runif(20), nc = 2),
  pch = 1:2, gp = gpar(col = 2:3)))
```



`ylim` controls the range on “y-axis” or the “data axis” (if it is a row annotation, the data axis is horizontal), `extend` controls the extended space on the data axis direction. `axis` controls whether to show the axis and `axis_param` controls the settings for axis. The default settings for axis are:

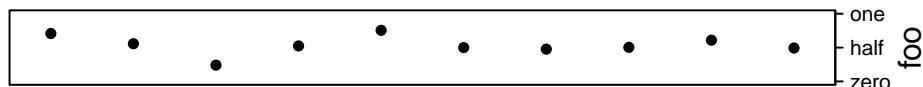
```
default_axis_param("column")
```

```
## $at
## NULL
##
## $labels
## NULL
##
## $labels_rot
## [1] 0
##
## $gp
## $fontsize
## [1] 8
##
##
```

```
## $side
## [1] "left"
##
## $facing
## [1] "outside"
##
## $direction
## [1] "normal"
```

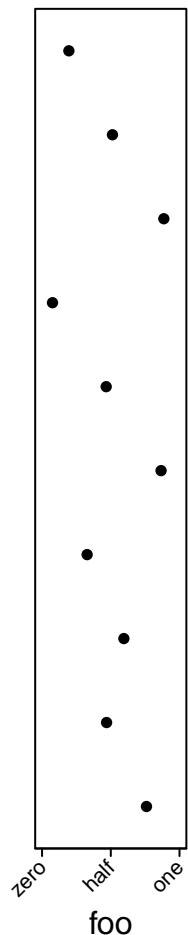
And you can overwrite some of them:

```
ha = HeatmapAnnotation(foo = anno_points(runif(10), ylim = c(0, 1),
  axis_param = list(
    side = "right",
    at = c(0, 0.5, 1),
    labels = c("zero", "half", "one")
  )))
)
```



One thing that might be useful is you can control the rotation of the axis labels.

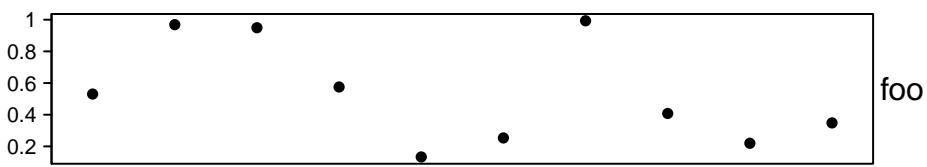
```
ha = rowAnnotation(foo = anno_points(runif(10), ylim = c(0, 1),
  width = unit(2, "cm"),
  axis_param = list(
    side = "bottom",
    at = c(0, 0.5, 1),
    labels = c("zero", "half", "one"),
    labels_rot = 45
  )))
)
```



The configuration of axis is same for all other annotation functions which have axes.

The default size of the points annotation is 5mm. It can be controlled by height/width argument in anno_points().

```
ha = HeatmapAnnotation(foo = anno_points(runif(10), height = unit(2, "cm")))
```



3.7 Line annotation

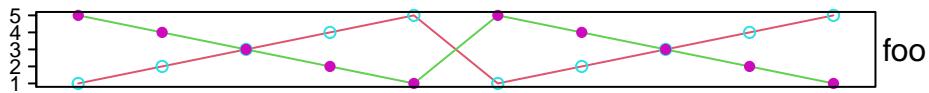
`anno_lines()` connects the data points by a list of segments. Similar as `anno_points()`, the data variable can be a numeric vector:

```
ha = HeatmapAnnotation(foo = anno_lines(runif(10)))
```



Or a matrix:

```
ha = HeatmapAnnotation(foo = anno_lines(cbind(c(1:5, 1:5), c(5:1, 5:1)),
                                         gp = gpar(col = 2:3), add_points = TRUE, pt_gp = gpar(col = 5:6), pch = c(1, 16)))
```



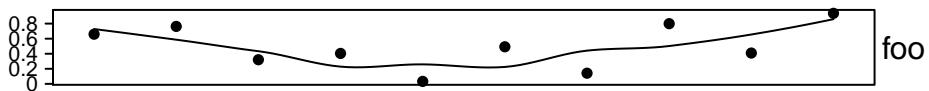
As shown above, points can be added to the lines by setting `add_points = TRUE`.

Smoothed lines (by `loess()`) can be added instead of the original lines by setting `smooth = TRUE`, but it should be used with caution because the order of columns in the heatmap is used as “x-value” for the fitting and only if you think the fitting against the reordered order makes sense.

Smoothing also works when the input data variable is a matrix that the smoothing is performed for each column separately.

If `smooth` is `TRUE`, `add_points` is set to `TRUE` by default.

```
ha = HeatmapAnnotation(foo = anno_lines(runif(10), smooth = TRUE))
```



The default size of the lines annotation is 5mm. It can be controlled by `height/width` argument in `anno_lines()`.

```
# code only for demonstration
ha = HeatmapAnnotation(foo = anno_lines(runif(10), height = unit(2, "cm")))
```

3.8 Barplot annotation

The data points can be represented as barplots. Some of the arguments in `anno_barplot()` such as `ylim`, `axis`, `axis_param` are the same as `anno_points()`.

```
ha = HeatmapAnnotation(foo = anno_barplot(1:10))
```



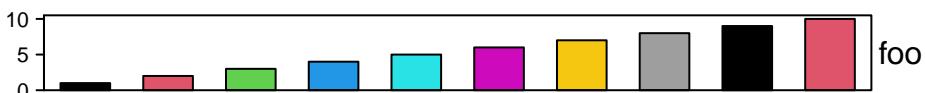
The width of bars is controlled by `bar_width`. It is a relative value to the width of the cell in the heatmap.

```
ha = HeatmapAnnotation(foo = anno_barplot(1:10, bar_width = 1))
```



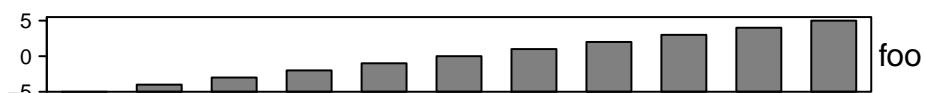
Graphic parameters are controlled by `gp`.

```
ha = HeatmapAnnotation(foo = anno_barplot(1:10, gp = gpar(fill = 1:10)))
```

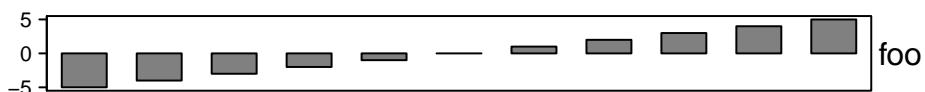


You can choose the baseline of bars by `baseline`.

```
ha = HeatmapAnnotation(foo = anno_barplot(seq(-5, 5), baseline = "min"))
```

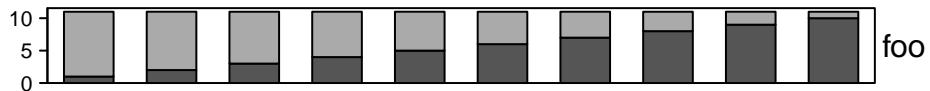


```
ha = HeatmapAnnotation(foo = anno_barplot(seq(-5, 5), baseline = 0))
```



If the input value is a matrix, it will be stacked barplots.

```
ha = HeatmapAnnotation(foo = anno_barplot(matrix(nc = 2, c(1:10, 10:1))))
```



And length of parameters in `gp` can be the number of the columns in the matrix:

```
ha = HeatmapAnnotation(foo = anno_barplot(cbind(1:10, 10:1),
                                           gp = gpar(fill = 2:3, col = 2:3)))
```

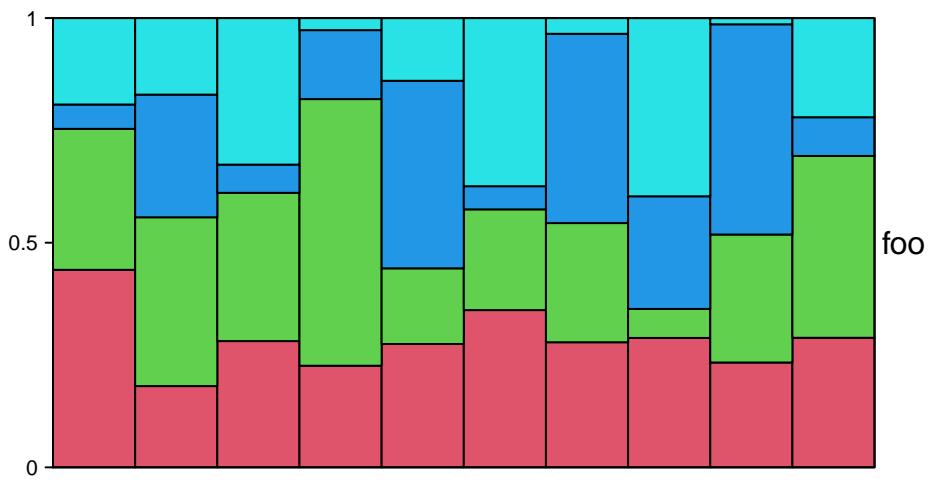


The default size of the barplot annotation is 5mm. It can be controlled by `height/width` argument in `anno_barplot()`.

```
# code only for demonstration
ha = HeatmapAnnotation(foo = anno_barplot(runif(10), height = unit(2, "cm")))
```

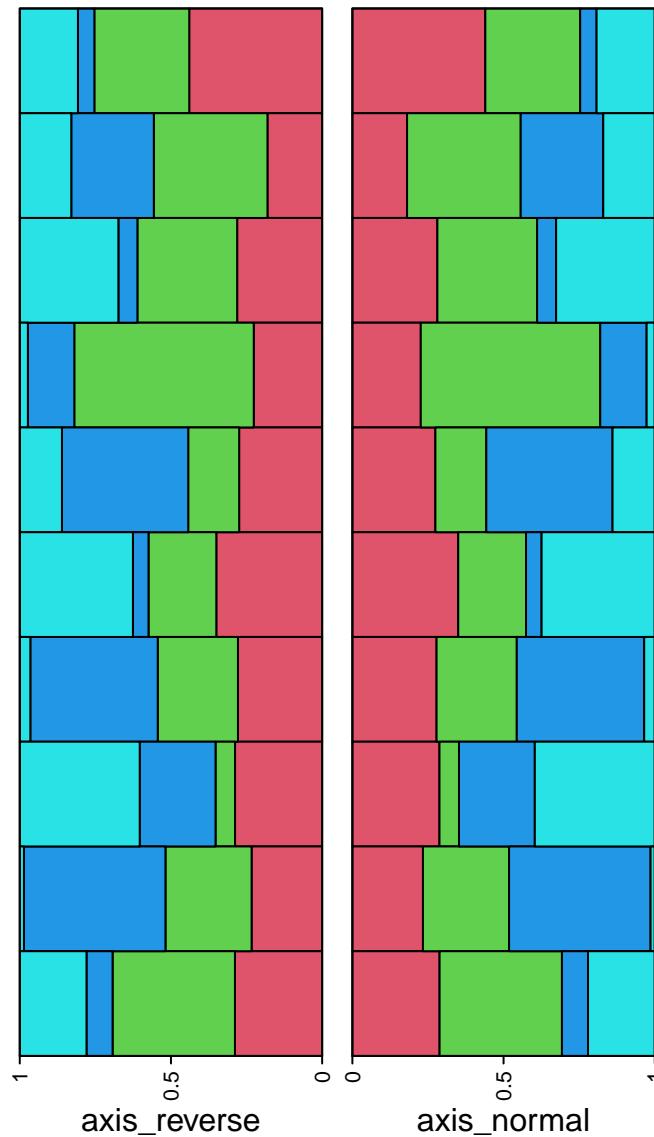
Following example shows a barplot annotation which visualizes a proportion matrix (for which row sums are 1).

```
m = matrix(runif(4*10), nc = 4)
m = t(apply(m, 1, function(x) x/sum(x)))
ha = HeatmapAnnotation(foo = anno_barplot(m, gp = gpar(fill = 2:5),
                                         bar_width = 1, height = unit(6, "cm")))
```



The direction of the axis can be reversed which is useful when the annotation is put on the left of the heatmap.

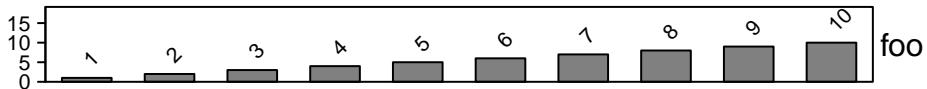
```
ha_list = rowAnnotation(axis_reverse = anno_barplot(m, gp = gpar(fill = 2:5),
  axis_param = list(direction = "reverse"),
  bar_width = 1, width = unit(4, "cm")))+
rowAnnotation(axis_normal = anno_barplot(m, gp = gpar(fill = 2:5),
  bar_width = 1, width = unit(4, "cm")))
draw(ha_list, ht_gap = unit(4, "mm"))
```



`direction = "reverse"` also works for other annotation functions which have axes, but it is more commonly used for barplot annotations.

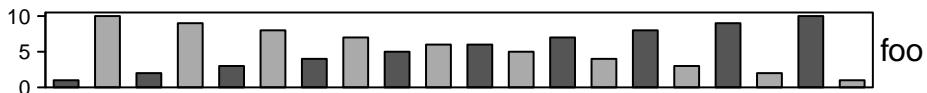
Argument `add_numbers` can be set to TRUE so that numbers associated to bars are drawn on top of the bars. For column annotation, texts are by default with 45 degree rotation.

```
ha = HeatmapAnnotation(foo = anno_barplot(1:10, add_numbers = TRUE,
                                          height = unit(1, "cm")))
```



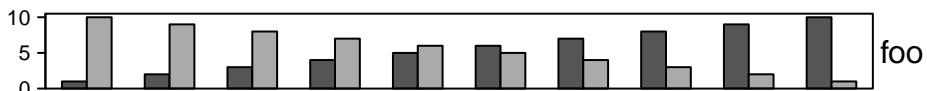
When the input for `anno_barplot()` is a matrix, argument `beside` can be set to TRUE so that bars for a same heatmap column are positioned by each other:

```
ha = HeatmapAnnotation(foo = anno_barplot(matrix(nc = 2, c(1:10, 10:1)),
                                           beside = TRUE))
```



Argument `attach` can be set to TRUE so that two adjacent bars are attached.

```
ha = HeatmapAnnotation(foo = anno_barplot(matrix(nc = 2, c(1:10, 10:1)),
                                           beside = TRUE, attach = TRUE))
```

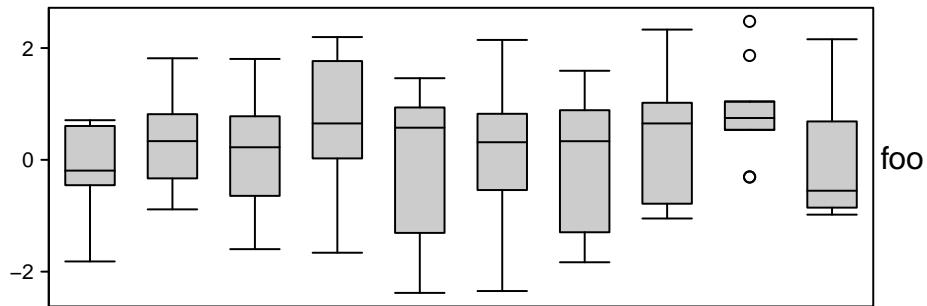


3.9 Boxplot annotation

Boxplot annotation as well as the annotation functions which are introduced later are more suitable for small matrices. I don't think you want to put boxplots as column annotation for a matrix with 100 columns.

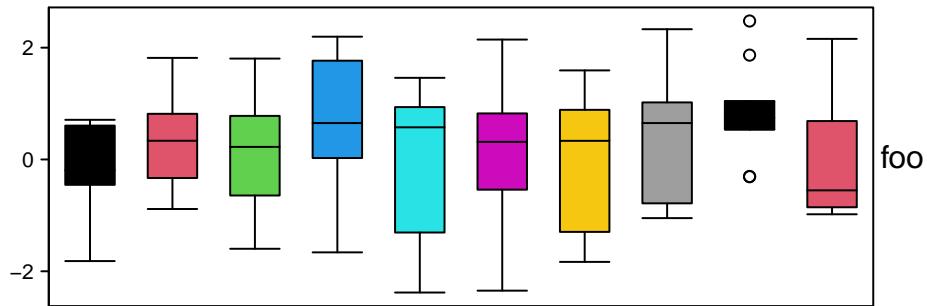
For `anno_boxplot()`, the input data variable should be a matrix or a list. If `x` is a matrix and if it is a column annotation, statistics for boxplots are calculated by columns, and if it is a row annotation, the calculation is done by rows.

```
set.seed(12345)
m = matrix(rnorm(100), 10)
ha = HeatmapAnnotation(foo = anno_boxplot(m, height = unit(4, "cm")))
```



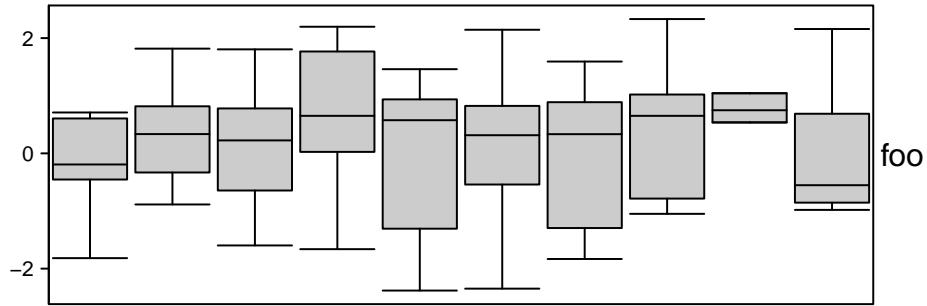
Graphic parameters are controlled by `gp`.

```
ha = HeatmapAnnotation(foo = anno_boxplot(m, height = unit(4, "cm"),
    gp = gpar(fill = 1:10)))
```



Width of the boxes are controlled by `box_width`. `outline` controls whether to show outlier points.

```
ha = HeatmapAnnotation(foo = anno_boxplot(m, height = unit(4, "cm"),
    box_width = 0.9, outline = FALSE))
```



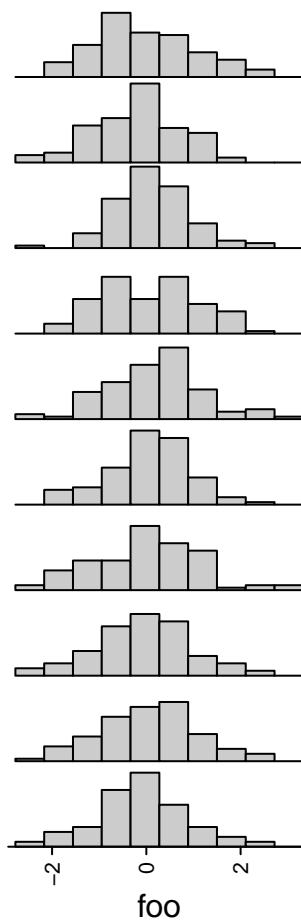
`anno_boxplot()` only draws one boxplot for one single row. Section 14.6 demonstrates how to define an annotation function which draws multiple boxplots for a single row, and Section 3.18 demonstrates how to draw one single boxplot for a group of rows.

3.10 Histogram annotation

Annotations as histograms are more suitable to put as row annotations. The setting for the data variable is the same as `anno_boxplot()` which can be a matrix or a list.

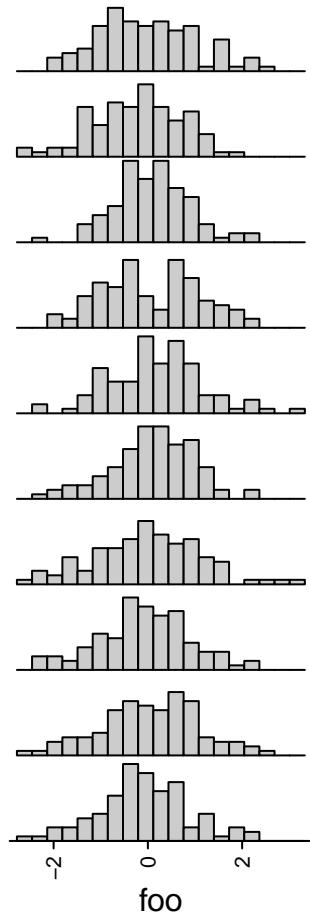
Similar as `anno_boxplot()`, the input data variable should be a matrix or a list. If `x` is a matrix and if it is a column annotation, histograms are calculated by columns, and if it is a row annotation, histograms are calculated by rows.

```
m = matrix(rnorm(1000), nc = 100)
ha = rowAnnotation(foo = anno_histogram(m)) # apply `m` on rows
```



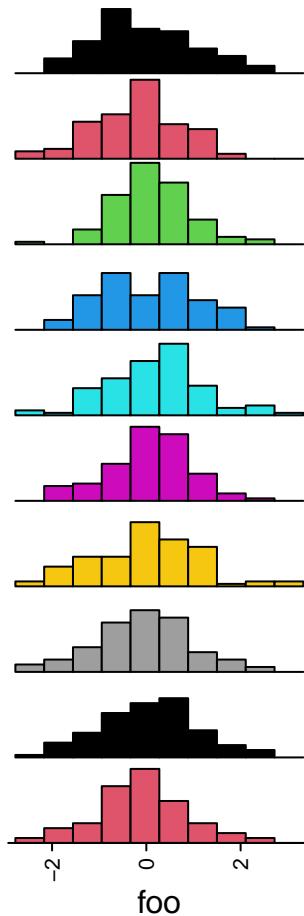
Number of breaks for histograms is controlled by `n_breaks`.

```
ha = rowAnnotation(foo = anno_histogram(m, n_breaks = 20))
```



Colors are controlled by `gp`.

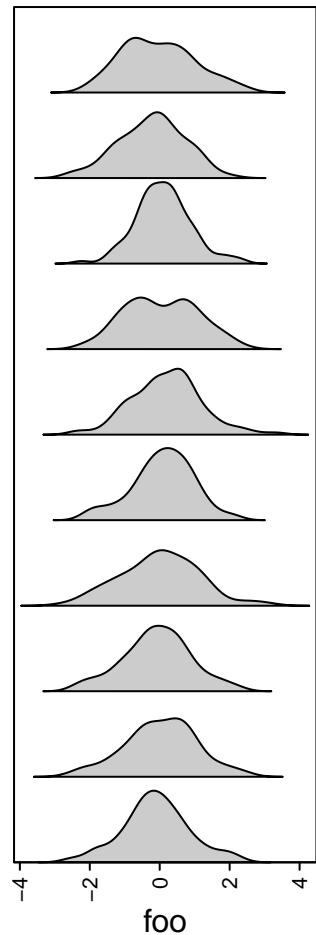
```
ha = rowAnnotation(foo = anno_histogram(m, gp = gpar(fill = 1:10)))
```



3.11 Density annotation

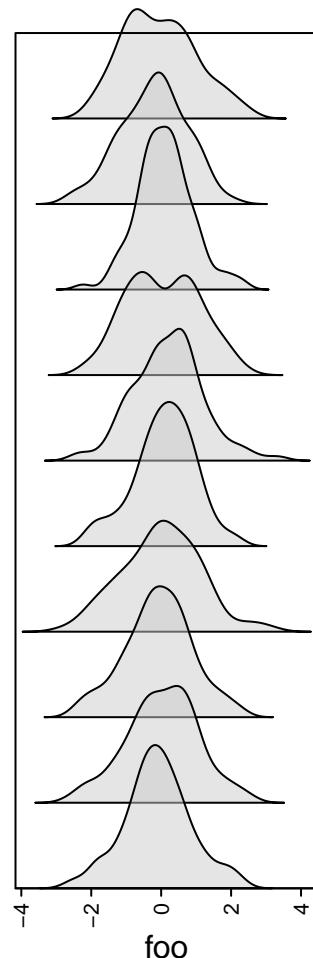
Similar as histogram annotations, `anno_density()` shows the distribution as a fitted curve.

```
ha = rowAnnotation(foo = anno_density(m))
```



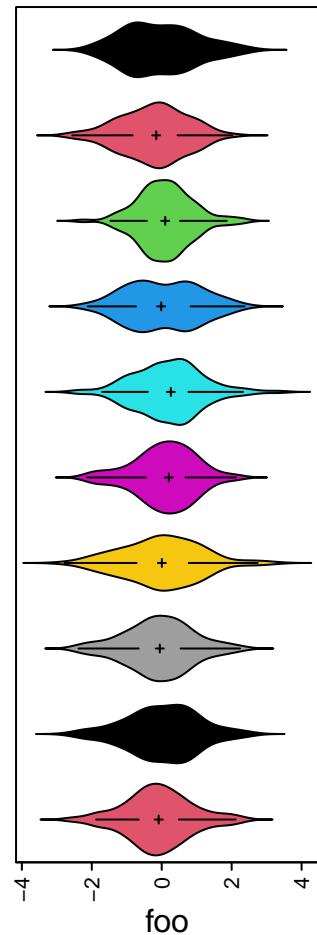
The height of the density peaks can be controlled to make the distribution look like a “joyplot”.

```
ha = rowAnnotation(foo = anno_density(m, joyplot_scale = 2,  
gp = gpar(fill = "#CCCCCC80")))
```



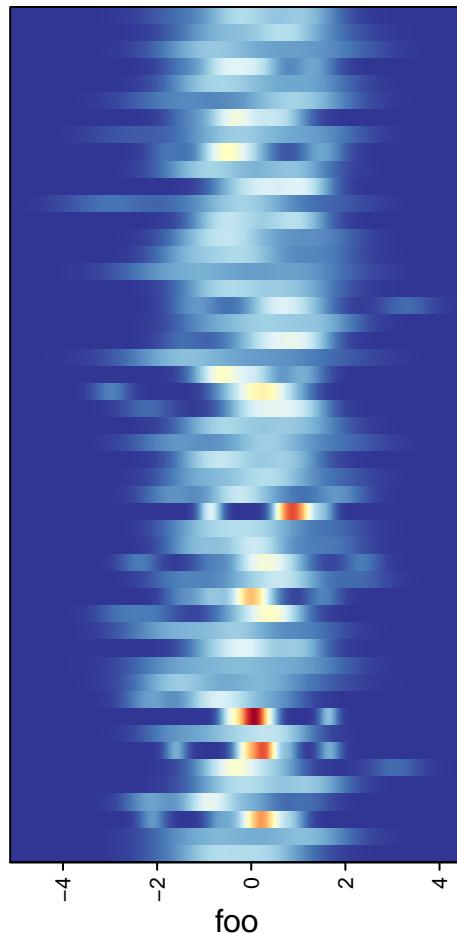
Or visualize the distribution as violin plot.

```
ha = rowAnnotation(foo = anno_density(m, type = "violin",
  gp = gpar(fill = 1:10)))
```



When there are too many rows in the input variable, the space for normal density peaks might be too small. In this case, we can visualize the distribution by heatmaps.

```
m2 = matrix(rnorm(50*10), nrow = 50)
ha = rowAnnotation(foo = anno_density(m2, type = "heatmap", width = unit(6, "cm")))
```



The color schema for heatmap distribution is controlled by `heatmap_colors`.

```
ha = rowAnnotation(foo = anno_density(m2, type = "heatmap", width = unit(6, "cm"),
  heatmap_colors = c("white", "orange")))
```



In **ComplexHeatmap** package, there is a `densityHeatmap()` function which visualizes distribution as a heatmap. It will be introduced in Section 11.1.

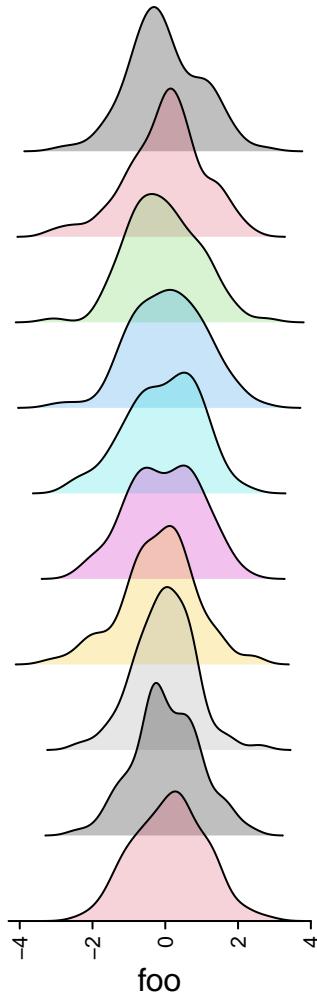
3.12 Joyplot annotation

`anno_joyplot()` is specific for so-called joyplot (<http://blog.revolutionanalytics.com/2017/07/joyplots.html>). The input data should be a matrix or a list.

Note `anno_joyplot()` is always applied to columns if the input is a matrix. Because joyplot visualizes parallel distributions and the matrix is not a necessary format while a list is already enough for it, if you are not sure about how to set as a matrix, just convert it to a list for using it.

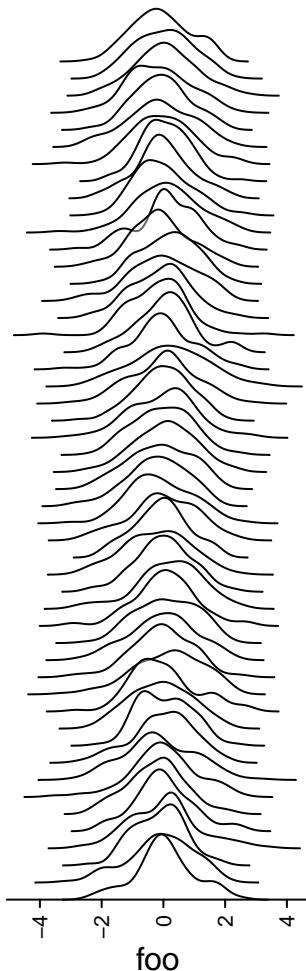
```
m = matrix(rnorm(1000), nc = 10)
lt = apply(m, 2, function(x) data.frame(density(x)[c("x", "y")]))
ha = rowAnnotation(foo = anno_joyplot(lt, width = unit(4, "cm")),
```

```
gp = gpar(fill = 1:10), transparency = 0.75))
```



Or only show the lines (`scale` argument controls the relative height of the curves).

```
m = matrix(rnorm(5000), nc = 50)
lt = apply(m, 2, function(x) data.frame(density(x)[c("x", "y")]))
ha = rowAnnotation(foo = anno_joyplot(lt, width = unit(4, "cm"), gp = gpar(fill = NA),
scale = 4))
```



The format of the input variable is special. It can be one of the following two:

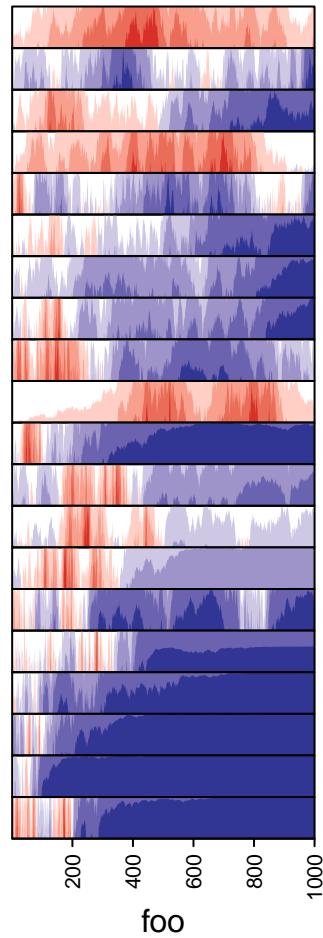
1. a matrix (remember `anno_joyplot()` is always applied to columns of the matrix) where x coordinate corresponds to `1:nrow(matrix)` and each column in the matrix corresponds to one distribution in the joyplot.
2. a list of data frames where each data frame has two columns which correspond to x coordinate and y coordinate.

3.13 Horizon chart annotation

Horizon chart as annotation can only be added as row annotation. The format of the input variable for `anno_horizon()` is the same as `anno_joyplot()` which is introduced in previous section.

The default style of horizon chart annotation is:

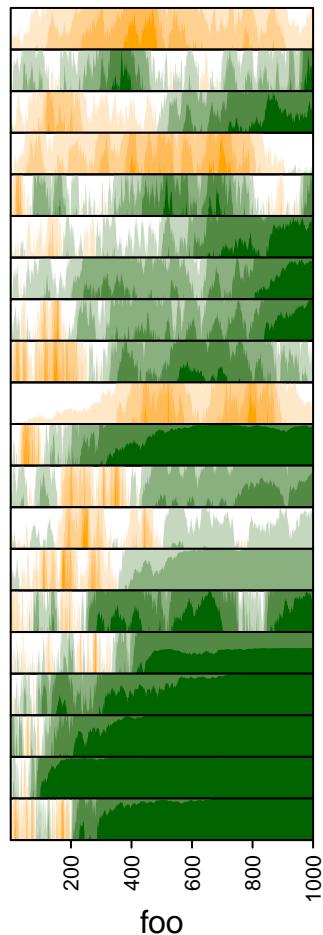
```
lt = lapply(1:20, function(x) cumprod(1 + runif(1000, -x/100, x/100)) - 1)
ha = rowAnnotation(foo = anno_horizon(lt))
```



Values in each track are normalized by $x/\max(\text{abs}(x))$.

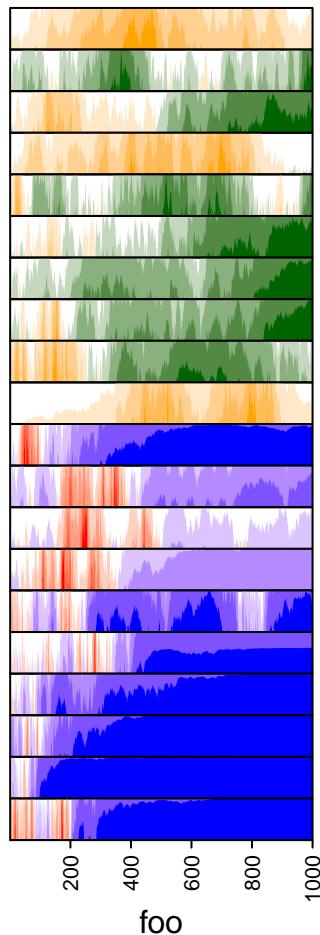
Colors for positive values and negative values are controlled by `pos_fill` and `neg_fill` in `gpar()`.

```
ha = rowAnnotation(foo = anno_horizon(lt,
gp = gpar(pos_fill = "orange", neg_fill = "darkgreen")))
```



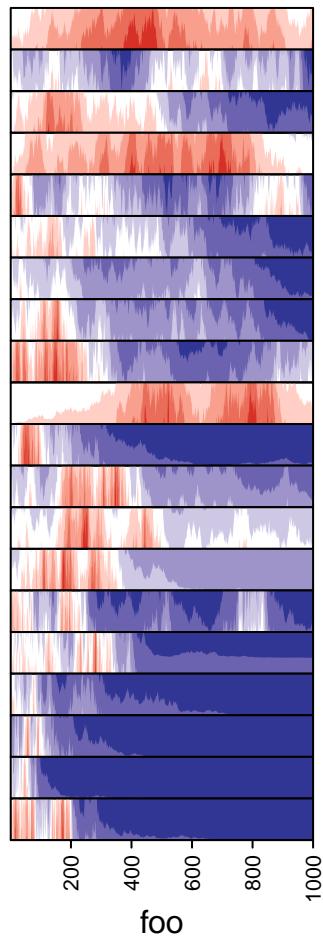
`pos_fill` and `neg_fill` can be assigned as a vector.

```
ha = rowAnnotation(foo = anno_horizon(lt,
  gp = gpar(pos_fill = rep(c("orange", "red"), each = 10),
  neg_fill = rep(c("darkgreen", "blue"), each = 10))))
```



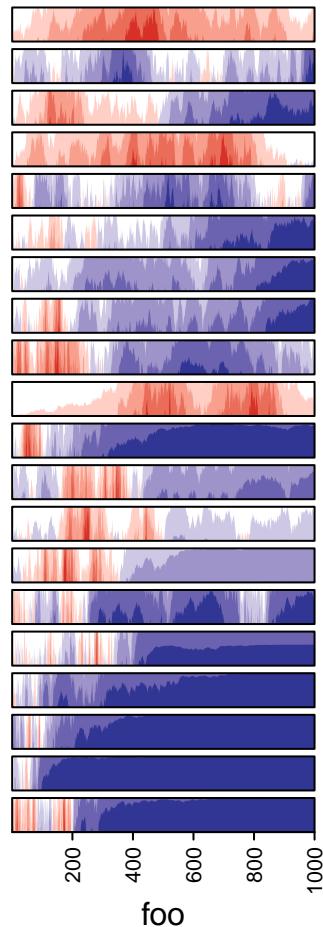
Whether the peaks for negative values start from the bottom or from the top.

```
ha = rowAnnotation(foo = anno_horizon(lt, negative_from_top = TRUE))
```



The space between every two neighbouring charts.

```
ha = rowAnnotation(foo = anno_horizon(lt, gap = unit(1, "mm")))
```



3.14 Text annotation

Text can be used as annotations by `anno_text()`. Graphics parameters are controlled by `gp`.

```
ha = rowAnnotation(foo = anno_text(month.name, gp = gpar(fontsize = 1:12+4)))
```

January

February

March

April

May

June

July

August

September

October

November

December

Locationsn are controlled by `location` and `just`. Rotation is controlled by `rot`.

```
ha = rowAnnotation(foo = anno_text(month.name, location = 1, rot = 30,
  just = "right", gp = gpar(fontsize = 1:12+4)))
```

January
February
March
April
May
June
July
August
September
October
November
December

```
ha = rowAnnotation(foo = anno_text(month.name, location = 0.5, just = "center"))
```

January

February

March

April

May

June

July

August

September

October

November

December

`location` and `just` are automatically calculated according the the position of the annotations put to the heatmap (e.g. text are aligned to the left if it is a right annotation to the heatmap and are aligned to the right if it is a left annotation).

The width/height are automatically calculated based on all the text. Normally you don't need to manually set the width/height of it.

Background colors can be set by `gp`. Here `fill` controls the filled background color, `col` controls the color of text and the non-standard `border` controls the background border color.

You can see we explicitly set `width` as 1.2 times the width of the longest text.

```
ha = rowAnnotation(foo = anno_text(month.name, location = 0.5, just = "center",
  gp = gpar(fill = rep(2:4, each = 4), col = "white", border = "black"),
  width = max_text_width(month.name)*1.2))
```



More complicated texts can be drawn by integrating with the `gridtext` package (Section 10.3.3). A quick example is as follows:

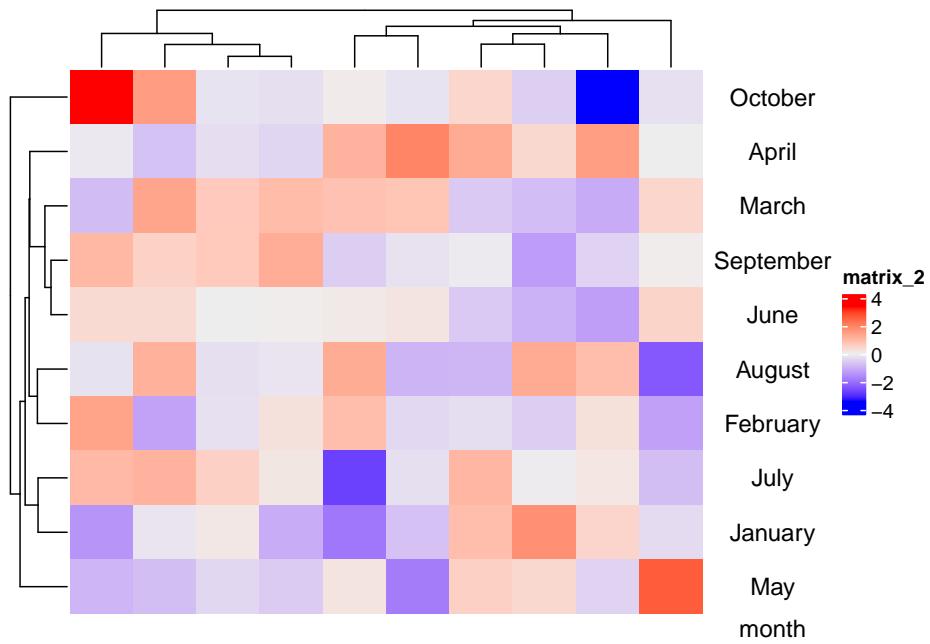
```
text = sapply(LETTERS[1:10], function(x) {
  qq("<span style='color:red'>**@{x}**<sub>@{x}</sub></span>_@{x}_<sup>@{x}</sup>")
})
ha = rowAnnotation(
  foo = anno_text(gt_render(text, align_widths = TRUE,
                            r = unit(2, "pt"),
                            padding = unit(c(2, 2, 2, 2), "pt")),
                  gp = gpar(box_col = "blue", box_lwd = 2),
                  just = "right",
                  location = unit(1, "npc"))
)

```



Unlike other annotations, by default there is no annotation title for text annotation. The title can be added by setting `show_name = TRUE` in `anno_text()`:

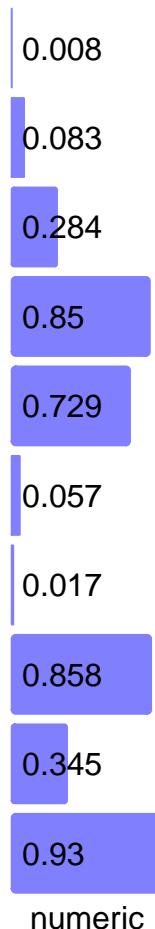
```
m = matrix(rnorm(100), 10)
Heatmap(m) + rowAnnotation(month = anno_text(month.name[1:10], just = "center",
                                             location = unit(0.5, "npc"), show_name = TRUE),
                            annotation_name_rot = 0)
```



3.15 Numeric labels annotation

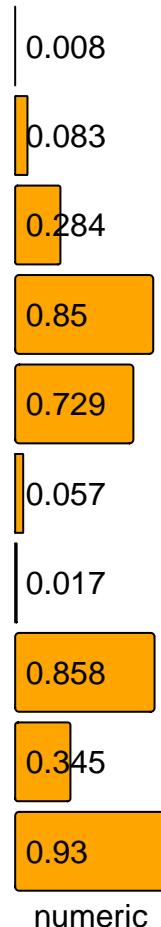
There is a special text annotation for numeric labels. To emphasize the visual effect, we also want to add bars to show the absolute values of the numbers. This can be done with `anno_numeric()` function. Note currently it only supports row annotation.

```
x = round(runif(10), 3)
ha = rowAnnotation(numeric = anno_numeric(x), annotation_name_rot = 0)
```



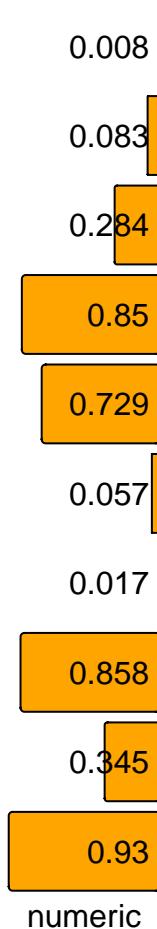
Background colors can be controlled by `bg_gp`.

```
ha = rowAnnotation(numeric = anno_numeric(x,
    bg_gp = gpar(fill = "orange", col = "black")),
annotation_name_rot = 0)
```



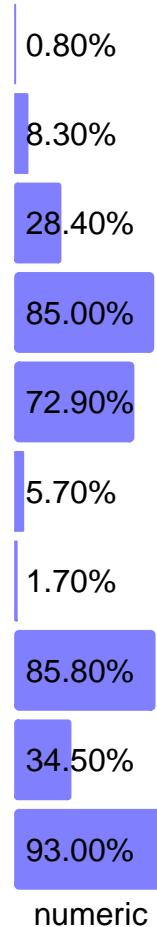
align_to can be set to "right" or "left":

```
ha = rowAnnotation(numeric = anno_numeric(x,
    bg_gp = gpar(fill = "orange", col = "black"),
    align_to = "right"),
    annotation_name_rot = 0)
```



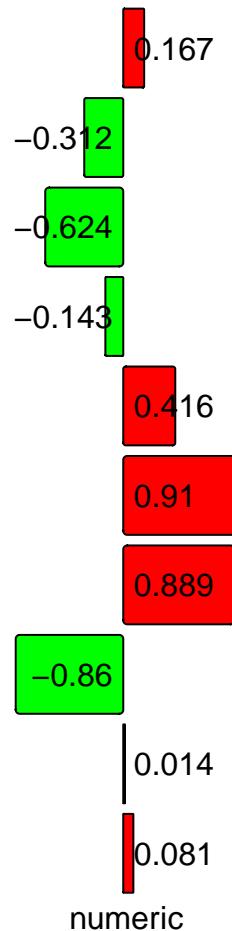
Format of labels on the plot can be controlled by `labels_format`:

```
ha = rowAnnotation(numeric = anno_numeric(x,
    labels_format = function(x) paste0(sprintf("%.2f", x*100), "%")),
    annotation_name_rot = 0)
```



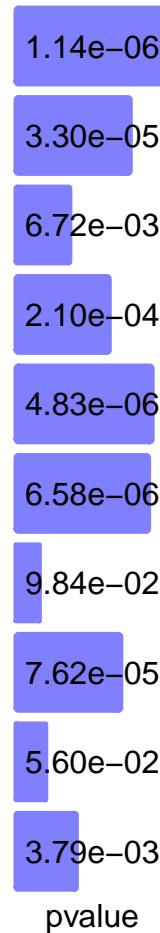
It is also possible to show a numeric vector with both negative and positive values. In this case, all graphics parameters should have length of two where the first one controls negative values and the second controls positive values.

```
x = round(runif(10, -1, 1), 3)
ha = rowAnnotation(numeric = anno_numeric(x,
    bg_gp = gpar(fill = c("green", "red"))),
    annotation_name_rot = 0)
```



Another example is to visualize a vector of p-values. Here we should apply `-log10()` on p-values for mapping to bar heights with `x_convert` argument:

```
x = 10^(-runif(10, 1, 6))
ha = rowAnnotation(pvalue = anno_numeric(x,
    rg = c(min(x), 1),
    x_convert = function(x) -log10(x),
    labels_format = function(x) sprintf("%.2e", x)),
    annotation_name_rot = 0)
```



3.16 Completely customized annotation

For most annotation functions implemented in **ComplexHeatmap**, they only draw one same type of annotation graphics, e.g. `anno_points()` only draws points or `anno_image()` only draws images. From **ComplexHeatmap** version 2.9.4, we added a new annotation function `anno_customize()`, with which you can completely freely define graphics for every annotation cell.

The input for `anno_customize()` should be a categorical vector.

```
x = c("a", "a", "a", "b", "b", "b", "c", "c", "d", "d")
```

For each level, you need to define a graphics function for it.

```
graphics = list(
  "a" = function(x, y, w, h) {
```

```

        grid.rect(x, y, w*0.8, h*0.33, gp = gpar(fill = "red"))
    },
    "b" = function(x, y, w, h) {
        grid.text("A", x, y, gp = gpar(col = "darkgreen"))
    },
    "c" = function(x, y, w, h) {
        grid.points(x, y, gp = gpar(col = "orange"), pch = 16)
    },
    "d" = function(x, y, w, h) {
        img = png::readPNG(system.file("extdata", "Rlogo.png", package = "circlize"))
        grid.raster(img, x, y, width = unit(0.8, "snpc"),
                    height = unit(0.8, "snpc")*nrow(img)/ncol(img))
    }
)

```

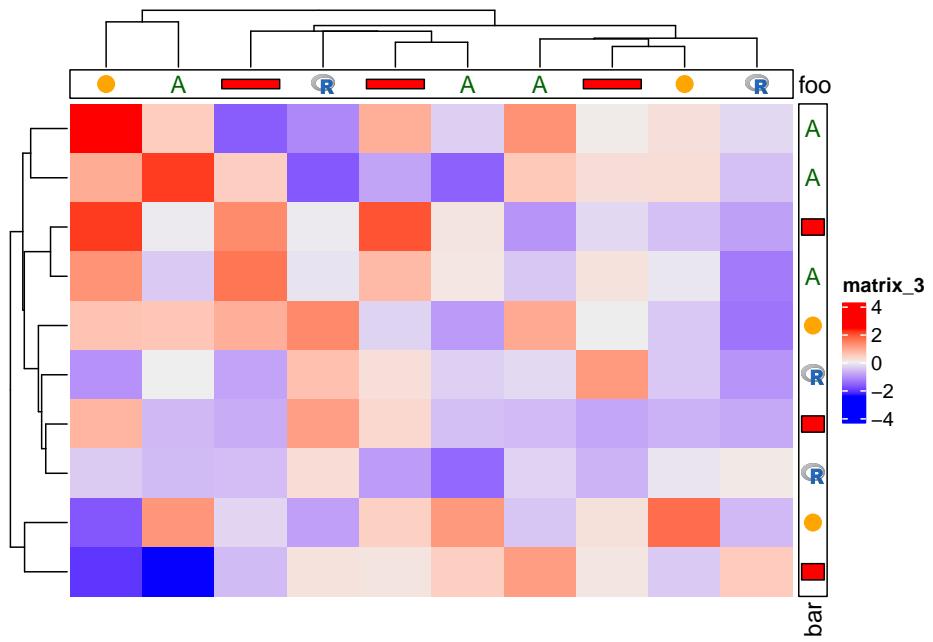
When adding graphics, each annotation cell is an independent viewport, thus, the self-defined graphics function accepts four arguments: `x` and `y`: the center of the viewport of the annotation cell, `w` and `h`: the width and height of the viewport. In the example above, we set a horizontal bar for "`a`", a text for "`b`", a point for "`c`" and an image for "`d`".

Then we add this new annotation to both row and column of the heatmap, just in the same way as normal annotations.

```

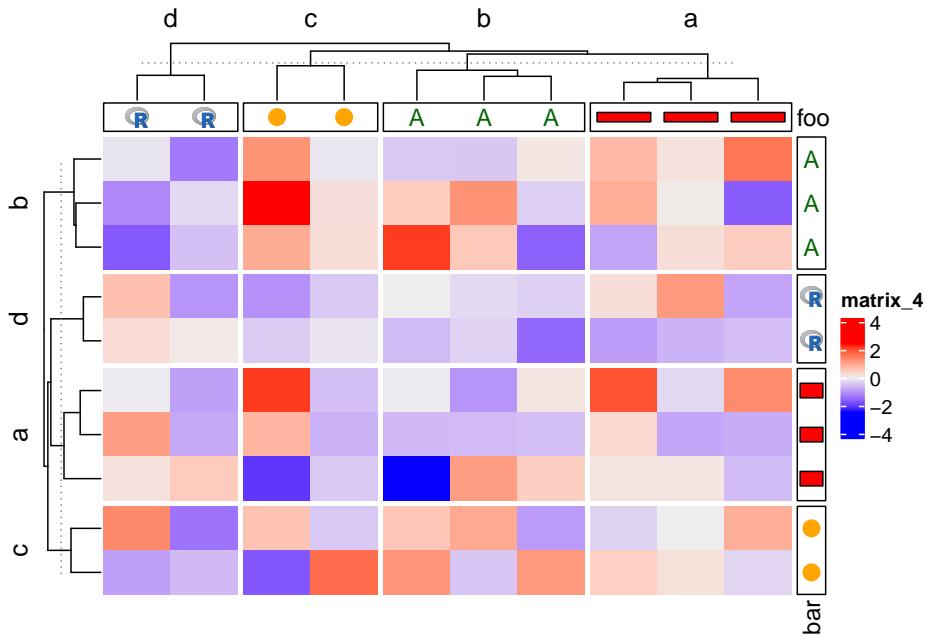
m = matrix(rnorm(100), 10)
Heatmap(m,
        top_annotation = HeatmapAnnotation(foo = anno_customize(x, graphics = graphics)),
        right_annotation = rowAnnotation(bar = anno_customize(x, graphics)))

```



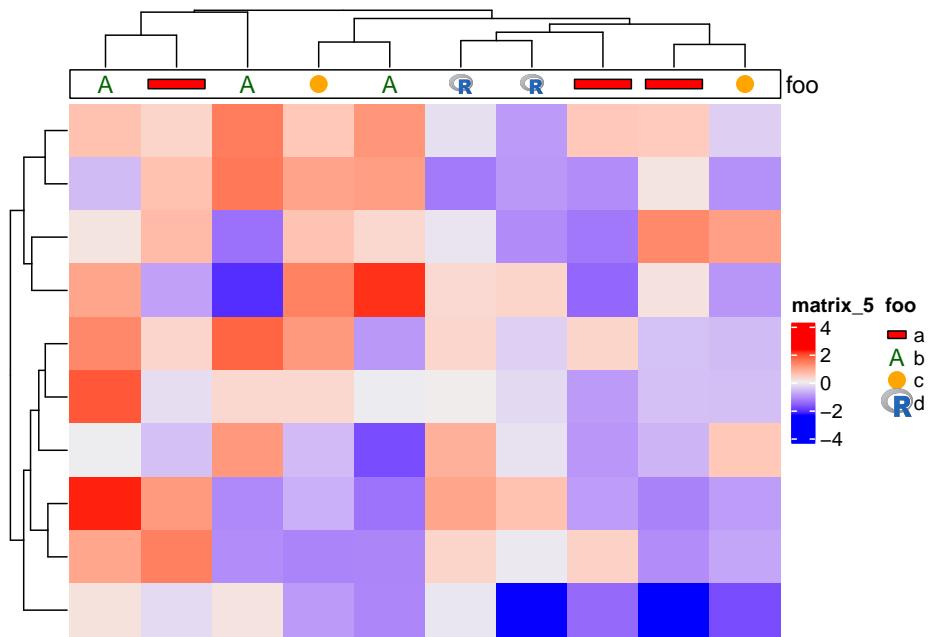
Reordering and splitting are automatically adjusted.

```
Heatmap(m,
        top_annotation = HeatmapAnnotation(foo = anno_customize(x, graphics = graphics)),
        right_annotation = rowAnnotation(bar = anno_customize(x, graphics = graphics)),
        column_split = x, row_split = x)
```



`Legend()` function also accepts a `graphics` argument, so it is easy to add legends for the “customized annotations.” How to use `Legend()` function will be introduced in Chapter 5.

```
m = matrix(rnorm(100), 10)
ht = Heatmap(m,
             top_annotation = HeatmapAnnotation(foo = anno_customise(x, graphics = graphics)))
lgd = Legend(title = "foo", at = names(graphics), graphics = graphics)
draw(ht, annotation_legend_list = lgd)
```

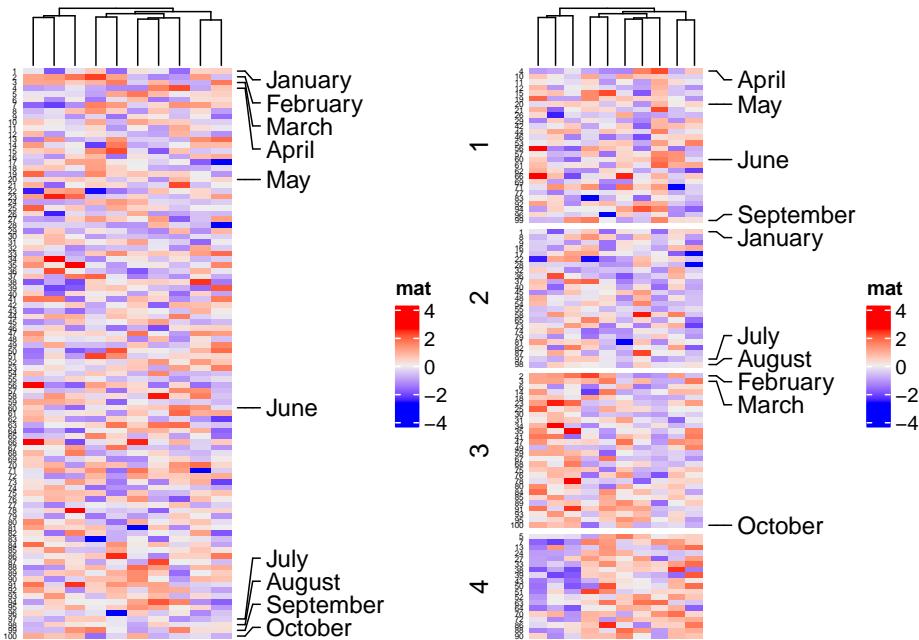


3.17 Mark annotation

Sometimes there are many rows or columns in the heatmap and we want to mark some of them. `anno_mark()` is used to mark subset of rows or columns and connect to labels with lines.

`anno_mark()` at least needs two arguments where `at` are the indices to the original matrix and `labels` are the corresponding text.

```
m = matrix(rnorm(1000), nrow = 100)
rownames(m) = 1:100
ha = rowAnnotation(foo = anno_mark(at = c(1:4, 20, 60, 97:100),
                                    labels = month.name[1:10]))
Heatmap(m, name = "mat", cluster_rows = FALSE, right_annotation = ha,
        row_names_side = "left", row_names_gp = gpar(fontsize = 4))
Heatmap(m, name = "mat", cluster_rows = FALSE, right_annotation = ha,
        row_names_side = "left", row_names_gp = gpar(fontsize = 4), row_km = 4)
```



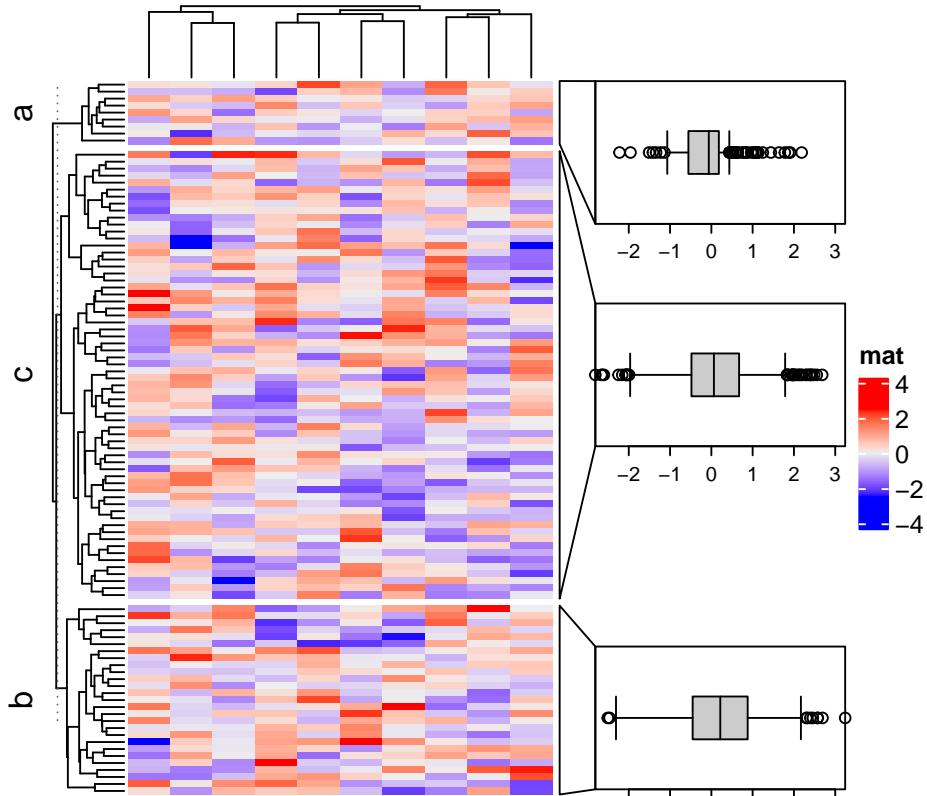
The calculation of positions of texts depends on the absolute size of the graphics device. If you resize the current interactive device or you use `grid.grabExpr()` to capture the current plot, you might see the positions of texts are all corrupt. Please refer to Section 10.2 for a solution.

3.18 Zoom/link annotation

`anno_mark()` connects single row or column on the heatmap to a label, the next annotation function `anno_link()` connects subsets of rows or columns to plotting regions where more comprehensive graphics can be added there. See following example where we make boxplot for every row group.

```
set.seed(123)
m = matrix(rnorm(100*10), nrow = 100)
subgroup = sample(letters[1:3], 100, replace = TRUE, prob = c(1, 5, 10))
rg = range(m)
panel_fun = function(index, nm) {
  pushViewport(viewport(xscale = rg, yscale = c(0, 2)))
  grid.rect()
  grid.xaxis(gp = gpar(fontsize = 8))
  grid.boxplot(m[index, ], pos = 1, direction = "horizontal")
  popViewport()
}
anno = anno_link(align_to = subgroup, which = "row", panel_fun = panel_fun,
  size = unit(2, "cm"), gap = unit(1, "cm"), width = unit(4, "cm"))
```

```
Heatmap(m, name = "mat", right_annotation = rowAnnotation(foo = anno),
        row_split = subgroup)
```



The important arguments for `anno_zoom()` are:

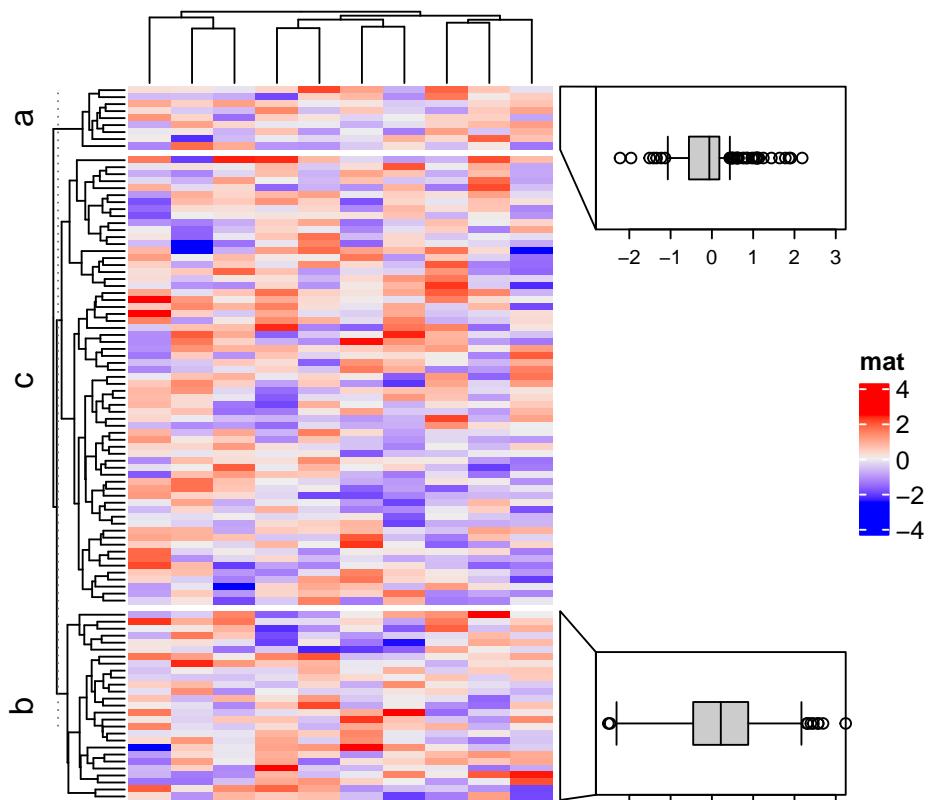
1. `align_to`: It defines how the plotting regions (or the boxes) correspond to the rows or the columns in the heatmap. If the value is a list of indices, each box corresponds to the rows or columns with indices in one vector in the list. If the value is a categorical variable (e.g. a factor or a character vector) that has the same length as the rows or columns in the heatmap, each box corresponds to the rows/columns in each level in the categorical variable.
2. `panel_fun`: A self-defined function that defines how to draw graphics in the box. The function must have a `index` argument which is the indices for the rows/columns that the box corresponds to. It can have a second argument `nm` which is the “name” of the selected part in the heatmap. The corresponding value for `nm` comes from `align_to` if it is specified as a categorical variable or a list with names.
3. `size`: The size of boxes. It can be pure numeric that they are treated as

relative fractions of the total height/width of the heatmap. The value of `size` can also be absolute units.

4. `gap`: Gaps between boxes. It should be a `unit` object.

In previous example, `align_to` is set as a categorical variable. In the next example, `align_to` is set as a list of indicies and only cover slices “a” and “b.”

```
align_to = split(1:nrow(m), subgroup)
align_to = align_to[c("a", "b")]
anno = anno_link(alignment = align_to, which = "row", panel_fun = panel_fun,
  size = unit(2, "cm"), gap = unit(1, "cm"), width = unit(4, "cm"))
Heatmap(m, name = "mat", right_annotation = rowAnnotation(foo = anno),
  row_split = subgroup)
```

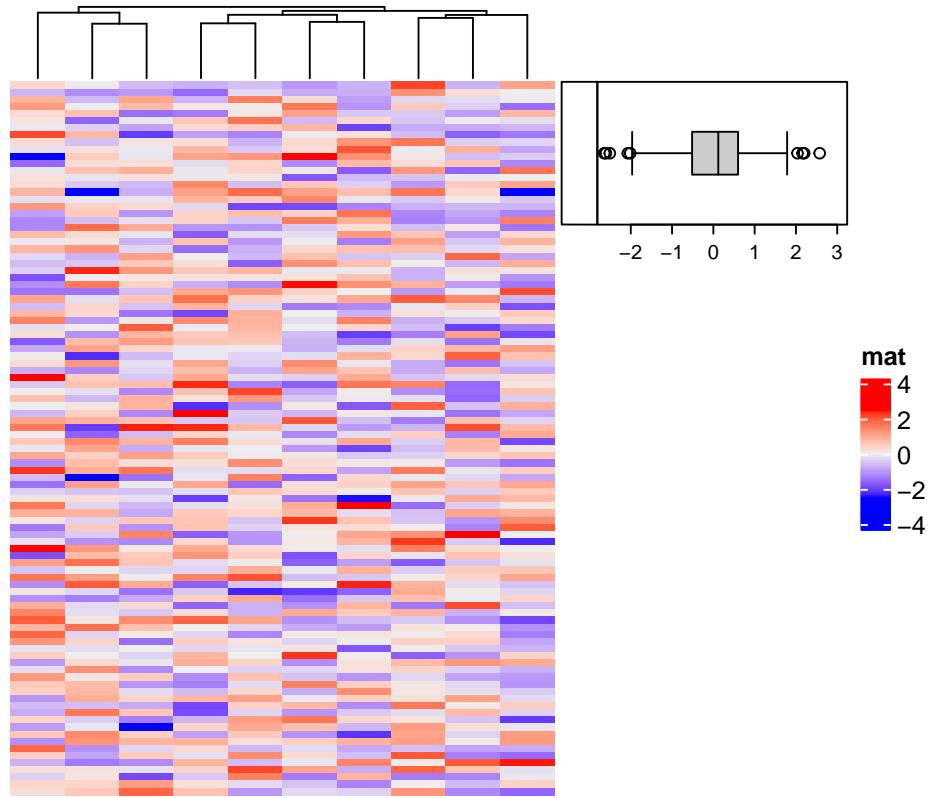


`anno_link()` also works for column annotations.

Similar as `anno_mark()`, the positions of the plotting regions also depends on the absolute size of the graphic device. If you resize the current interactive device or you use `grid.grabExpr()` to capture the current plot, you might see the positions of texts are all corrupt. Please refer to Section 10.2 for a solution.

As shown in the previous examples, `anno_link()` normally works together with `row_split/column_split` to associate additional graphics to slices. However, it is not necessary, as long as rows with indices in `align_to` are continuously adjacent on heatmaps.

```
align_to = list(1:20)
anno = anno_link(align_to = align_to, which = "row", panel_fun = panel_fun,
                 size = unit(2, "cm"), gap = unit(1, "cm"), width = unit(4, "cm"))
# here row indices 1 to 20 are adjacent since no clustering is applied on rows
Heatmap(m, name = "mat", right_annotation = rowAnnotation(foo = anno),
        cluster_rows = FALSE)
```



3.19 Text box annotation

From **ComplexHeatmap** 2.11.1, there are supports for drawing text boxes and associating them to heatmaps.

3.19.1 Construct the text box

Text box annotation depends on the text box “grob.” The text box grob can be constructed with the function `text_box_grob()`. It accpets a character vector with words or phrases/sentences. We first demonstrate the use of words.

Following functionn `random_text()` generates a vector of random words or phrases.

```
random_text = function(n, n_words = 1) {
  sapply(1:n, function(i) {
    w = replicate(sample(n_words, 1),
                 paste0(sample(letters, sample(4:10, 1)), collapse = ""))
    if(n_words > 1) {
      paste(w, collapse = " ")
    } else {
      w
    }
  })
}
```

Following code shows how to set graphics parameters:

```
set.seed(123)
words = random_text(10)
words

## [1] "sncjrkeyti" "hgjisd"      "kgul"       "mgixj"      "ugzfbe"
## [6] "mrafuoi"    "ptfkh"       "gpqvrxbdm" "sytvnchpl" "nczg"

grid.newpage()
gb = text_box_grob(words)
grid.draw(gb)
```

sncjrkeyti hgjisd kgul mgixj ugzfbe mrafuoi ptfkh
gpqvrxbdm sytvnchpl nczg

```
grid.newpage()
gb = text_box_grob(words, gp = gpar(col = 1:10))
grid.draw(gb)
```

sncjrkeyti hgjisd kgul mgixj ugzfbe mrafuoi ptfkh
gpqvrxbdm sytvnchpl nczg

```
grid.newpage()
gb = text_box_grob(words, gp = gpar(fontsize = runif(10, 5, 20)))
grid.draw(gb)
```

```
sncjrkeyti hgjisd kgul mgixj ugzfbe mrafuoi  
ptfkh gpqvrxbdm sytvnchpl ncsg
```

Following code shows how to set the background:

```
grid.newpage()  
gb = text_box_grob(words, background_gp = gpar(fill = "#CCCCCC", col = "#808080"))  
grid.draw(gb)
```

```
sncjrkeyti hgjisd kgul mgixj ugzfbe mrafuoi ptfkh  
gpqvrxbdm sytvnchpl ncsg
```

```
grid.newpage()  
gb = text_box_grob(words, background_gp = gpar(fill = "#CCCCCC", col = NA),  
round_corners = TRUE)  
grid.draw(gb)
```

```
sncjrkeyti hgjisd kgul mgixj ugzfbe mrafuoi ptfkh  
gpqvrxbdm sytvnchpl ncsg
```

```
grid.newpage()  
gb = text_box_grob(words, background_gp = gpar(fill = "#CCCCCC", col = "#808080"),  
padding = unit(4, "mm"))  
grid.draw(gb)
```

```
sncjrkeyti hgjisd kgul mgixj ugzfbe mrafuoi ptfkh  
gpqvrxbdm sytvnchpl ncsg
```

Following code shows how to set spaces between words and lines, and how to set the width of the text box:

```
grid.newpage()  
gb = text_box_grob(words, line_space = unit(5, "mm"))  
grid.draw(gb)
```

```
sncjrkeyti hgjisd kgul mgixj ugzfbe mrafuoi ptfkh  
gpqvrxbdm sytvnchpl ncsg
```

```
grid.newpage()  
gb = text_box_grob(words, text_space = unit(5, "mm"))  
grid.draw(gb)
```

```
sncjrkeyti hgjisd kgul mgixj ugzfbe mrafuoi  
ptfkh gpqvrxbdm sytvnchpl ncsg
```

```
grid.newpage()
gb = text_box_grob(words, max_width = unit(12, "cm"))
grid.draw(gb)
```

sncjrkeyti hgjisd kgul mgixj ugzfbe mrafuoi ptfkh gpqvrxbdm
sytvnchpl nc zg

Words can be added from the bottom left of the box or from the top left:

```
fontsize = sort(runif(10, 5, 20))
grid.newpage()
gb = text_box_grob(words, gp = gpar(fontsize = fontsize))
grid.draw(gb)
```

sncjrkeyti hgjisd kgul mgixj ugzfbe mrafuoi ptfkh
gpqvrxbdm sytvnchpl nc zg

```
grid.newpage()
gb = text_box_grob(words, gp = gpar(fontsize = fontsize),
                    first_text_from = "bottom")
grid.draw(gb)
```

gpqvrxbdm sytvnchpl nc zg
sncjrkeyti hgjisd kgul mgixj ugzfbe mrafuoi ptfkh

Sentences or phrases are composed of several words, so there are some extra parameters. If the longest sentence is longer than `max_width`, the width of the longest sentence will be the width of the text box. Setting `word_wrap = TRUE` can adjust the text according to the width of the text box.

```
sentences = random_text(5, 8)
sentences

## [1] "ihztacybxv gmtnpzdyls"
## [2] "ilhgo qkgscf djqpjhclne fiatnsebx"
## [3] "dkevzsc gewi xque bvrjdjh"
## [4] "ekoxjw"
## [5] "lidcgxh exfdckjz kjos jmxcrzh jgyfqta jmzbhvc fnosi"
fontsize = runif(5, 5, 20)
grid.newpage()
gb = text_box_grob(sentences, gp = gpar(col = 1:5, fontsize = fontsize))
grid.draw(gb)
```

```
ihztacybx gmtnpzdyls ilhgo qkgscf djqphucne fiatnsebx
dkevzsc gewi xque bvrjoh ekoxjw
lidcgxh exfdckjz kjos jmxcrzh jgyfqta jmzbhvc fnosi
```

```
grid.newpage()
gb = text_box_grob(sentences, gp = gpar(col = 1:5, fontsize = fontsize),
                    word_wrap = TRUE)
grid.draw(gb)
```

ihztacybx gmtnpzdyls ilhgo qkgscf djqphucne
flatnsebx dkevzsc gewi xque bvrjoh ekoxjw
lidcgxh exfdckjz kjos jmxcrzh
jgyfqta jmzbhvc fnosi

Also argument `add_new_line` can be set to `TRUE` so each sentence will be in a single line.

```
grid.newpage()
gb = text_box_grob(sentences, gp = gpar(col = 1:5, fontsize = fontsize),
                    add_new_line = TRUE)
grid.draw(gb)
```

ihztacybx gmtnpzdyls
ilhgo qkgscf djqphucne fiatnsebx
dkevzsc gewi xque bvrjoh
ekoxjw
lidcgxh exfdckjz kjos jmxcrzh jgyfqta jmzbhvc fnosi

```
grid.newpage()
gb = text_box_grob(sentences, gp = gpar(col = 1:5, fontsize = fontsize),
                    add_new_line = TRUE, word_wrap = TRUE)
grid.draw(gb)
```

ihztacybx gmtnpzdyls
ilhgo qkgscf djqphucne fiatnsebx
dkevzsc gewi xque bvrjoh
ekoxjw
lidcgxh exfdckjz kjos jmxcrzh
jgyfqta jmzbhvc fnosi

Finally, the width and height of the text box grob can be obtained by `grobWidth()` and `grobHeight()`. Also there is a companion function `grid.text_box()` that directly draws text box at a certain position.

```
# code only for demonstration
gb = text_box_grob(words)
grobWidth(gb)
grobHeight(gb)
grid.text_box(words, x, y)
```

3.19.2 Draw text box annotation

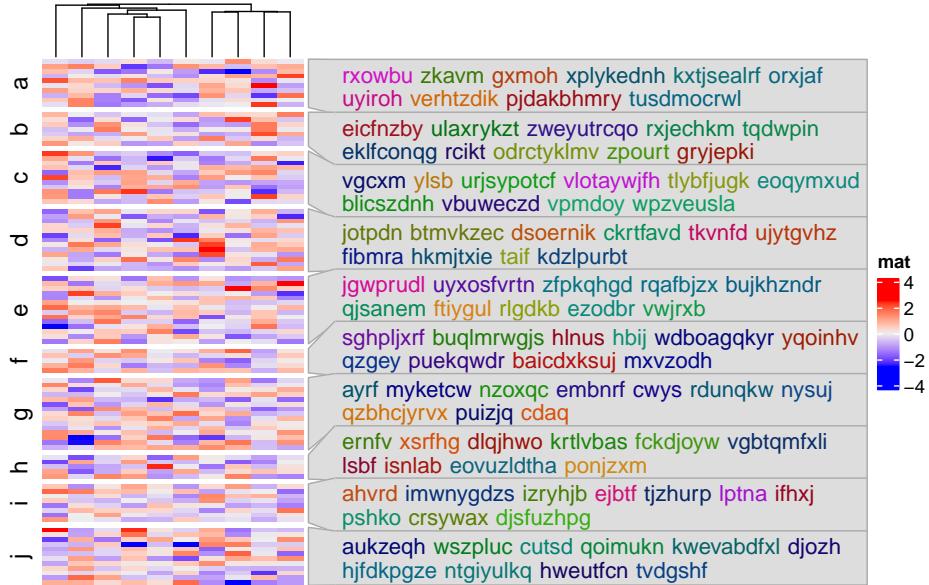
The function `anno_text_box()` draws several text boxes and associate them to heatmaps. Note you don't need to directly use `text_box_grob()` function with `anno_text_box()`, but there are parameters that control texts in `anno_text_box()` that are directly passed to `text_box_grob()`.

Since texts are already wrapped into boxes, the text box annotations are basically implemented by `anno_link()` and `anno_block()`.

By default, the text box annotation is implemented by `anno_link()`. Since heights of text boxes are not always the same as the height of subsets of rows they correspond to, there are connections between the heatmap and text boxes. In the following example, we generated 10 text boxes and each one is linked to a heatmap slice. In this case, the first parameter for `anno_text_box()` is a categorical variable which also split heatmap rows. The second parameter `text` is a named list of character vectors where name of the list should correspond to the levels in `split`.

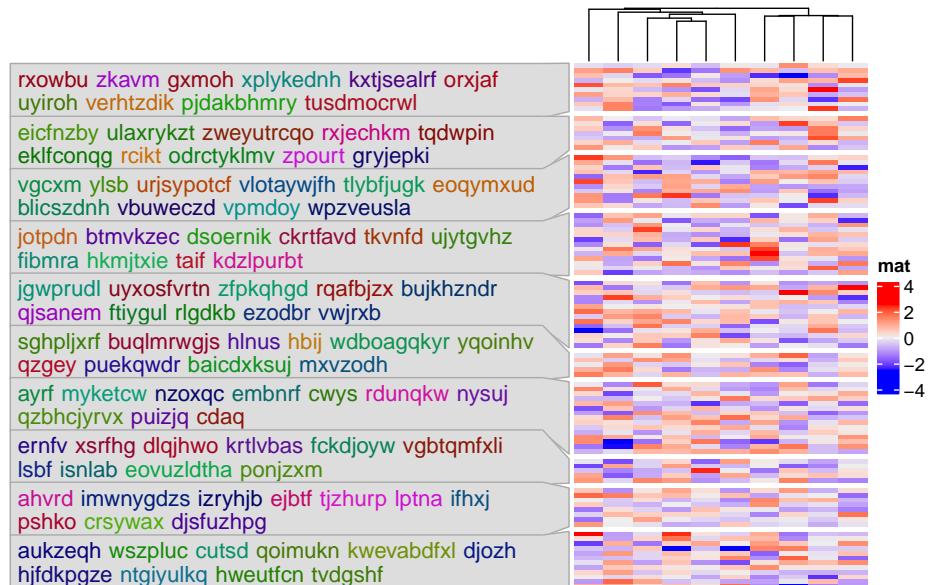
```
mat = matrix(rnorm(100*10), nrow = 100)
split = sample(letters[1:10], 100, replace = TRUE)
text = lapply(unique(split), function(x) {
  random_text(10)
})
names(text) = unique(split)

Heatmap(mat, name = "mat", cluster_rows = FALSE, row_split = split,
        right_annotation = rowAnnotation(text_box = anno_text_box(split, text)))
)
```



Or put on the left side of the heatmap:

```
Heatmap(mat, name = "mat", cluster_rows = FALSE, row_split = split, row_title = NULL,
       left_annotation = rowAnnotation(text_box = anno_text_box(split, text, side = "left")))
```

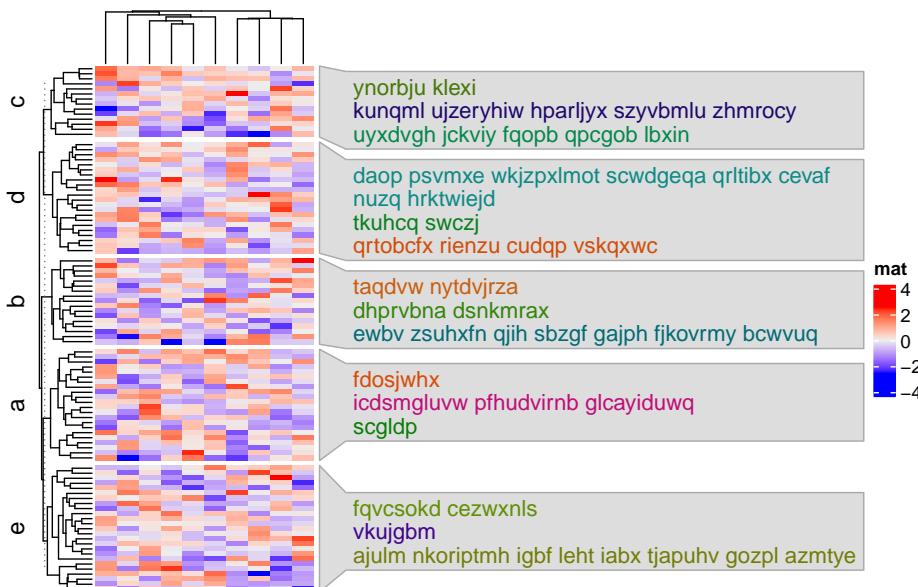


Parameters for `text_box_grob()` can be directly passed in `anno_text_box()`. E.g. when the text are sentences, to control word wrap and new lines:

```

split = sample(letters[1:5], 100, replace = TRUE)
sentences = lapply(unique(split), function(x) {
  random_text(3, 8)
})
names(sentences) = unique(split)

Heatmap(mat, name = "mat", row_split = split,
        right_annotation = rowAnnotation(
          text_box = anno_text_box(
            split, sentences,
            word_wrap = TRUE,
            add_new_line = TRUE)
        )
      )
    )
  )
)
  
```

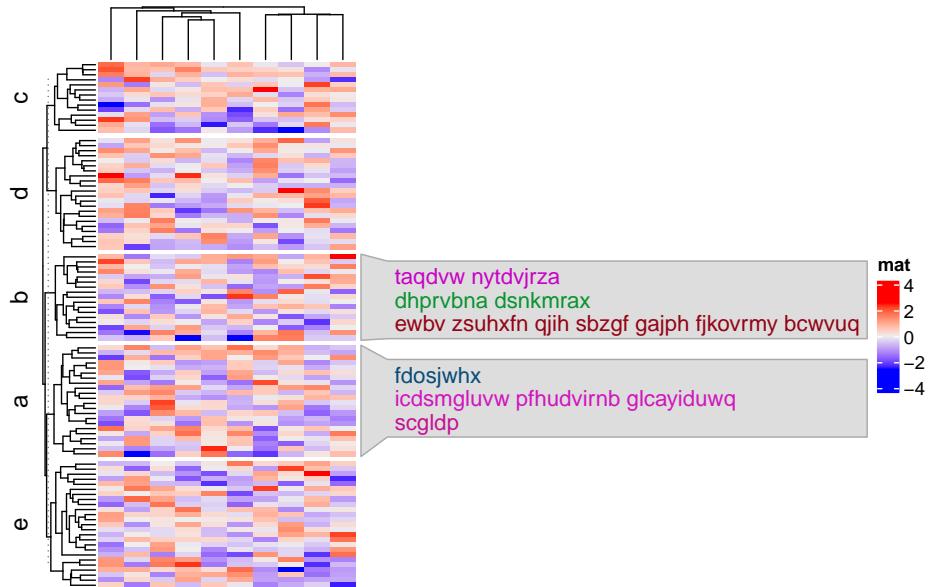


The first argument for `anno_text_box()` can also be set as a list of indices. Note the indices for rows should be continuously adjacent on heatmaps.

```

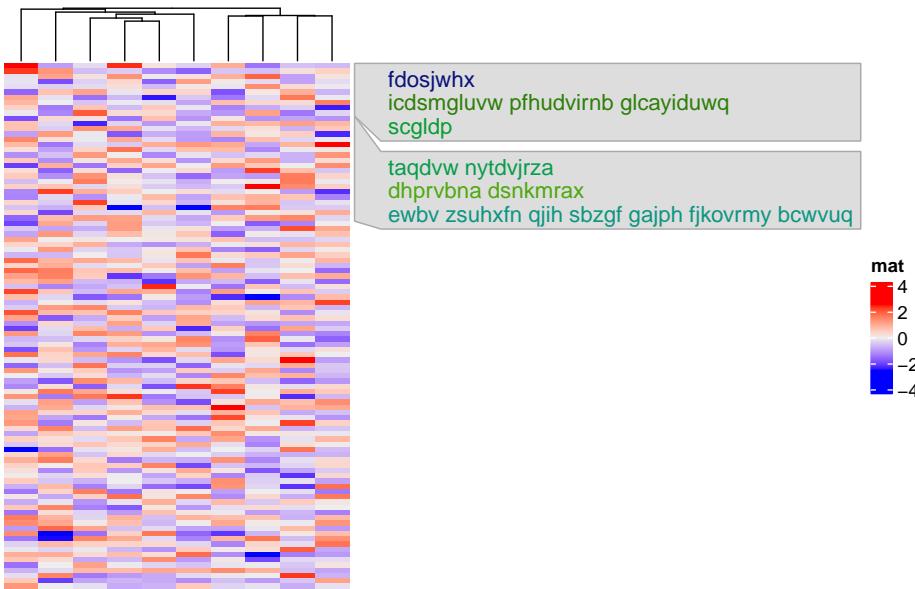
align_to = split(seq_along(split), split)
Heatmap(mat, name = "mat", row_split = split,
        right_annotation = rowAnnotation(
          text_box = anno_text_box(
            align_to[c("a", "b")],
            sentences[c("a", "b")], # names should match
            word_wrap = TRUE,
            add_new_line = TRUE)
        )
      )
    )
  )
)
  
```

)



`anno_text_box()` works mainly together with `row_split` to associate more information to every heatmap slice. But this is not necessary, as long as you can ensure the indices are continuously adjacent on heatmaps. In the next example, rows are not clustered, we add two text boxes to correspond to row 1-10 and row 11-30.

```
Heatmap(mat, name = "mat", cluster_rows = FALSE,
        right_annotation = rowAnnotation(
          text_box = anno_text_box(
            list("a" = 1:10, "b" = 11:30),
            sentences[c("a", "b")],
            word_wrap = TRUE,
            add_new_line = TRUE)
        )
    )
```



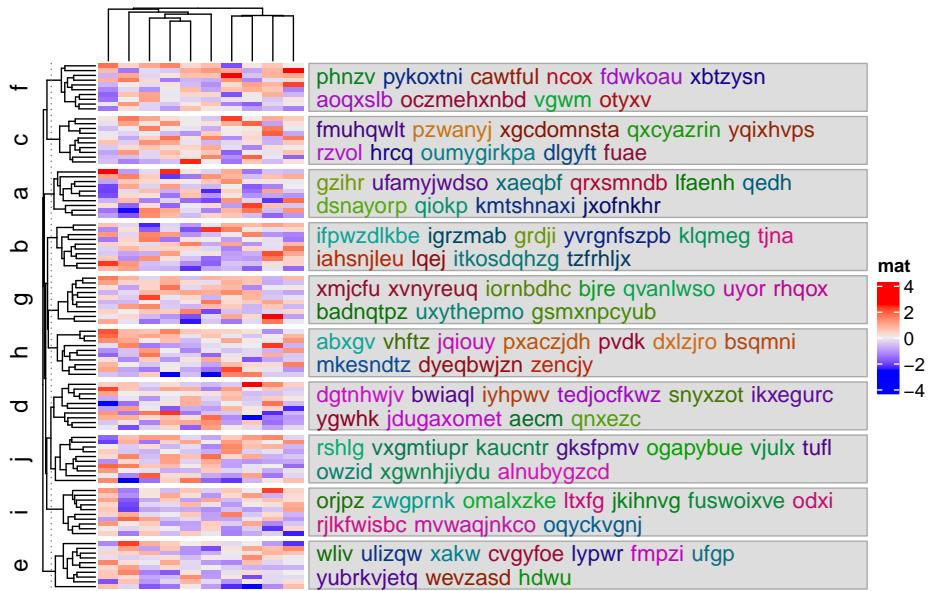
There are some cases that users want to put the text boxes exactly at the positions of the corresponding heatmap slices. In this case, the `by` argument should be set to `"anno_block"`. And now `anno_block()` is internally used to implement text box annotations.

```

split = rep(letters[1:10], 10)
text = lapply(unique(split), function(x) {
  random_text(10)
})
names(text) = unique(split)

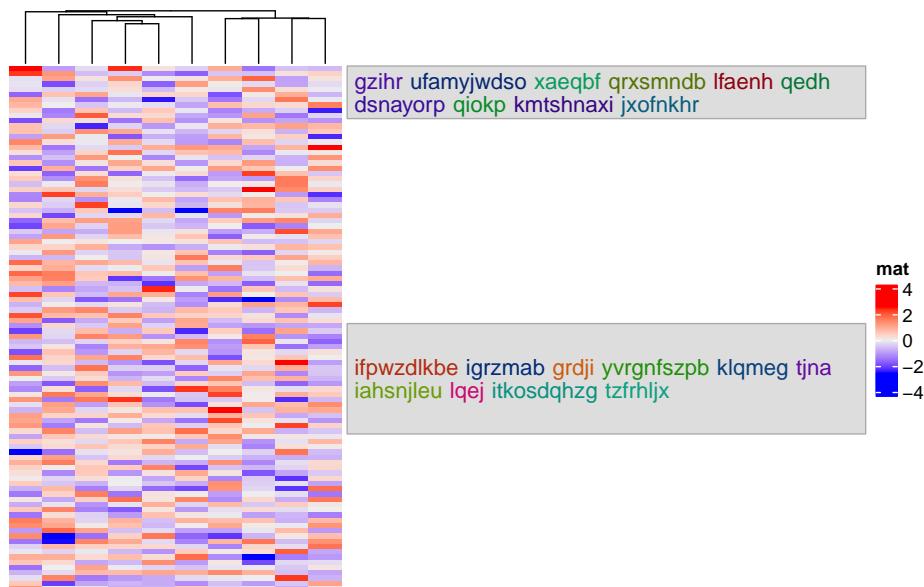
Heatmap(mat, name = "mat", row_split = split,
        right_annotation = rowAnnotation(
          text_box = anno_text_box(split, text, by = "anno_block")
        )
)

```



Similarly, the first parameter can be set as a list of indicies:

```
align_to = split(seq_along(split), split)
Heatmap(mat, name = "mat", row_split = split,
        right_annotation = rowAnnotation(
          text_box = anno_text_box(
            align_to[c("a", "b")],
            text[c("a", "b")], # names should match
            by = "anno_block")
        )
    )
```

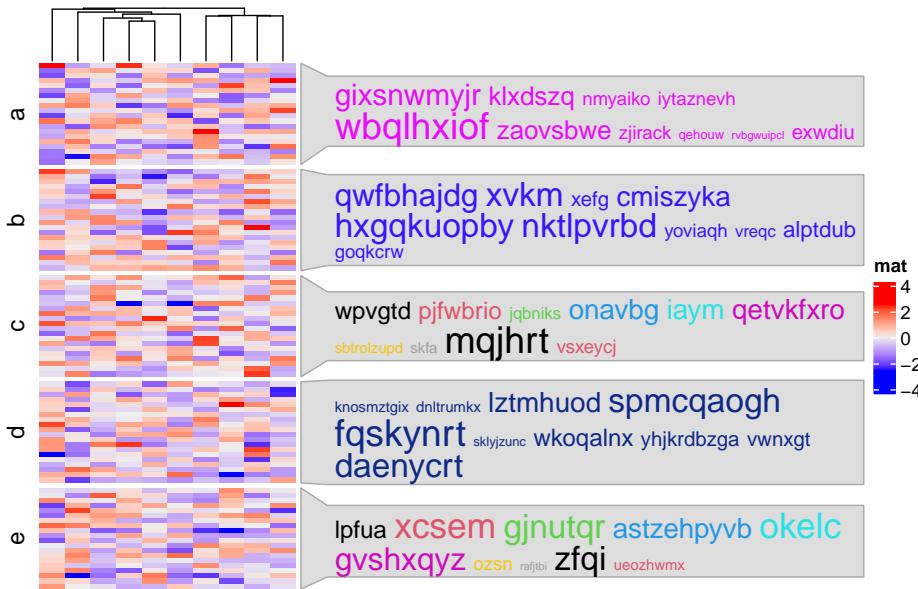


So far, the texts are assigned with random colors. Now the question is how to exactly control graphics parameters for texts in the text boxes. In previous examples, the value of `text` is a list of character vectors. The graphics parameters can be integrated into `text` by setting `text` as a list of data frames. In each data frame, the first column contains texts that will be put into boxes. The data frame can also contain the following four columns ("`col`", "`fontsize`", "`fontfamily`" and "`fontface`") to exactly control the corresponding text.

```
split = rep(letters[1:5], 20)
text = lapply(unique(split), function(x) {
  df = data.frame(text = random_text(10))
  df$fontsize = runif(10, 6, 20)
  if(runif(1) > 0.5) {
    df$col = rep(rand_color(1), 10)
  } else {
    df$col = 1:10
  }
  df
})
names(text) = unique(split)
head(text[[1]])
```

```
##          text  fontsize      col
## 1 gixsnwmyjr 16.409436 #E80DF8FF
## 2 klxdszq 14.065497 #E80DF8FF
## 3 nmyaiko 10.148988 #E80DF8FF
## 4 iytaznevh 9.719641 #E80DF8FF
## 5 wbqlhxiof 19.864118 #E80DF8FF
## 6 zaovsbwe 14.295501 #E80DF8FF

Heatmap(mat, name = "mat", cluster_rows = FALSE, row_split = split,
        right_annotation = rowAnnotation(text_box = anno_text_box(split, text)))
)
```



In the **simplifyEnrichment** package, we use `anno_text_box()` to implement a word cloud annotation to visualize summaries of biological functions in each GO cluster.

3.20 Summary annotation

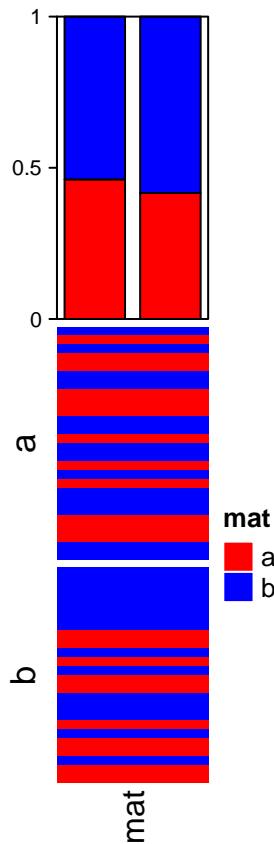
There is one special annotation `anno_summary()` which only works with one-column heatmap or one-row heatmap (we can say the heatmap only contains a vector). It shows summary statistics for the vector in the heatmap. If the corresponding vector is discrete, the summary annotation is presented as barplots and if the vector is continuous, the summary annotation is boxplot. `anno_summary()` is always used when the heatmap is split so that statistics can be compared between heatmap slices.

The first example shows the summary annotation for discrete heatmap. The barplot shows the proportion of each level in each slice. The absolute values can already be seen by the height of the heatmap slice.

The color schema for the barplots is automatically extracted from the heatmap.

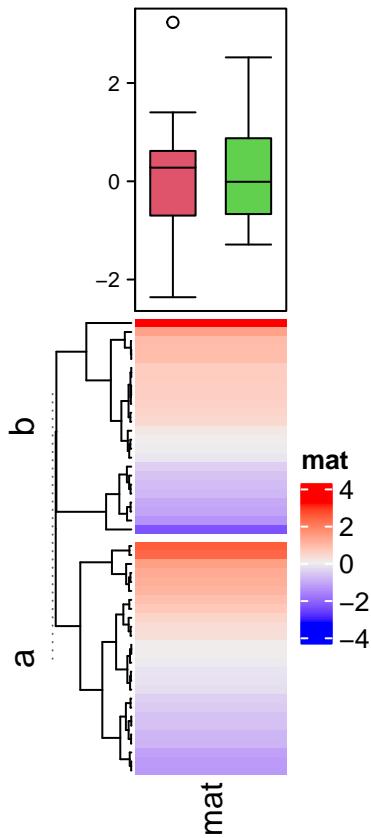
```
ha = HeatmapAnnotation(summary = anno_summary(height = unit(4, "cm")))
v = sample(letters[1:2], 50, replace = TRUE)
split = sample(letters[1:2], 50, replace = TRUE)

Heatmap(v, name = "mat", col = c("a" = "red", "b" = "blue"),
        top_annotation = ha, width = unit(2, "cm"), row_split = split)
```



The second example shows the summary annotation for continuous heatmap. The graphic parameters should be manually set by `gp`. The legend of the boxplot can be created and added as introduced in Section 5.2, last second paragraph.

```
ha = HeatmapAnnotation(summary = anno_summary(gp = gpar(fill = 2:3),
                                               height = unit(4, "cm")))
v = rnorm(50)
Heatmap(v, name = "mat", top_annotation = ha, width = unit(2, "cm"),
        row_split = split)
```



Normally we don't draw this one-column heatmap along. It is always combined with other "main heatmaps." E.g. A gene expression matrix with a one-column heatmap which shows whether the gene is a protein coding gene or a linc-RNA gene.

In following, we show a simple example of a "main heatmap" with two one-column heatmaps. The functionality of heatmap concatenation will be introduced in Chapter 4.

```
m = matrix(rnorm(50*10), nrow = 50)
ht_list = Heatmap(m, name = "main_matrix")

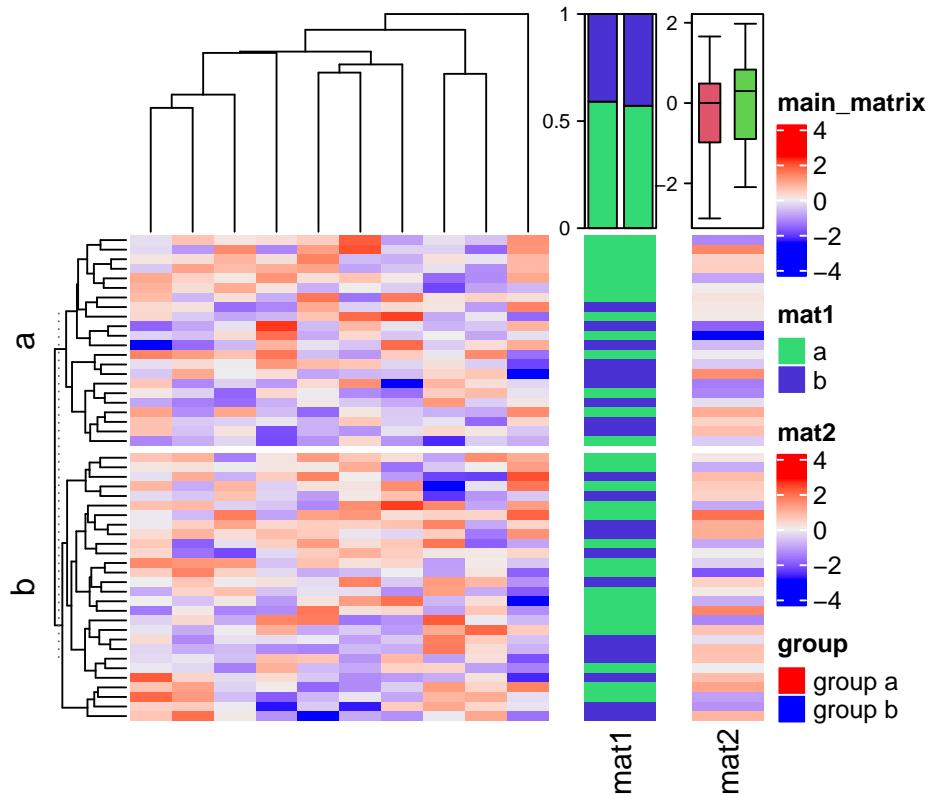
ha = HeatmapAnnotation(summary = anno_summary(height = unit(3, "cm")))
v = sample(letters[1:2], 50, replace = TRUE)
ht_list = ht_list + Heatmap(v, name = "mat1", top_annotation = ha, width = unit(1, "cm"))

ha = HeatmapAnnotation(summary = anno_summary(gp = gpar(fill = 2:3),
height = unit(3, "cm")))
v = rnorm(50)
ht_list = ht_list + Heatmap(v, name = "mat2", top_annotation = ha, width = unit(1, "cm"))
```

```

split = sample(letters[1:2], 50, replace = TRUE)
lgd_boxplot = Legend(labels = c("group a", "group b"), title = "group",
                      legend_gp = gpar(fill = c("red", "blue")))
draw(ht_list, row_split = split, ht_gap = unit(5, "mm"),
      heatmap_legend_list = list(lgd_boxplot))

```



3.21 Multiple annotations

3.21.1 General settings

As mentioned before, to put multiple annotations in `HeatmapAnnotation()`, they just need to be specified as name-value pairs. In `HeatmapAnnotation()`, there are some arguments which controls multiple annotations. For these arguments, they are specified as a vector which has same length as number of the annotations, or a named vector with subset of the annotations.

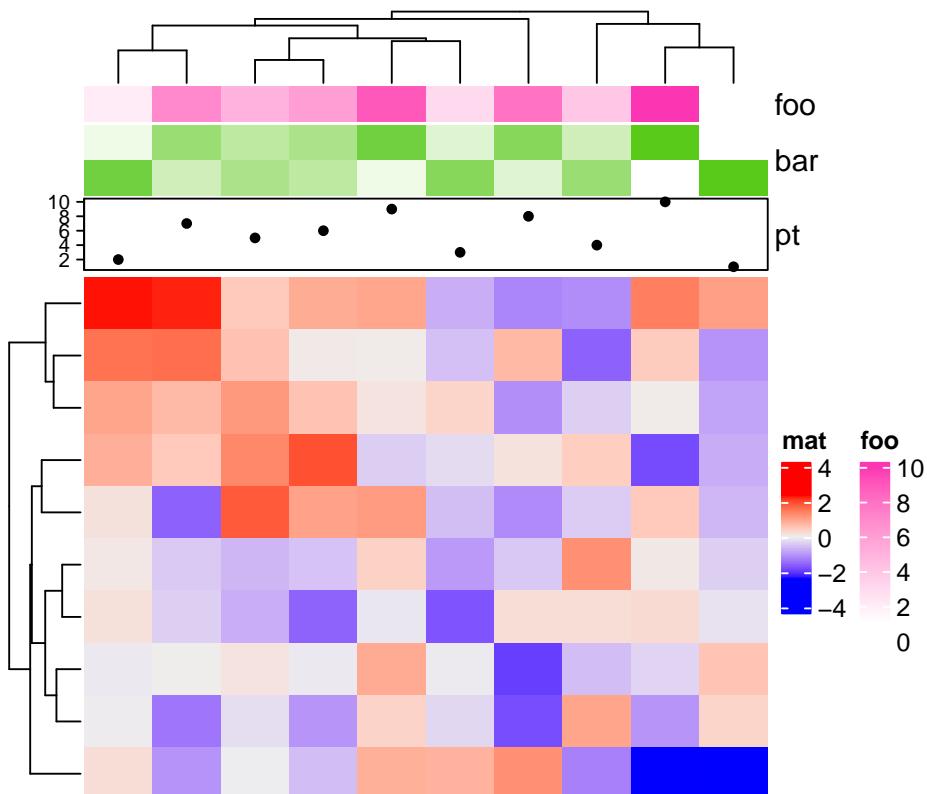
The simple annotations which are specified as vectors, matrices and data frames will automatically have legends on the heatmap. `show_legend` controls whether

to draw the legends for them. Note here if `show_legend` is a vector, the value of `show_legend` should be in one of the following formats:

- A logical vector with the same length as the number of simple annotations.
- A logical vector with the same length as the number of total annotations. The values for complex annotations are ignored.
- A named vector to control subset of the simple annotations.

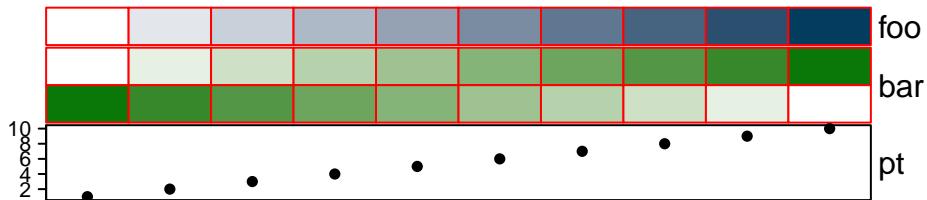
For customization on the annotation legends, please refer to Section 5.4.

```
ha = HeatmapAnnotation(foo = 1:10,
  bar = cbind(1:10, 10:1),
  pt = anno_points(1:10),
  show_legend = c("bar" = FALSE))
)
Heatmap(matrix(rnorm(100), 10), name = "mat", top_annotation = ha)
```



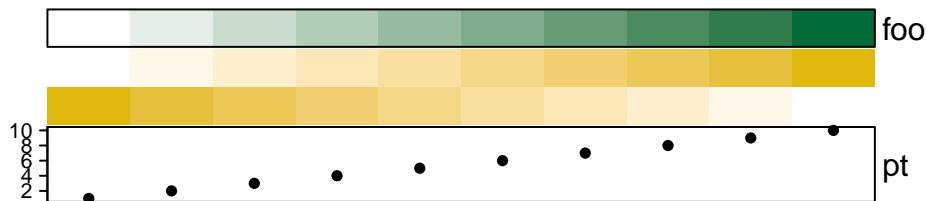
`gp` controls graphic parameters (except `fill`) for the simple annotations, such as the border of annotation grids.

```
ha = HeatmapAnnotation(foo = 1:10,
                      bar = cbind(1:10, 10:1),
                      pt = anno_points(1:10),
                      gp = gpar(col = "red")
)
```



`border` controls the border of every single annotations. `show_annotation_name` controls whether show annotation names. As mentioned, the value can be a single value, a vector or a named vector.

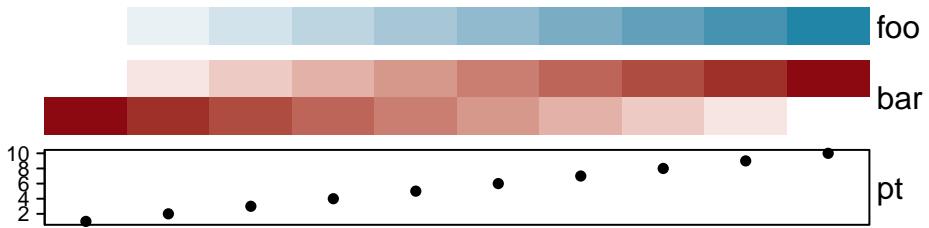
```
ha = HeatmapAnnotation(foo = 1:10,
                      bar = cbind(1:10, 10:1),
                      pt = anno_points(1:10),
                      show_annotation_name = c(bar = FALSE), # only turn off `bar`
                      border = c(foo = TRUE) # turn on `foo`
)
```



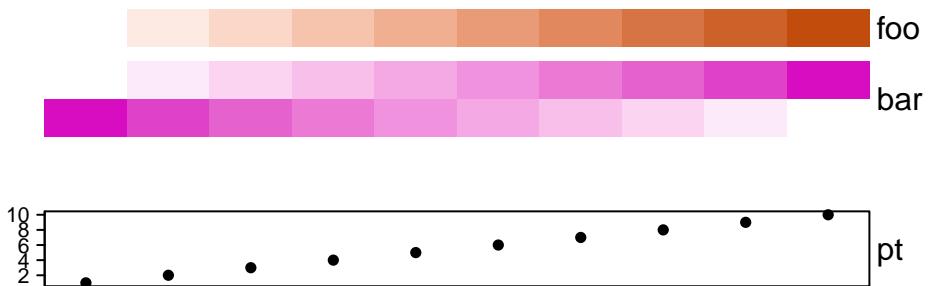
`annotation_name_gp`, `annotation_name_offset`, `annotation_name_side` and `annotation_name_rot` control the style and position of the annotation names. The latter three can be specified as named vectors. If `annotation_name_offset` is specified as a named vector, it can be specified as characters while not `unit` objects: `annotation_name_offset = c(foo = "1cm")`.

`gap` controls the space between every two neighbouring annotations. The value can be a single unit or a vector of units.

```
ha = HeatmapAnnotation(foo = 1:10,
                      bar = cbind(1:10, 10:1),
                      pt = anno_points(1:10),
                      gap = unit(2, "mm"))
```



```
ha = HeatmapAnnotation(foo = 1:10,
                      bar = cbind(1:10, 10:1),
                      pt = anno_points(1:10),
                      gap = unit(c(2, 10), "mm"))
```

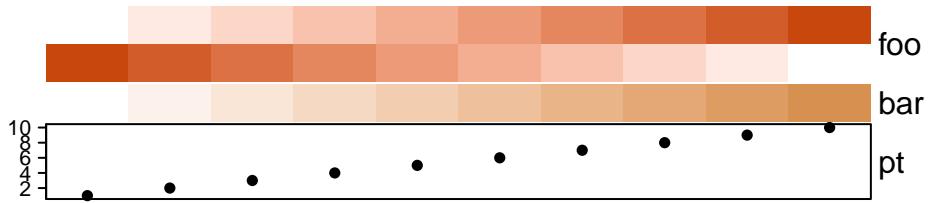


3.21.2 Size of annotations

`height`, `width`, `annotation_height` and `annotation_width` control the height or width of the complete heatmap annotations. Normally you don't need to set them because all the single annotations have fixed height/width and the final height/width for the whole heatmap annotation is the sum of them. Resizing these values will involve rather complicated adjustment depending on whether it is a simple annotation or complex annotation. The resizing of heatmap annotations will also happen when adjusting a list of heatmaps. In following examples, we take column annotations as examples and demonstrate some scenarios for the resizing adjustment.

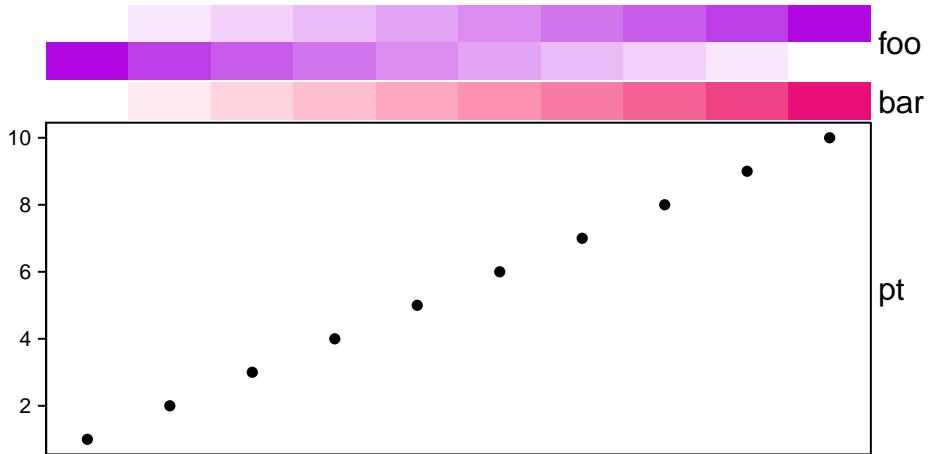
First the default height of `ha`:

```
# foo: 1cm, bar: 5mm, pt: 1cm
ha = HeatmapAnnotation(foo = cbind(1:10, 10:1),
                      bar = 1:10,
                      pt = anno_points(1:10))
```



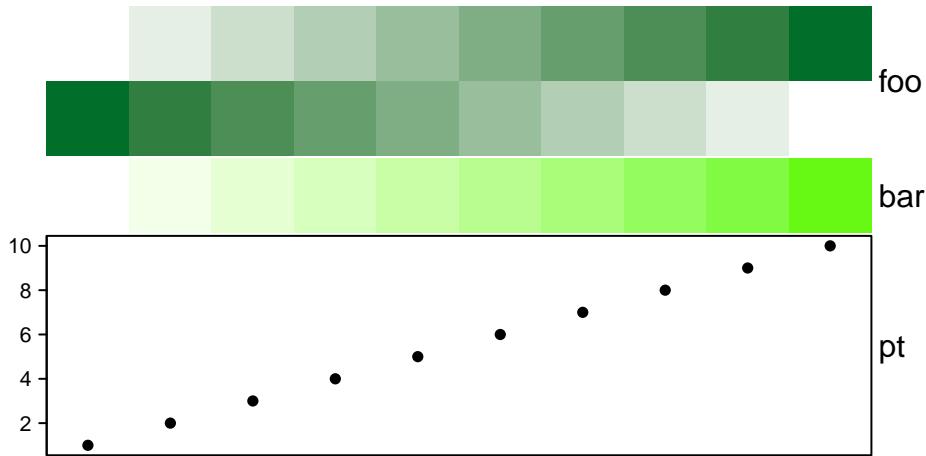
If `height` is set, the size of the simple annotation will not change, while only the complex annotations are adjusted. If there are multiple complex annotations, they are adjusted according to the ratio of their original size.

```
# foo: 1cm, bar: 5mm, pt: 4.5cm
ha = HeatmapAnnotation(foo = cbind(1:10, 10:1),
                      bar = 1:10,
                      pt = anno_points(1:10),
                      height = unit(6, "cm"))
```



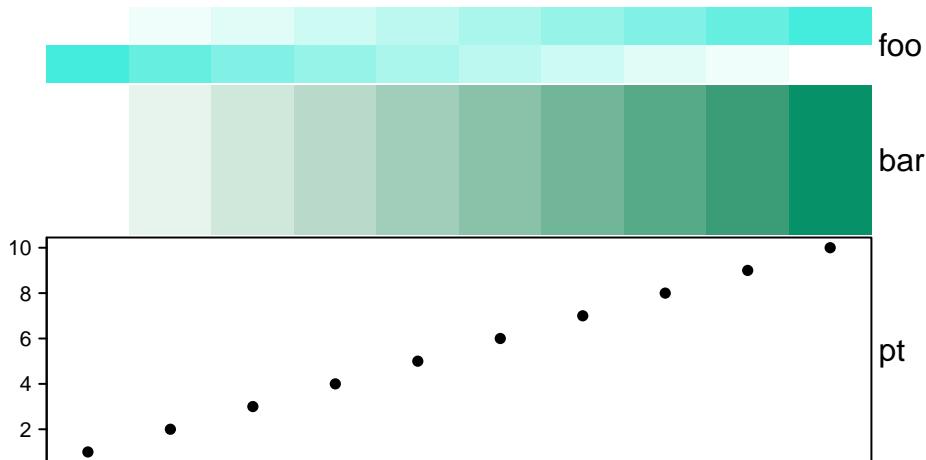
`simple_anno_size` controls the height of all simple annotations. Recall `ht_opt$simple_anno_size` can be set to globally control the size of simple annotations in all heatmaps.

```
# foo: 2cm, bar:1cm, pt: 3cm
ha = HeatmapAnnotation(foo = cbind(1:10, 10:1),
                      bar = 1:10,
                      pt = anno_points(1:10),
                      simple_anno_size = unit(1, "cm"), height = unit(6, "cm"))
```



If `annotation_height` is set as a vector of absolute units, the height of all three annotations are adjusted accordingly.

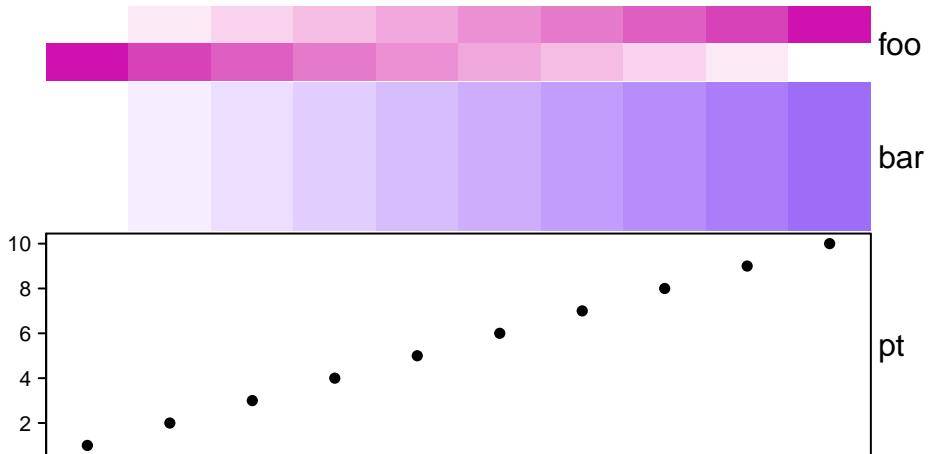
```
# foo: 1cm, bar: 2cm, pt: 3cm
ha = HeatmapAnnotation(foo = cbind(1:10, 10:1),
                      bar = 1:10,
                      pt = anno_points(1:10),
                      annotation_height = unit(1:3, "cm"))
```



If `annotation_height` is set as pure numbers which is treated as relative ratios for annotations, `height` should also be set as an absolute unit and the size of every single annotation is adjusted by the ratios.

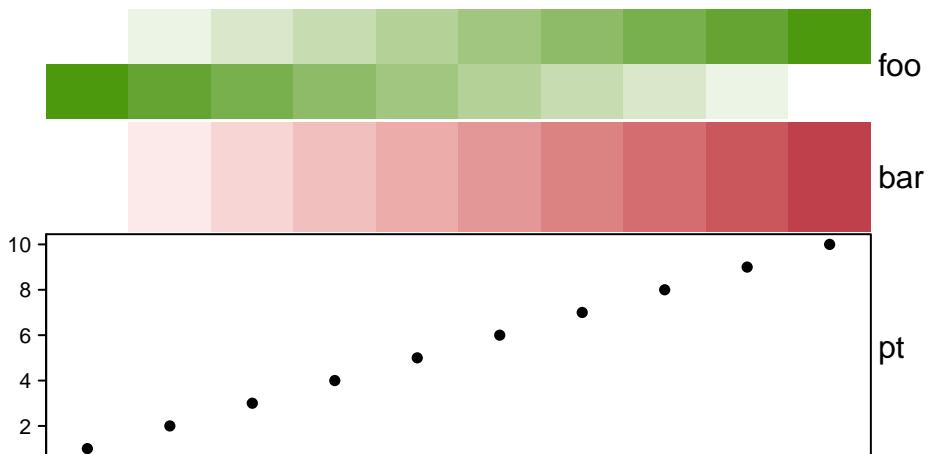
```
# foo: 1cm, bar: 2cm, pt: 3cm
ha = HeatmapAnnotation(foo = cbind(1:10, 10:1),
                      bar = 1:10,
```

```
pt = anno_points(1:10),
annotation_height = 1:3, height = unit(6, "cm"))
```

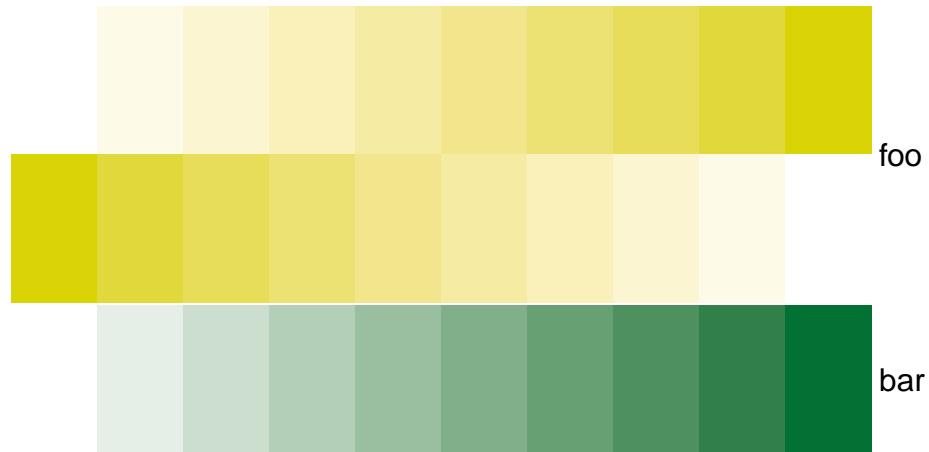


`annotation_height` can be mixed with relative units (in `null` unit) and absolute units.

```
# foo: 1.5cm, bar: 1.5cm, pt: 3cm
ha = HeatmapAnnotation(foo = cbind(1:10, 10:1),
  bar = 1:10,
  pt = anno_points(1:10),
  annotation_height = unit(c(1, 1, 3), c("null", "null", "cm")), height = unit(6, "cm"))
)
```



```
# foo: 2cm, bar: 1cm, pt: 3cm
ha = HeatmapAnnotation(foo = cbind(1:10, 10:1),
```

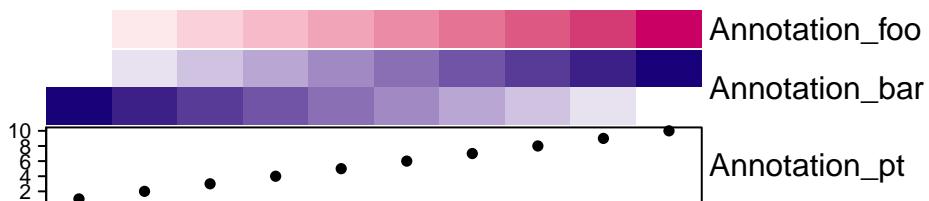



Section 4.6 introduces how the annotation sizes are adjusted among a list of heatmaps.

3.21.3 Annotation labels

From version 2.3.3, alternative labels for annotations can be set by `annotation_label` argument:

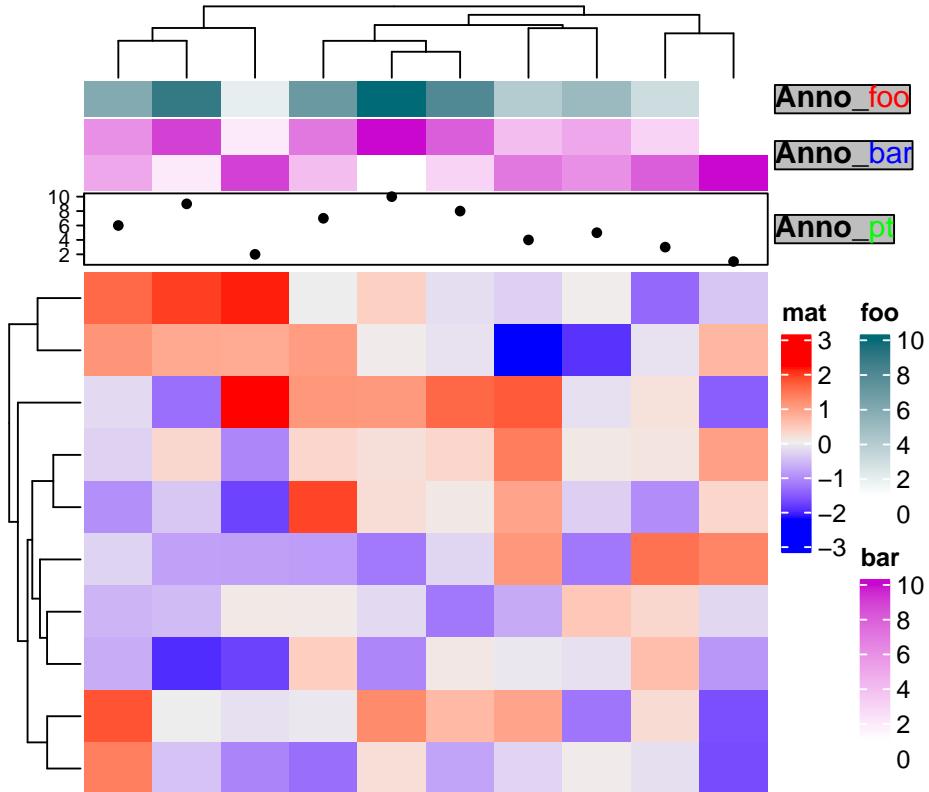
```
ha = HeatmapAnnotation(foo = 1:10,
                      bar = cbind(1:10, 10:1),
                      pt = anno_points(1:10),
                      annotation_label = c("Annotation_foo", "Annotation_bar", "Annotation_pt")
)
```



Annotation labels can also be set with complex text:

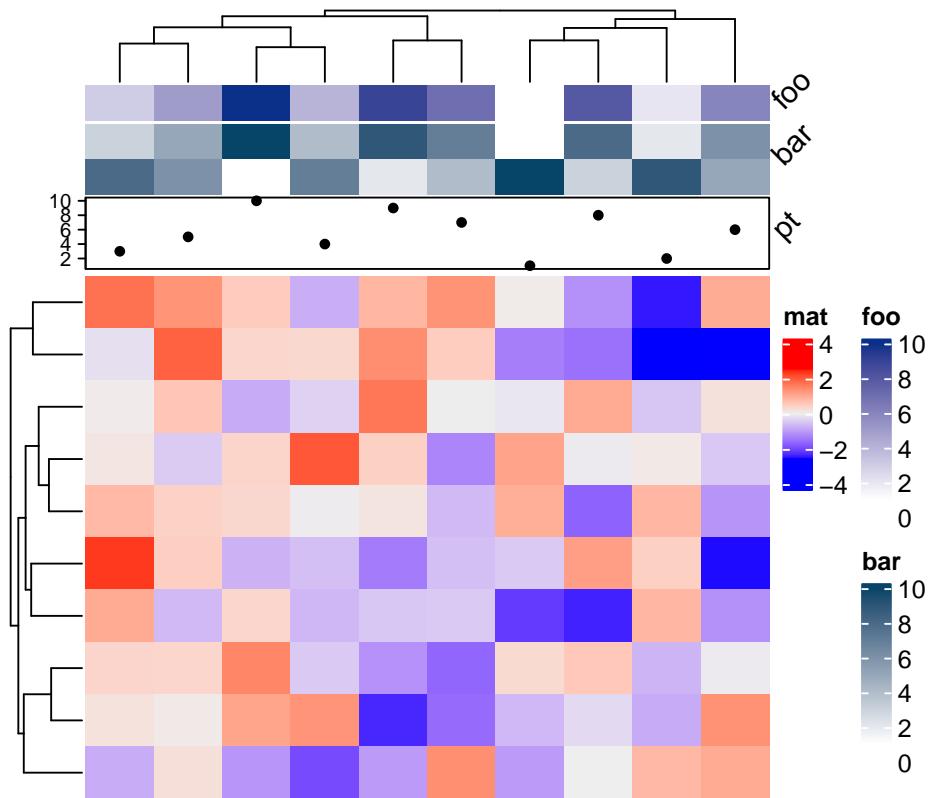
```
ha = HeatmapAnnotation(foo = 1:10,
                      bar = cbind(1:10, 10:1),
                      pt = anno_points(1:10),
                      annotation_label = gt_render(
                        c("**Anno**_<span style='color:red'>foo</span>",
                          "**Anno**_<span style='color:blue'>bar</span>",
                          "**Anno**_<span style='color:green'>pt</span>"),
                        gp = gpar(box_fill = "grey")
```

```
)
)
Heatmap(matrix(rnorm(100), 10), name = "mat", top_annotation = ha)
```



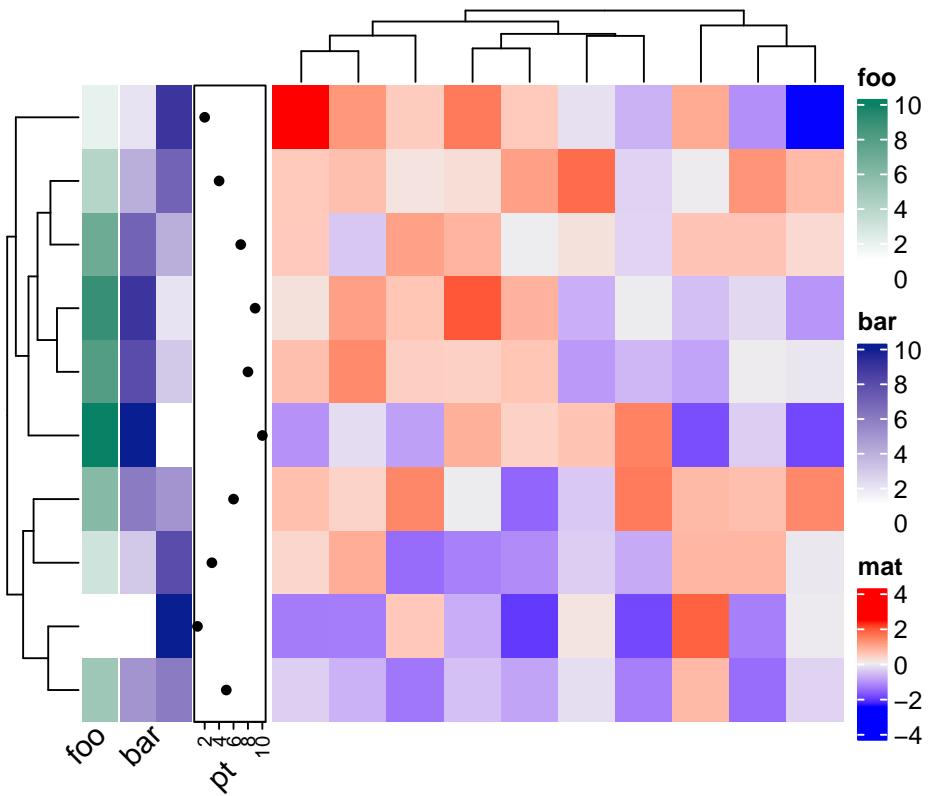
From version 2.5.6, rotations of annotation names can be configured.

```
ha = HeatmapAnnotation(foo = 1:10,
  bar = cbind(1:10, 10:1),
  pt = anno_points(1:10),
  annotation_name_rot = 45
)
Heatmap(matrix(rnorm(100), 10), name = "mat", top_annotation = ha)
```



Or on the rows:

```
ha = rowAnnotation(foo = 1:10,
                   bar = cbind(1:10, 10:1),
                   pt = anno_points(1:10),
                   annotation_name_rot = 45
)
Heatmap(matrix(rnorm(100), 10), name = "mat", left_annotation = ha)
```



3.22 Utility functions

There are some utility functions which make the manipulation of heatmap annotation easier. Just see following examples.

```
ha = HeatmapAnnotation(foo = 1:10,
                      bar = cbind(1:10, 10:1),
                      pt = anno_points(1:10))
length(ha)
```

```
## [1] 3
nobs(ha)
```

```
## [1] 10
```

Get or set the names of the annotations:

```
names(ha)
```

```
## [1] "foo" "bar" "pt"
```

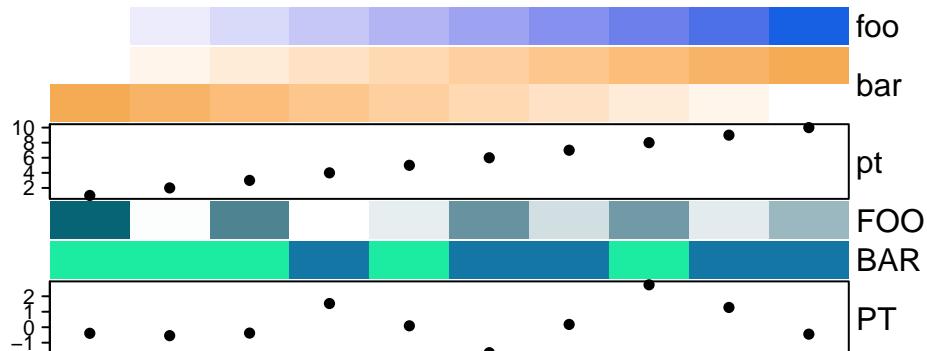
```
names(ha) = c("FOO", "BAR", "PT")
names(ha)
```

```
## [1] "FOO" "BAR" "PT"
```

You can concatenate two `HeatmapAnnotation` objects if they contain same number of observations and different annotation names.

```
ha1 = HeatmapAnnotation(foo = 1:10,
                       bar = cbind(1:10, 10:1),
                       pt = anno_points(1:10))
ha2 = HeatmapAnnotation(FOO = runif(10),
                       BAR = sample(c("a", "b"), 10, replace = TRUE),
                       PT = anno_points(rnorm(10)))
ha = c(ha1, ha2)
names(ha)
```

```
## [1] "foo" "bar" "pt"   "FOO" "BAR" "PT"
```

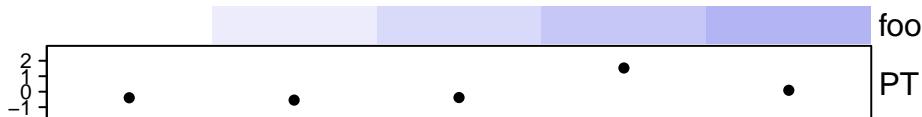


`HeatmapAnnotation` object sometimes is subsettable. The row index corresponds to observations in the annotation and column index corresponds to the annotations. If the annotations are all simple annotations or the complex annotation created by `anno_*`() functions in **ComplexHeatmap** package, the `HeatmapAnnotation` object is always subsettable.

```
ha_subset = ha[1:5, c("foo", "PT")]
ha_subset
```

```
## A HeatmapAnnotation object with 2 annotations
##  name: heatmap_annotation_136
##  position: column
##  items: 5
##  width: 1npc
##  height: 15.3514598035146mm
##  this object is subsettable
```

```
##   5.21733333333mm extension on the left
##   6.75733333333mm extension on the right
##
##   name      annotation_type color_mapping height
##   foo       continuous vector      random     5mm
##   PT        anno_points()           10mm
```



The construction of heatmaps and annotations can be separated, later the annotations can be filled into the heatmap objects by the `attach_annotation()` function.

```
# code only for demonstration
ha1 = HeatmapAnnotation(foo = 1:10)
ha2 = rowAnnotation(bar = letters[1:10])
ht = Heatmap(mat)
ht = attach_annotation(ht, ha1, side = "top")
ht = attach_annotation(ht, ha2, side = "left")
```

3.23 Implement new annotation functions

All the annotation functions defined in `ComplexHeatmap` are constructed by the `AnnotationFunction` class. The `AnnotationFunction` class not only stores the “real R function” which draws the graphics, it also calculates the spaces produced by the annotation axis. More importantly, it allows splitting the annotation graphics according to the split of the main heatmap.

As expected, the main part of the `AnnotationFunction` class is a function which defines how to draw at specific positions which correspond to rows or columns in the heatmap. The function should have three arguments: `index`, `k` and `n` (the names of the arguments can be arbitrary) where `k` and `n` are optional. `index` corresponds to the indices of rows or columns of the heatmap. The value of `index` is not necessarily to be the whole row indices or column indices in the heatmap. It can also be a subset of the indices if the annotation is split into slices according to the split of the heatmap. `index` is reordered according to the reordering of heatmap rows or columns (e.g. by clustering). So, `index` actually contains a list of row or column indices for the current slice after row or column reordering.

As mentioned, annotation can be split into slices. `k` corresponds to the current slice and `n` corresponds to the total number of slices. The annotation function draws in every slice repeatedly. The information of `k` and `n` sometimes can be useful, for example, we want to add axis in the annotation, and if it is a column

annotation and axis is drawn on the very right of the annotation area, the axis is only drawn when `k == n`.

Since the function only allows `index`, `k` and `n`, the function sometimes uses several external variables which can not be defined inside the function, e.g. the data points for the annotation. These variables should be imported into the `AnnotationFunction` class by `var_import` so that the function can correctly find these variables.

One important feature for `AnnotationFunction` class is it can be subsettable, which is the base for splitting. To allow subsetting on the object, users need to define the rule for the imported variables if there is any. The rules are simple functions which accept the variable and indices, and return the subset of the variable. The subset rule functions implemented in this package are `subset_gp()`, `subset_matrix_by_row()` and `subset_vector()`. If the subsetting rule is not provided, it is inferred by the type of the object.

In the following example, we first construct an `AnnotationFunction` object which needs external variable and supports subsetting. This annotation contains a list of “lollipops” (points plus vertical segments) which is drawn in a viewport. Y-axis is drawn in the first slices.

The variable `x` is from outside of the function, so it should be added to `var_import`.

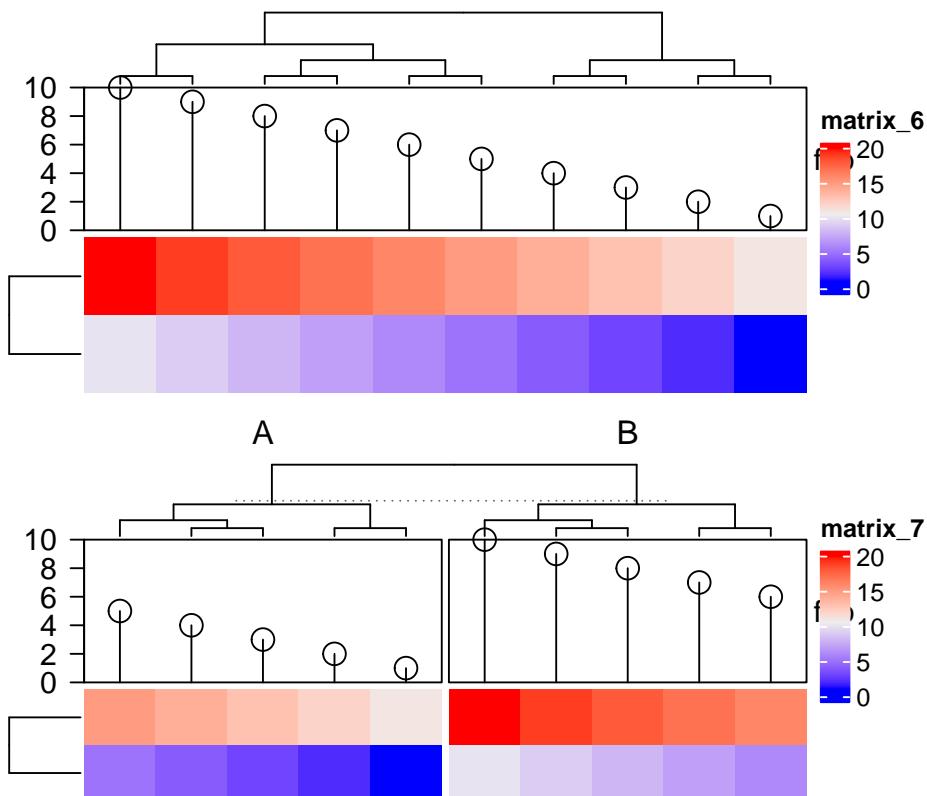
```
x = 1:10
anno1 = AnnotationFunction(
  fun = function(index, k, n) {
    n = length(index)
    pushViewport(viewport(xscale = c(0.5, n + 0.5), yscale = c(0, 10)))
    grid.rect()
    grid.points(1:n, x[index], default.units = "native")
    grid.segments(1:n, 0, 1:n, x[index], default.units = "native")
    if(k == 1) grid.yaxis()
    popViewport()
  },
  var_import = list(x = x),
  n = 10,
  subsettable = TRUE,
  height = unit(2, "cm")
)
anno1

## An AnnotationFunction object
##   function: user-defined
##   position: column
##   items: 10
##   width: 1npc
##   height: 2cm
```

```
##   imported variable: x
##   this object is subsettable
```

Then we can assign `anno1` in `HeatmapAnnotation()` function. Since `anno1` is subsettable, you can split columns of the heatmap.

```
m = rbind(1:10, 11:20)
Heatmap(m, top_annotation = HeatmapAnnotation(foo = anno1))
Heatmap(m, top_annotation = HeatmapAnnotation(foo = anno1),
        column_split = rep(c("A", "B"), each = 5))
```



The second way is to put all data variables inside the function and no need to import other variables.

```
# code only for demonstration
anno2 = AnnotationFunction(
  fun = function(index) {
    x = 1:10
    n = length(index)
    pushViewport(viewport())
    grid.points(1:n, x[index])
```

```

        popViewport()
    },
    n = 10,
    subsettable = TRUE
)

```

The most compact way to only specify the function to the constructor. This allows reordering, but it does not work when you split heatmap.

```

# code only for demonstration
anno3 = AnnotationFunction(
  fun = function(index) {
    x = 1:10
    n = length(index)
    pushViewport(viewport())
    grid.points(1:n, x[index])
    popViewport()
  }
)

```

All the `anno_*`() functions introduced in this section actually are not really annotation functions, while they are functions generating annotation functions with specific configurations.

```

anno_points(1:10)

## An AnnotationFunction object
##   function: anno_points()
##   position: column
##   items: 10
##   width: 1npc
##   height: 1cm
##   imported variable: data_scale, axis_param, border, size, value, pch_as_image, axis
##   subsettable variable: gp, value, size, pch
##   this object is subsettable
##   5.138311111111mm extension on the left

```

In most cases, you don't need to manually construct your `AnnotationFunction` objects. The annotation function `anno_*`() implemented in `ComplexHeatmap` are already enough for most of the analysis tasks. On the other hand, users can also use `anno_empty()` and `decorate_annotation()` to quickly add self-defined annotation graphics. E.g. we can re-implement previous heatmap as (of course it is lengthy):

```

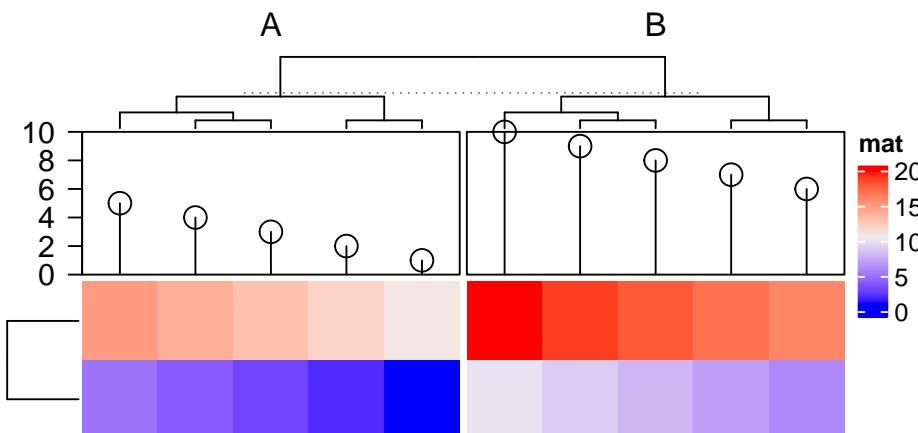
ht = Heatmap(m, name = "mat",
  top_annotation = HeatmapAnnotation(foo = anno_empty(height = unit(2, "cm"))),
  column_split = rep(c("A", "B"), each = 5))
ht = draw(ht)

```

```

co = column_order(ht)
decorate_annotation("foo", slice = 1, {
  od = co[[1]]
  pushViewport(viewport(xscale = c(0.5, length(od) + 0.5), yscale = c(0, 10)))
  grid.points(seq_along(od), x[od])
  grid.segments(seq_along(od), 0, seq_along(od), x[od], default.units = "native")
  grid.yaxis()
  popViewport()
})
decorate_annotation("foo", slice = 2, {
  od = co[[2]]
  pushViewport(viewport(xscale = c(0.5, length(od) + 0.5), yscale = c(0, 10)))
  grid.points(seq_along(od), x[od])
  grid.segments(seq_along(od), 0, seq_along(od), x[od], default.units = "native")
  popViewport()
})

```



3.23.1 Construct annotation function by `cell_fun`

To simplify the use of `AnnotationFunction()`, from version 2.9.3, it has a new argument `cell_fun` which accepts a self-defined function that only draws in a single “annotation cell.” This would be convenient for annotation only with a few cells. See the following example which visualizes percent values by text and bars. Actually this is how `anno_numeric()` (Section 3.15) is implemented.

```

anno_pct = function(x) {

  max_x = max(x)
  text = paste0(sprintf("%.2f", x*100), "%")
  cell_fun_pct = function(i) {

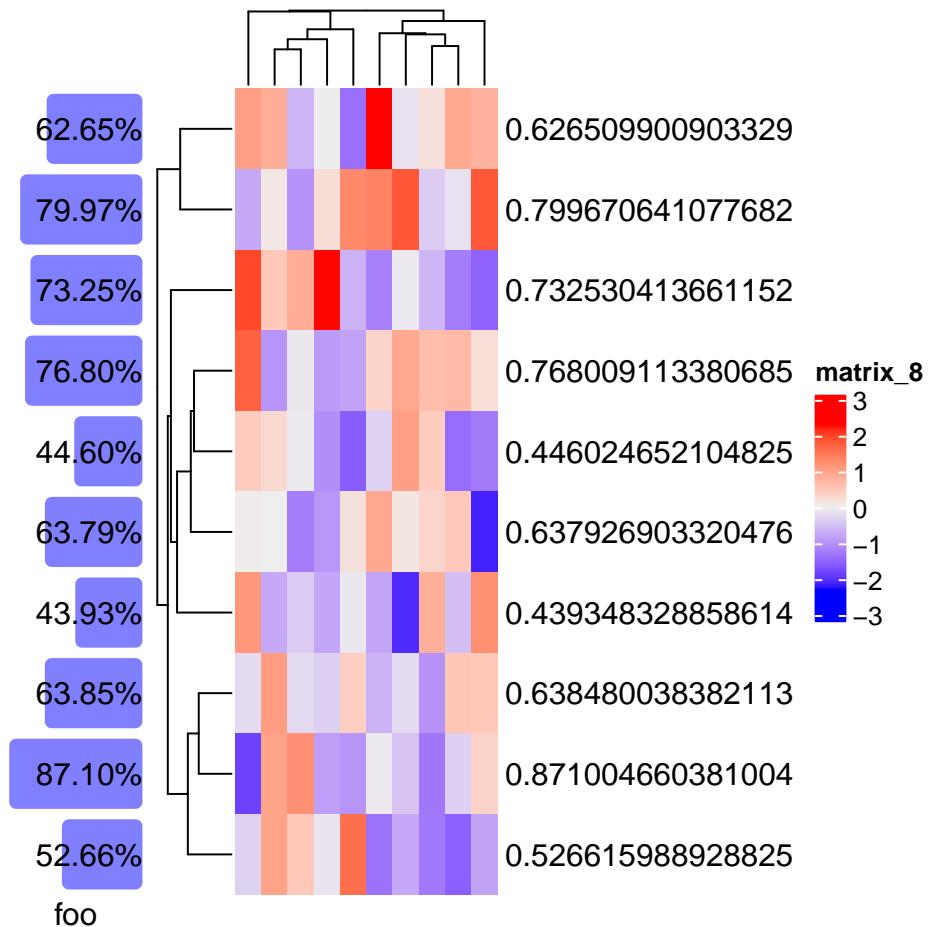
```

```
pushViewport(viewport(xscale = c(0, max_x)))
grid.roundrect(x = unit(1, "npc"), width = unit(x[i], "native"),
               height = unit(1, "npc") - unit(4, "pt"),
               just = "right", gp = gpar(fill = "#0000FF80", col = NA))
grid.text(text[i], x = unit(1, "npc"), just = "right")
popViewport()
}

AnnotationFunction(
  cell_fun = cell_fun_pct,
  var_import = list(max_x, x, text),
  which = "row",
  width = max_text_width(text)*1.25
)
}

x = runif(10)
ha = rowAnnotation(foo = anno_pct(x), annotation_name_rot = 0)

m = matrix(rnorm(100), 10)
rownames(m) = x
ha + Heatmap(m)
```



Note if `cell_fun` is set in `AnnotationFunction`, the returned annotation function is subsettable, which means it also works when heatmap is split.

Chapter 4

A List of Heatmaps

The main feature of **ComplexHeatmap** package is it supports to concatenate a list of heatmaps and annotations horizontally or vertically so that it makes it possible to visualize the associations from various sources of information. In this chapter, we mainly introduce the horizontal concatenation because this is the major case we will use in the analysis. In the end we show some examples of vertical concatenation. The concept behind for horizontal and vertical concatenation basically is similar.

For the horizontal concatenation, the number of rows for all heatmaps and annotations should be the same. In following we first introduce the concatenation of heatmaps and later we will show how to concatenate heatmaps with annotations.

In following example, there are three matrices where the third heatmap is a vector and it will be transformed as a one-column matrix. The one-column heatmap is sometimes useful when you concatenate a list of heatmaps that it can show e.g. annotations for each row or some scores of each row. E.g. if rows are genes, the type of the genes (i.e. protein coding or not) can be represented as a one-column character matrix, and the p-value or the fold change from differential expression analysis can be represented as a one-column numeric matrix, and be concatenated to the main expression heatmap.

```
set.seed(123)
mat1 = matrix(rnorm(80, 2), 8, 10)
mat1 = rbind(mat1, matrix(rnorm(40, -2), 4, 10))
rownames(mat1) = paste0("R", 1:12)
colnames(mat1) = paste0("C", 1:10)

mat2 = matrix(runif(60, max = 3, min = 1), 6, 10)
mat2 = rbind(mat2, matrix(runif(60, max = 2, min = 0), 6, 10))
rownames(mat2) = paste0("R", 1:12)
colnames(mat2) = paste0("C", 1:10)
```

```

le = sample(letters[1:3], 12, replace = TRUE)
names(le) = paste0("R", 1:12)

ind = sample(12, 12)
mat1 = mat1[ind, ]
mat2 = mat2[ind, ]
le = le[ind]

```

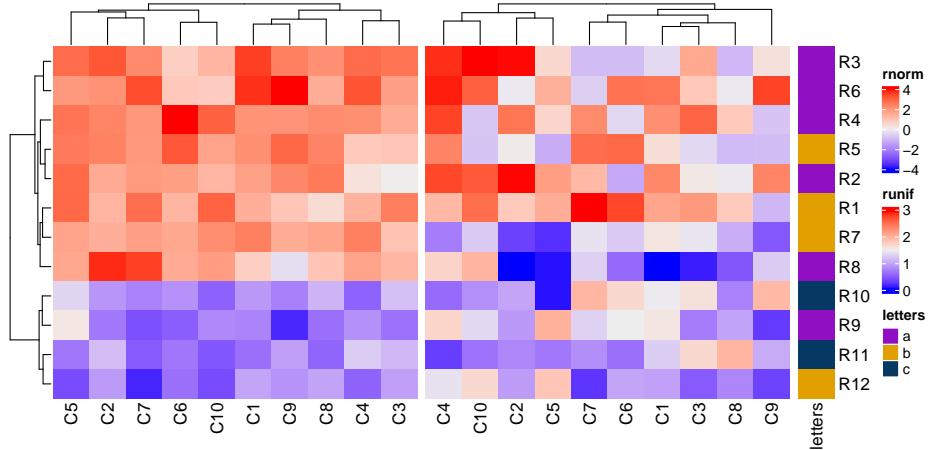
To concatenate heatmaps, simply use + operator.

```

ht1 = Heatmap(mat1, name = "rnorm")
ht2 = Heatmap(mat2, name = "runif")
ht3 = Heatmap(le, name = "letters")

ht1 + ht2 + ht3

```



Under default mode, dendograms from the second heatmap will be removed and row orders will be the same as the first one. Also row names for the first two heatmaps are removed as well.

The returned value of the concatenation is a `HeatmapList` object. Similar as explained in Section 2.11, directly printing `ht_list` will call `draw()` method with default settings. With explicitly calling `draw()` method, you can have more controls on the heatmap list.

```

ht_list = ht1 + ht2 + ht3
class(ht_list)

```

```

## [1] "HeatmapList"
## attr(,"package")
## [1] "ComplexHeatmap"

```

You can append any number of heatmaps to the heatmap list. Also you can append a heatmap list to a heatmap list.

```
# code only for demonstration
ht1 + ht_list
ht_list + ht1
ht_list + ht_list
```

`NULL` can be added to the heatmap list. It would be convenient when users want to construct a heatmap list through a `for` loop.

```
# code only for demonstration
ht_list = NULL ## Heatmap(...) + NULL gives you a HeatmapList object
for(s in sth) {
  ht_list = ht_list + Heatmap(...)
}
```

You can also add heatmap annotations to the heatmap list, see more details in Section 4.7.

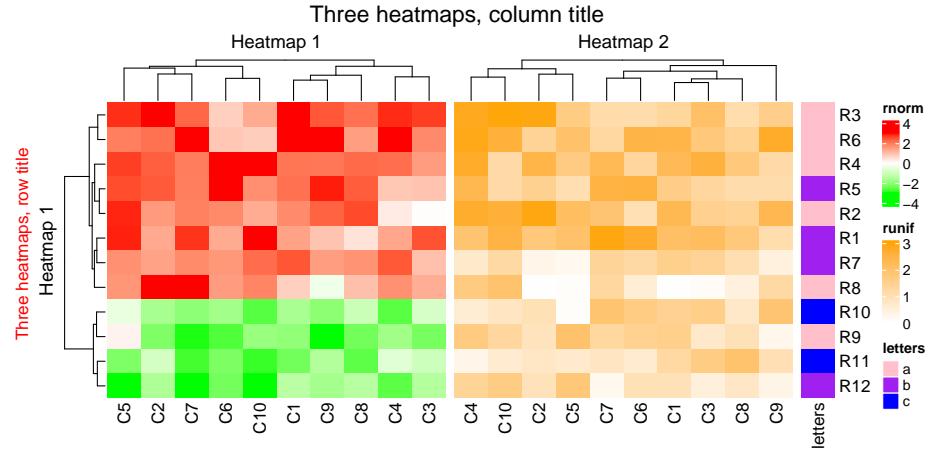
4.1 Titles

A heatmap list also has title which is like a global title covering all heatmaps. `row_title` and `column_title` should be set in the `draw()` function.

From following example, we set different colors for each heatmap to make them distinguishable.

```
col_rnorm = colorRamp2(c(-3, 0, 3), c("green", "white", "red"))
col_runif = colorRamp2(c(0, 3), c("white", "orange"))
col_letters = c("a" = "pink", "b" = "purple", "c" = "blue")
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm,
  row_title = "Heatmap 1", column_title = "Heatmap 1")
ht2 = Heatmap(mat2, name = "runif", col = col_runif,
  row_title = "Heatmap 2", column_title = "Heatmap 2")
ht3 = Heatmap(le, name = "letters", col = col_letters)
ht_list = ht1 + ht2 + ht3

draw(ht_list, row_title = "Three heatmaps, row title", row_title_gp = gpar(col = "red"),
  column_title = "Three heatmaps, column title", column_title_gp = gpar(fontsize = 16))
```

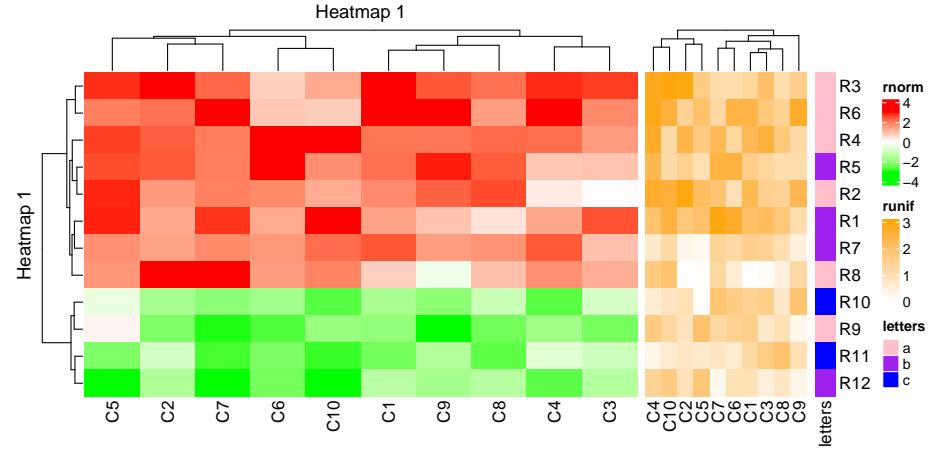


You can use `gt_render()` to construct complicated text, see Section 10.3.

4.2 Size of heatmaps

The width for some heatmaps can be set to absolute units. Note `width` controls the width of the heatmap body.

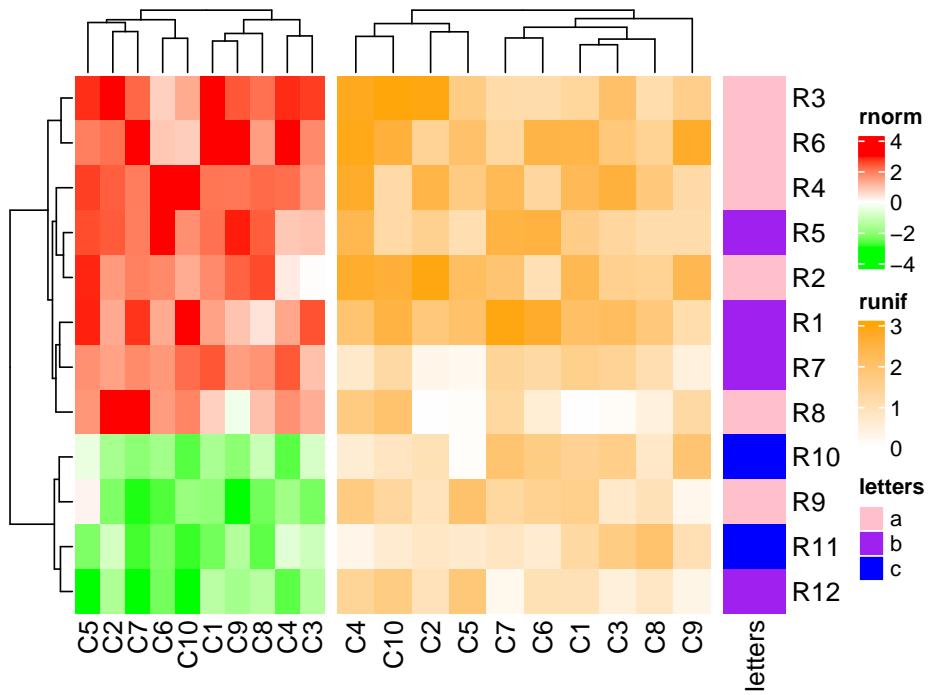
```
ht2 = Heatmap(mat2, name = "runif", col = col_runif, width = unit(4, "cm"))
ht3 = Heatmap(le, name = "letters", col = col_letters, width = unit(5, "mm"))
ht1 + ht2 + ht3
```



The width of all heatmaps can be set as absolute units.

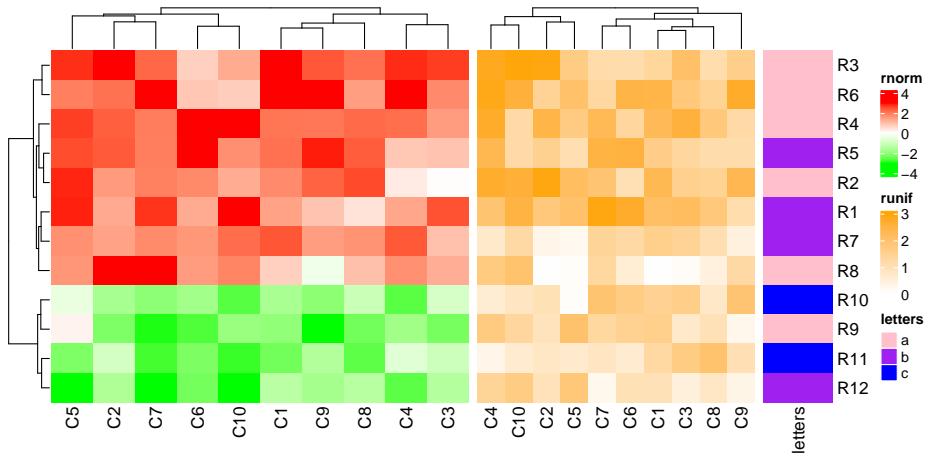
```
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm, width = unit(4, "cm"))
ht2 = Heatmap(mat2, name = "runif", col = col_runif, width = unit(6, "cm"))
ht3 = Heatmap(le, name = "letters", col = col_letters, width = unit(1, "cm"))
```

```
ht1 + ht2 + ht3
```



If width is numeric, it is converted as a null unit.

```
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm, width = 6)
ht2 = Heatmap(mat2, name = "runif", col = col_runif, width = 4)
ht3 = Heatmap(le, name = "letters", col = col_letters, width = 1)
ht1 + ht2 + ht3
```

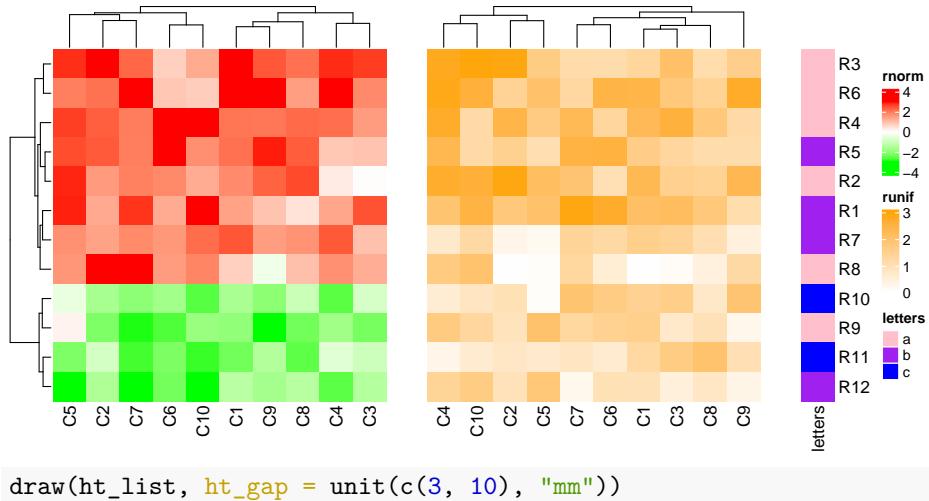


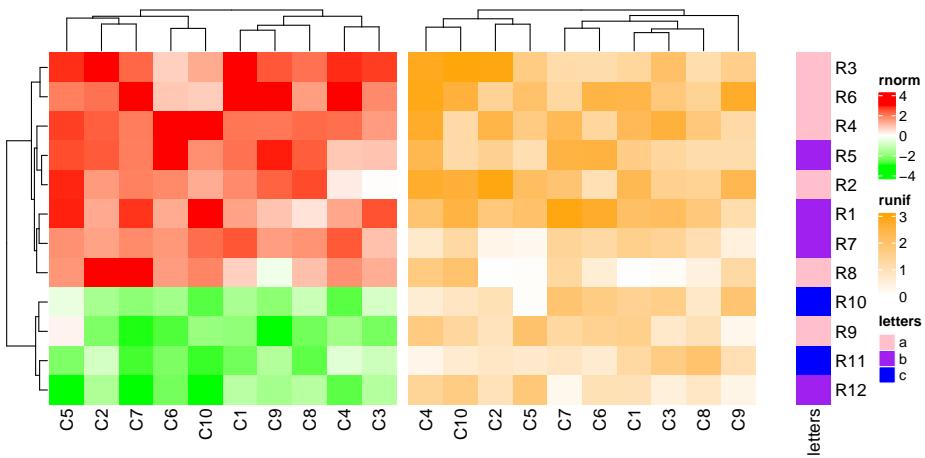
`heatmap_width` also can control the width of the heatmap, but it is the total width of the heatmap body plus the heatmap components.

4.3 Gap between heatmaps

`ht_gap` controls the space between heatmaps. The value can be a single unit or a vector of units.

```
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm)
ht2 = Heatmap(mat2, name = "runif", col = col_runif)
ht3 = Heatmap(le, name = "letters", col = col_letters)
ht_list = ht1 + ht2 + ht3
draw(ht_list, ht_gap = unit(1, "cm"))
```





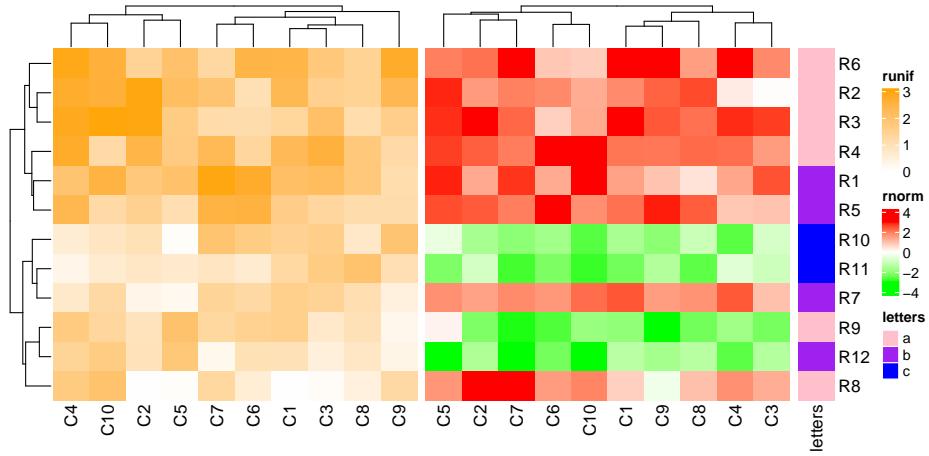
4.4 Automatic adjustment to the main heatmap

There is always a main heatmap in the heatmap list that controls the global row ordering. All the other heatmaps are automatically adjusted according to the settings in the main heatmap. For these non-main heatmaps, the adjustments are:

- No row clustering is performed and they all take the row ordering of the main heatmap.
- Row titles are removed.
- If the main heatmap is split by rows, all other heatmaps will also be split by same levels as the main heatmap.
- The height of the main heatmap are taken as the height of all heatmaps.

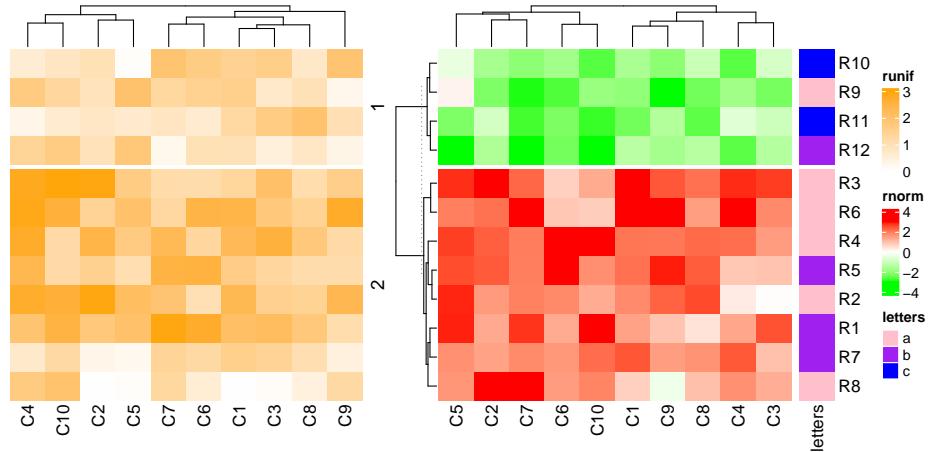
By default, the first heatmap is taken as the main heatmap.

```
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm, row_km = 2)
ht2 = Heatmap(mat2, name = "runif", col = col_runif)
ht3 = Heatmap(le, name = "letters", col = col_letters)
ht2 + ht1 + ht3 # ht2 is the main heatmap and row_km in ht1 is ignored
```



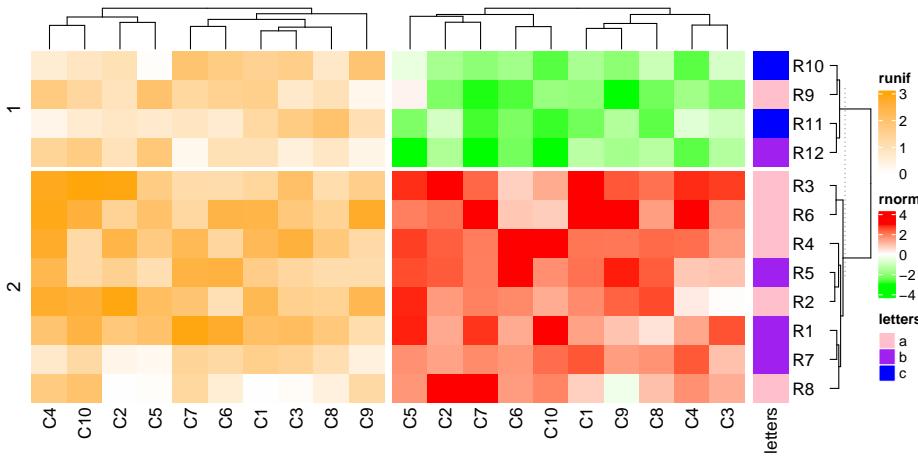
The main heatmap can be specified by `main_heatmap` argument. The value can be a numeric index or the name of the heatmap (of course, you need to set the heatmap name when you create the `Heatmap` object). In following example, although `ht1` is the second heatmap, we can set it as the main heatmap.

```
ht_list = ht2 + ht1 + ht3
draw(ht_list, main_heatmap = "rnorm")
```



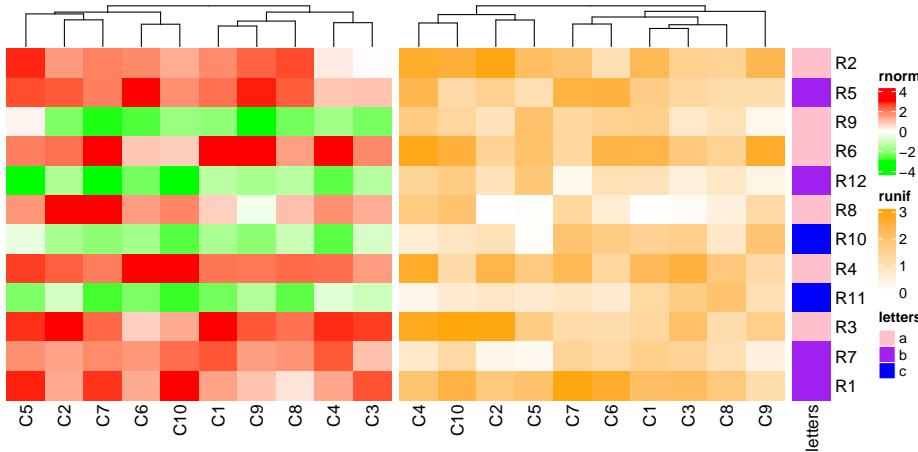
By default, the dendrogram and the row title are plotted just beside the main heatmap, just to emphasize the clustering or the splitting is calculated from the main heatmap while not other heatmaps. However, the position of the dendrogram and row title of the main heatmap can be controlled by `row_dend_side` and `row_sub_title_side` in `draw()` function.

```
ht_list = ht2 + ht1 + ht3
draw(ht_list, main_heatmap = "rnorm", row_dend_side = "right", row_sub_title_side = "left")
```



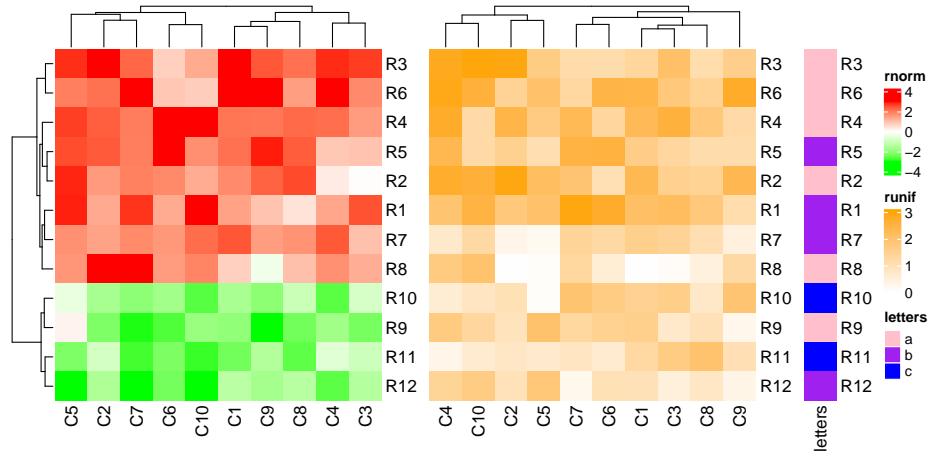
Similarly, if there is no row clustering in the main heatmap, all other heatmaps are not clustered neither.

```
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm, cluster_rows = FALSE)
ht2 = Heatmap(mat2, name = "runif", col = col_runif)
ht3 = Heatmap(le, name = "letters", col = col_letters)
ht1 + ht2 + ht3
```



As you may have observed, all the row names between heatmaps are removed from the plot. You can show them by setting `auto_adjust = FALSE`.

```
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm)
ht2 = Heatmap(mat2, name = "runif", col = col_runif)
ht3 = Heatmap(le, name = "letters", col = col_letters)
ht_list = ht1 + ht2 + ht3
draw(ht_list, auto_adjust = FALSE)
```



4.5 Control main heatmap in `draw()` function

Settings of the main heatmap can be controlled in the main `Heatmap()` function. To make it convenient, settings that affect heatmap rows can also be directly set in `draw()`. If some of these settings are set, corresponding settings in the main `Heatmap()` will be overwritten.

In `draw()` function, following main heatmap settings control row orders of all heatmaps.

- `cluster_rows`
- `clustering_distance_rows`
- `clustering_method_rows`
- `row_dend_width`
- `show_row_dend`
- `row_dend_reorder`
- `row_dend_gp`
- `row_order`

Following settings control the row slices.

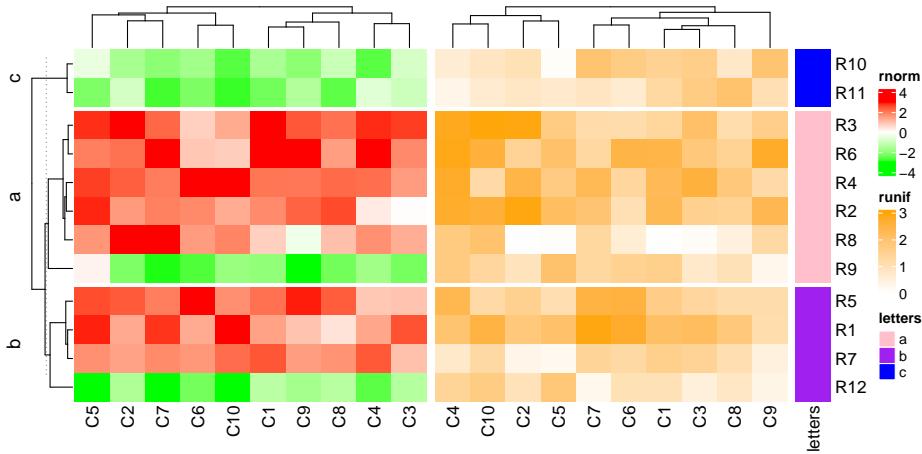
- `row_gap`
- `row_km`
- `row_km_repeats`
- `row_split`

Following settings control the heatmap height.

- `height`
- `heatmap_height`

In following example, `row_km = 2`, `cluster_rows = FALSE` for `ht1` is overwritten in `draw()`.

```
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm, row_km = 2, cluster_rows = FALSE)
ht2 = Heatmap(mat2, name = "runif", col = col_runif)
ht3 = Heatmap(le, name = "letters", col = col_letters)
ht_list = ht1 + ht2 + ht3
draw(ht_list, row_km = 1, row_split = le, cluster_rows = TRUE)
```

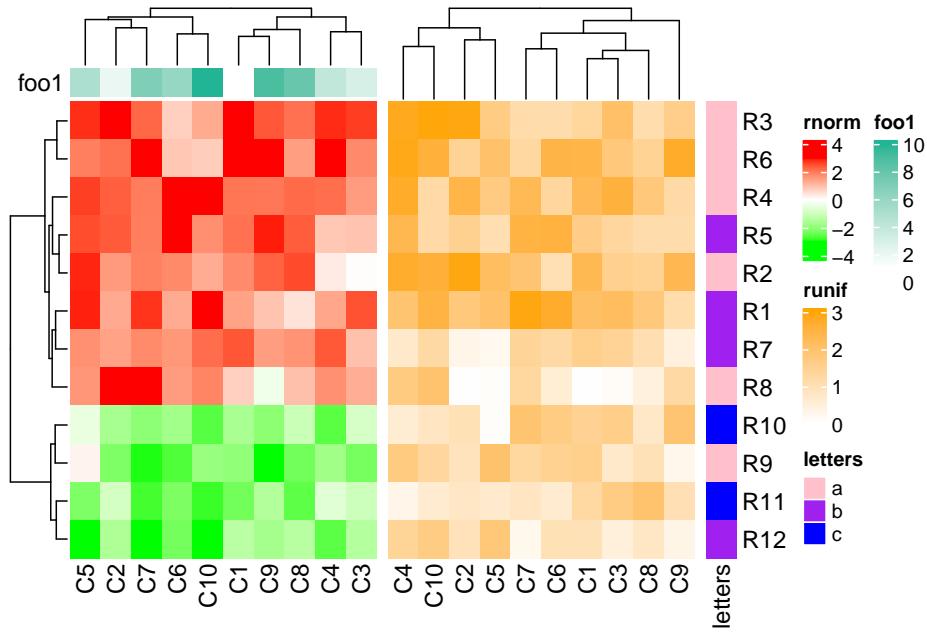


4.6 Annotations as components are adjusted

If some of the heatmaps in the heatmap list have annotations, in most of the cases, the heights of the heatmap annotations are different for different heatmaps. There are automatic adjustment for heatmap annotations, and this adjustment will also involve adjustment of dendograms.

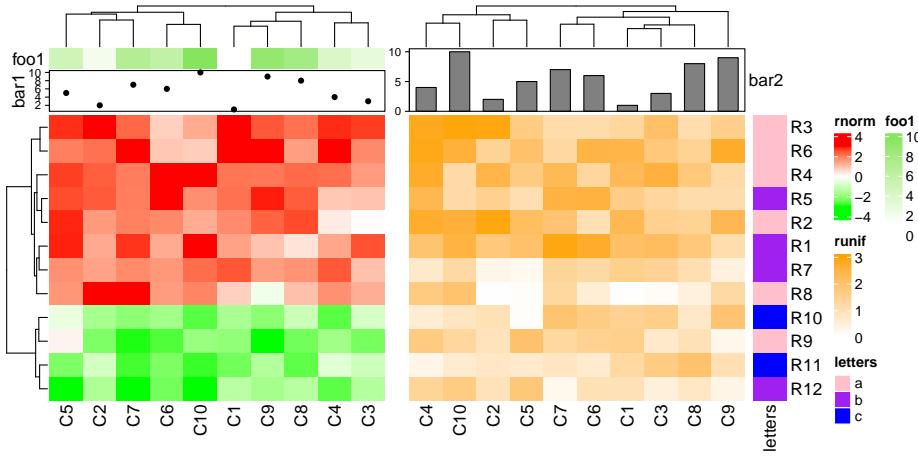
Normally, the size of simple annotations will not change in the adjustment. In following example, the dendrogram for the second heatmap is adjusted. Note you still can change the size of simple annotation by setting `anno_simple_size` in `HeatmapAnnotation()` or globally set `ht_opt$anno_simple_size`.

```
ha1 = HeatmapAnnotation(fool = 1:10, annotation_name_side = "left")
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm, top_annotation = ha1)
ht2 = Heatmap(mat2, name = "runif", col = col_runif)
ht3 = Heatmap(le, name = "letters", col = col_letters)
ht1 + ht2 + ht3
```



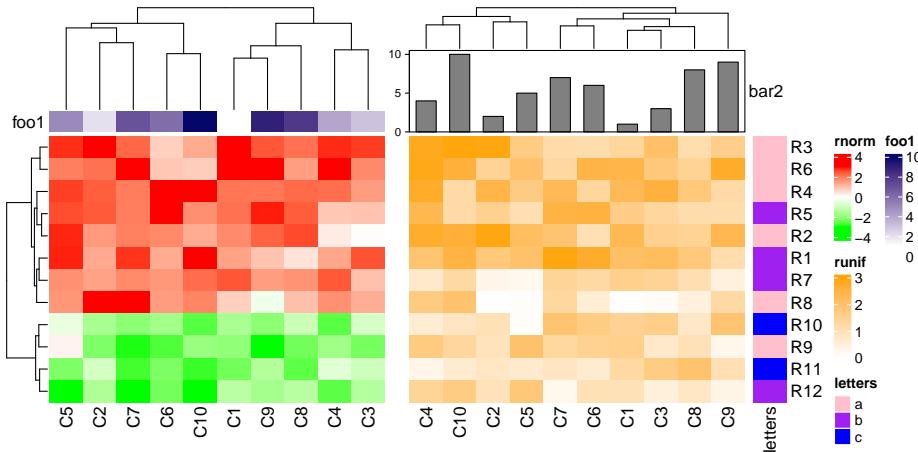
If the first two heatmaps all have annotations, since the size of simple annotations keep unchanged, the size of complex annotations will be adjusted to make the total heights of the two heatmap annotations the same.

```
ha1 = HeatmapAnnotation(foo1 = 1:10, bar1 = anno_points(1:10),
annotation_name_side = "left")
ha2 = HeatmapAnnotation(bar2 = anno_barplot(1:10))
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm, top_annotation = ha1)
ht2 = Heatmap(mat2, name = "runif", col = col_runif, top_annotation = ha2)
ht3 = Heatmap(le, name = "letters", col = col_letters)
ht_list = ht1 + ht2 + ht3
draw(ht_list, ht_gap = unit(c(6, 2), "mm"))
```



Similarly, if the first heatmap only contains simple annotations, dendrogram will be adjusted.

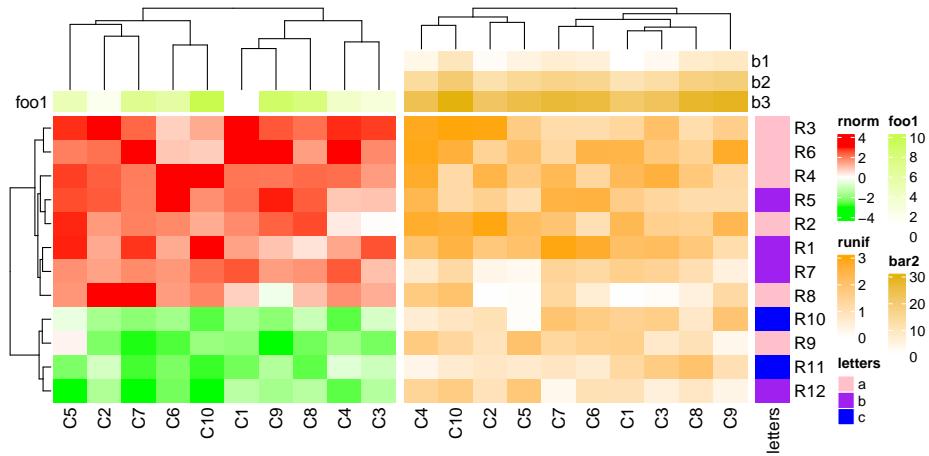
```
ha1 = HeatmapAnnotation(foo1 = 1:10, annotation_name_side = "left")
ha2 = HeatmapAnnotation(bar2 = anno_barplot(1:10, height = unit(2, "cm")))
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm, top_annotation = ha1)
ht2 = Heatmap(mat2, name = "runif", col = col_runif, top_annotation = ha2)
ht3 = Heatmap(le, name = "letters", col = col_letters)
ht_list = ht1 + ht2 + ht3
draw(ht_list, ht_gap = unit(c(6, 2), "mm"))
```



If the both heatmaps only contain simple annotations but with unequal number, dendrogram will be adjusted.

```
ha1 = HeatmapAnnotation(foo1 = 1:10, annotation_name_side = "left")
ha2 = HeatmapAnnotation(bar2 = cbind(b1 = 1:10, b2 = 11:20, b3 = 21:30))
```

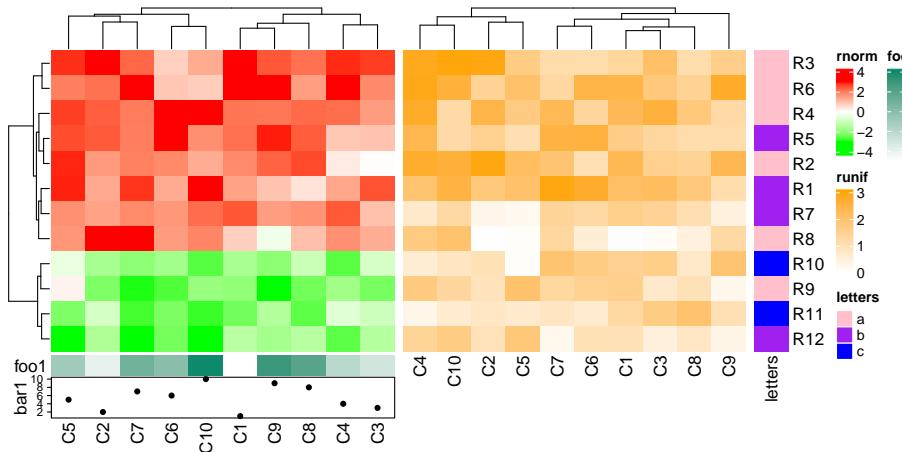
```
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm, top_annotation = ha1)
ht2 = Heatmap(mat2, name = "runif", col = col_runif, top_annotation = ha2)
ht3 = Heatmap(le, name = "letters", col = col_letters)
ht_list = ht1 + ht2 + ht3
draw(ht_list)
```



If you also want to automatically adjust the size of simple annotations, set `simple_anno_size_adjust = TRUE` in every `HeatmapAnnotation()` calls.

If the second heatmap has no bottom annotation, column names for the second heatmap are adjusted to be put directly below the heatmap body.

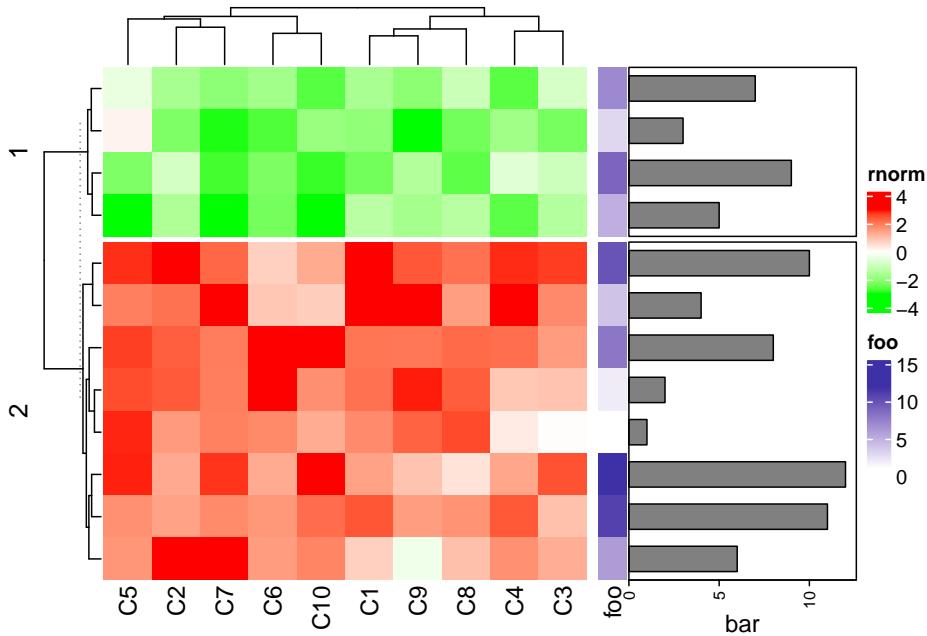
```
ha1 = HeatmapAnnotation(foo1 = 1:10, bar1 = anno_points(1:10), annotation_name_side = "bottom")
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm, bottom_annotation = ha1)
ht2 = Heatmap(mat2, name = "runif", col = col_runif)
ht3 = Heatmap(le, name = "letters", col = col_letters)
ht_list = ht1 + ht2 + ht3
draw(ht_list)
```



4.7 Concatenate with annotations

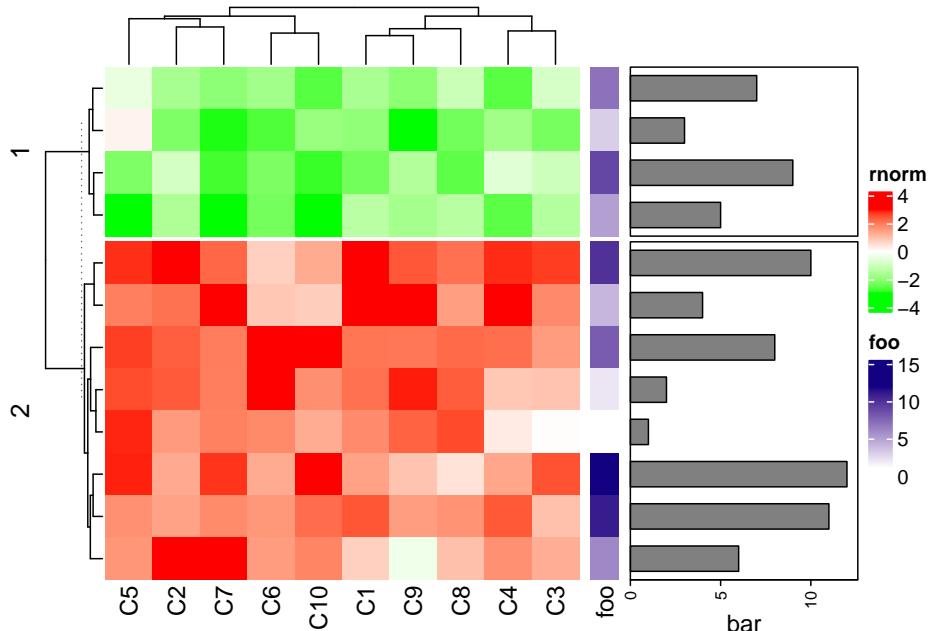
Row annotations can be concatenated to the horizontal heatmap list, while not only a component of the heatmap. See following examples which are very straightforward.

```
ha1 = rowAnnotation(foo = 1:12, bar = anno_barplot(1:12, width = unit(4, "cm")))
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm, row_km = 2)
ht1 + ha1
```



The `foo` and `bar` annotations can be defined in two separated `rowAnnotation()` calls.

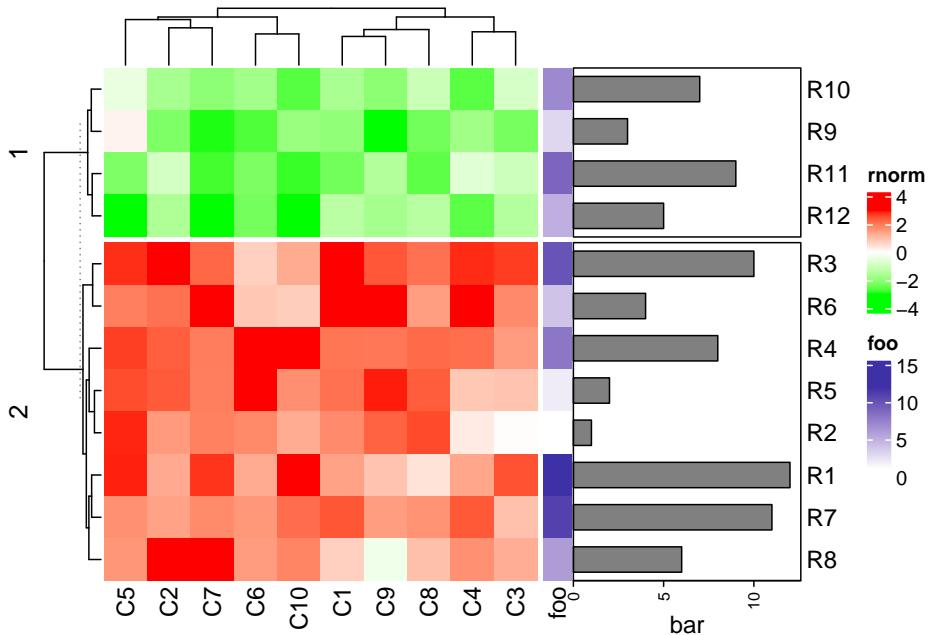
```
Heatmap(mat1, name = "rnorm", col = col_rnorm, row_km = 2) +
  rowAnnotation(foo = 1:12) +
  rowAnnotation(bar = anno_barplot(1:12, width = unit(4, "cm")))
```



You may wonder how to recover the row names of `mat1`. There are two ways.

1. you can set the row annotation as the “right annotation” of the heatmap and put the heatmap as the last one.

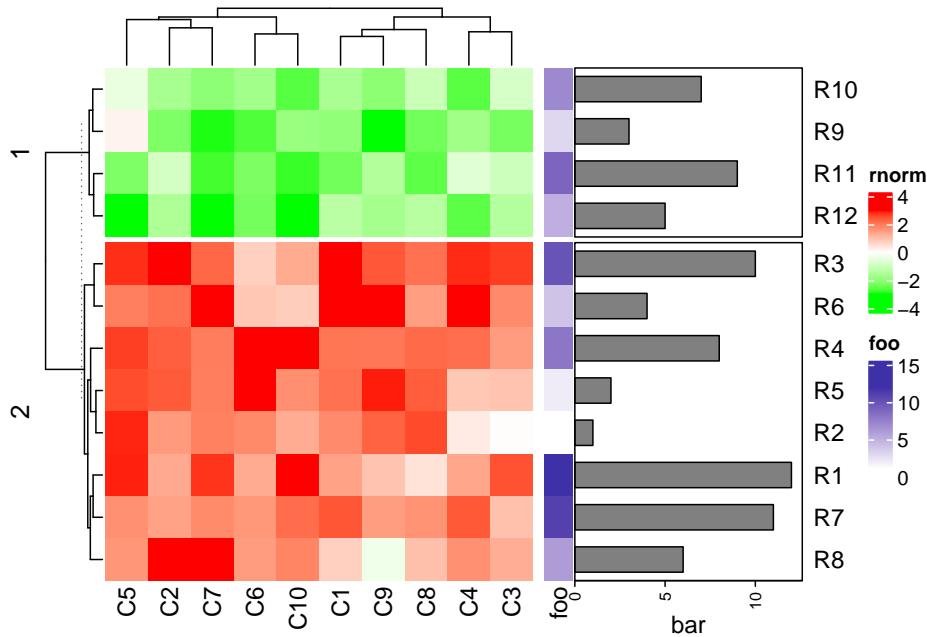
```
Heatmap(mat1, name = "rnorm", col = col_rnorm, row_km = 2, right_annotation = ha1)
```



```
# or using the previous variable
# attach_annotation(ht1, ha1, side = "right")
```

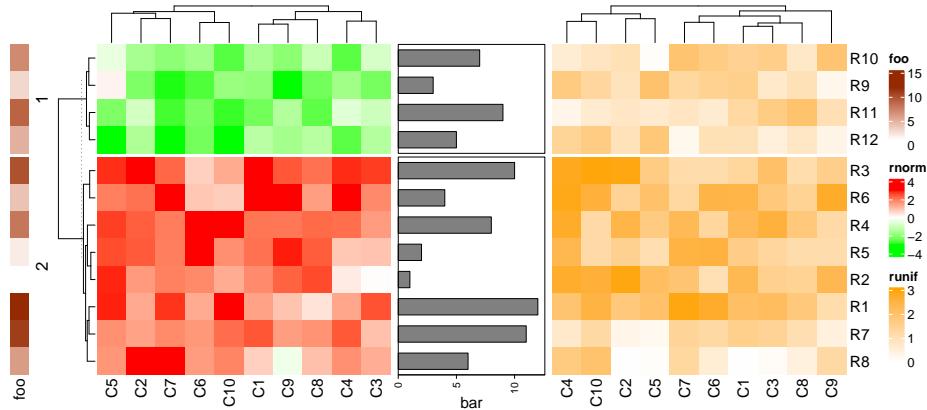
2. add the row names by adding a text annotation:

```
ht1 + ha1 + rowAnnotation(rn = anno_text(rownames(mat1),
  location = unit(0, "npc"), just = "left"))
```



Basically heatmaps and row annotations can be concatenated arbitrarily.

```
rowAnnotation(foo = 1:12) +
  Heatmap(mat1, name = "rnorm", col = col_rnorm, row_km = 2) +
  rowAnnotation(bar = anno_barplot(1:12, width = unit(4, "cm"))) +
  Heatmap(mat2, name = "runif", col = col_runif)
```

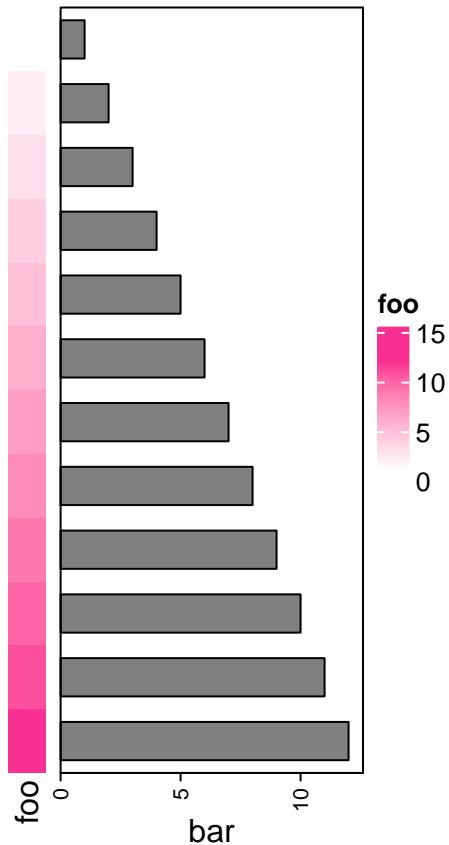


As mentioned in previous chapters, row annotations can also be heatmap components as left annotations or right annotations. The difference of row annotations as independent ones and as heatmap components is discussed in Section 4.9.

4.8 Concatenate only the annotations

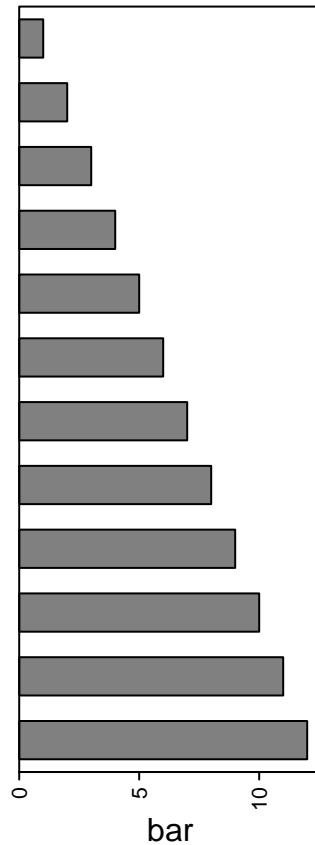
The concatenation can be done without any heatmap.

```
rowAnnotation(foo = 1:12) +
  rowAnnotation(bar = anno_barplot(1:12, width = unit(4, "cm")))
```



If there is only one `HeatmapAnnotation` object, you must concatenated with `NULL`.

```
rowAnnotation(bar = anno_barplot(1:12, width = unit(4, "cm")))) + NULL
```



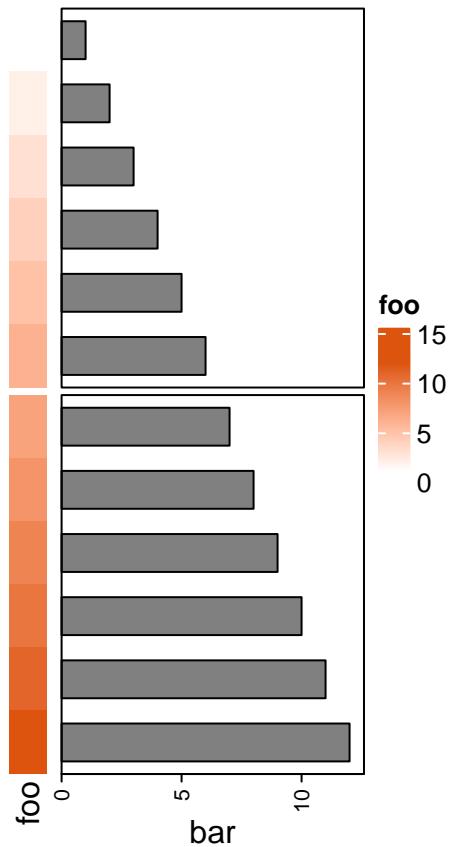
The annotation list is actually also a `HeatmapList` object.

```
anno_list = rowAnnotation(foo = 1:12) +
  rowAnnotation(bar = anno_barplot(1:12, width = unit(4, "cm")))
class(anno_list)
```

```
## [1] "HeatmapList"
## attr(,"package")
## [1] "ComplexHeatmap"
```

Thus, you can use some functionalities of the `draw()` function for the annotation list, such as row splitting.

```
draw(anno_list, row_split = rep(c("A", "B"), each = 6))
```

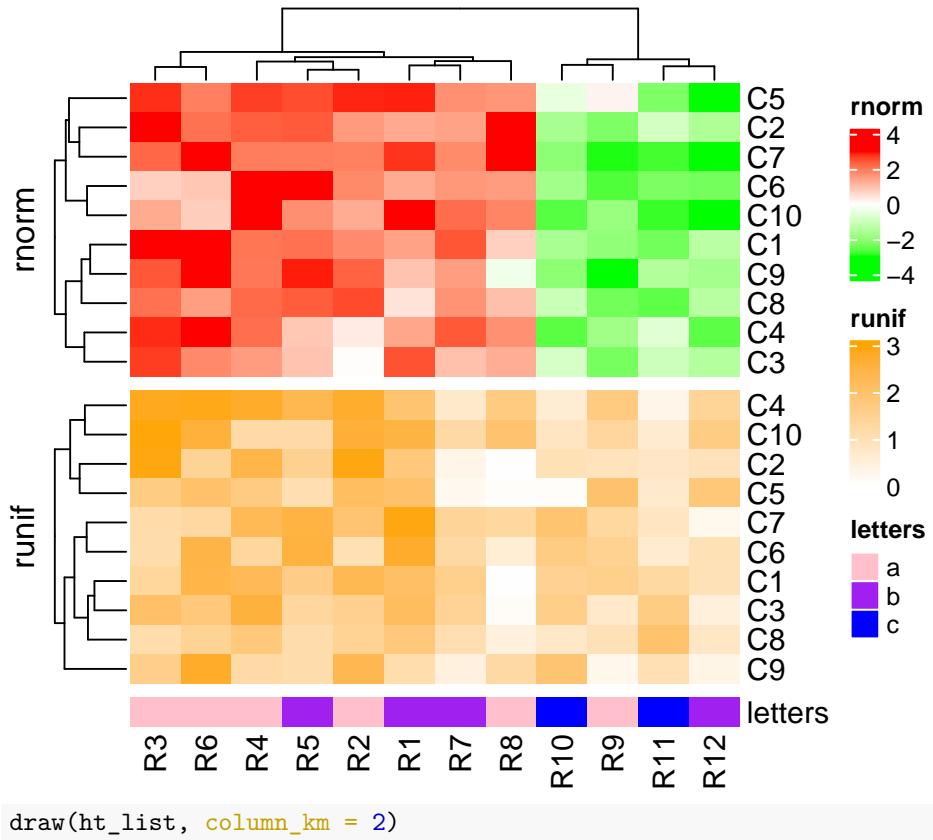


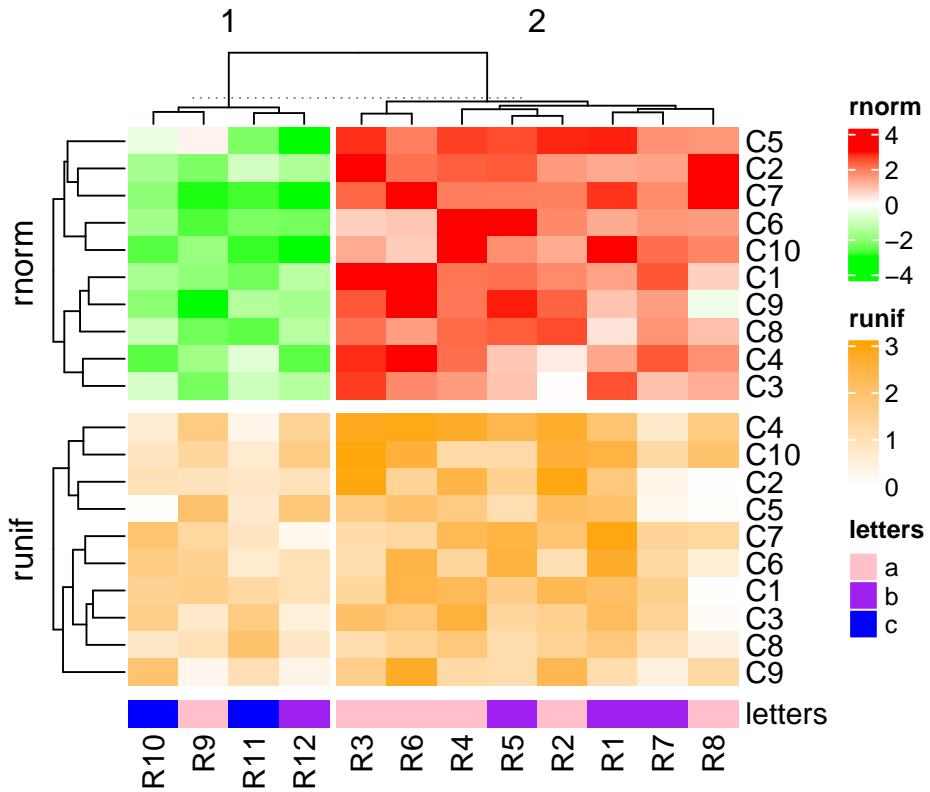
In Section 11.2, we will show how to use a list of annotations to visualize multiple summary statistics.

4.9 Vertical concatenation

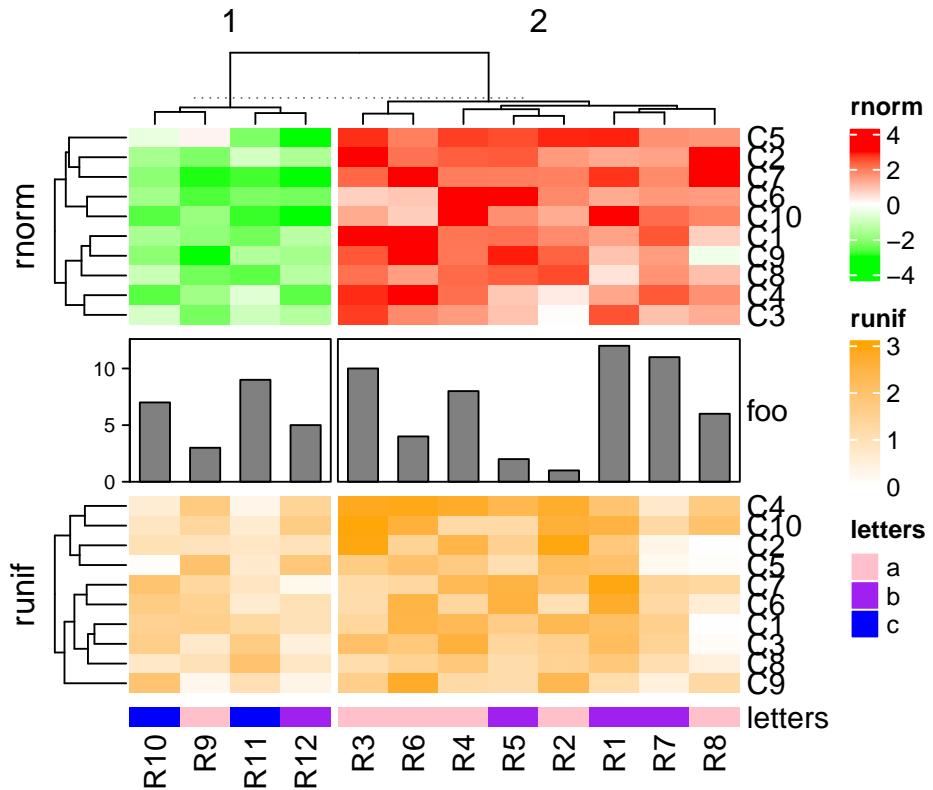
Heatmaps and annotations (now it is column annotation) can be concatenated vertically by the `%v%` operator. All the related settings and adjustments are very similar as the horizontal concatenation. Please check following examples.

```
mat1t = t(mat1)
mat2t = t(mat2)
ht1 = Heatmap(mat1t, name = "rnorm", col = col_rnorm, row_title = "rnorm")
ht2 = Heatmap(mat2t, name = "runif", col = col_runif, row_title = "runif")
ht3 = Heatmap(rbind(letters = le), name = "letters", col = col_letters)
ht_list = ht1 %v% ht2 %v% ht3
draw(ht_list)
```





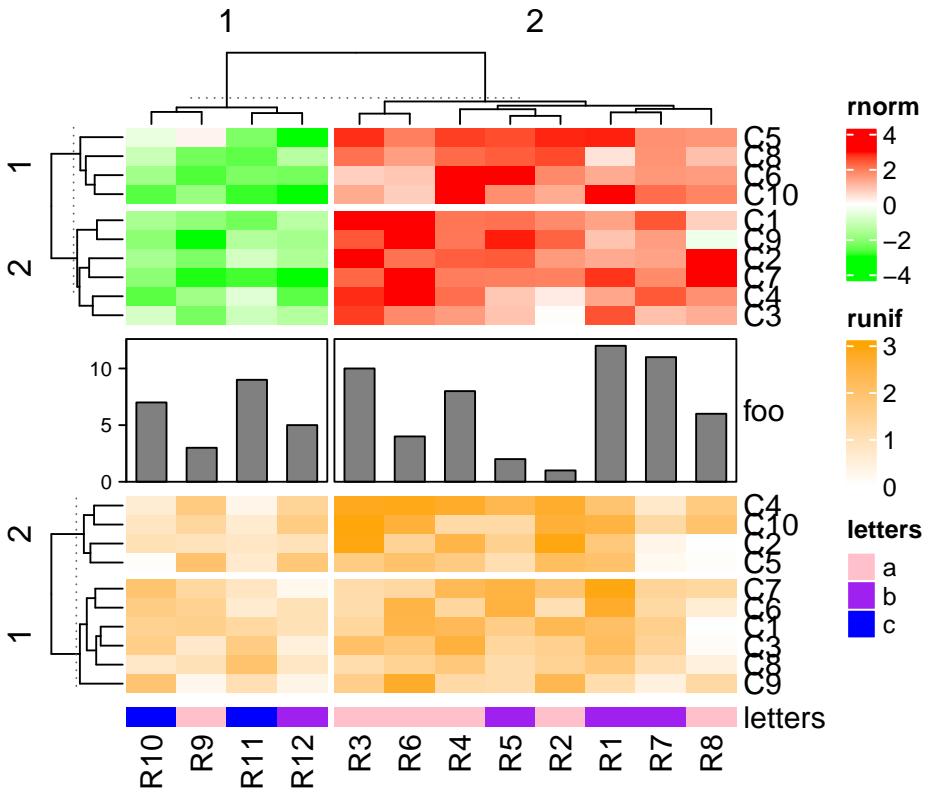
```
ha = HeatmapAnnotation(foo = anno_barplot(1:12, height = unit(2, "cm")))
ht_list = ht1 %v% ha %v% ht2 %v% ht3
draw(ht_list, column_km = 2)
```



```

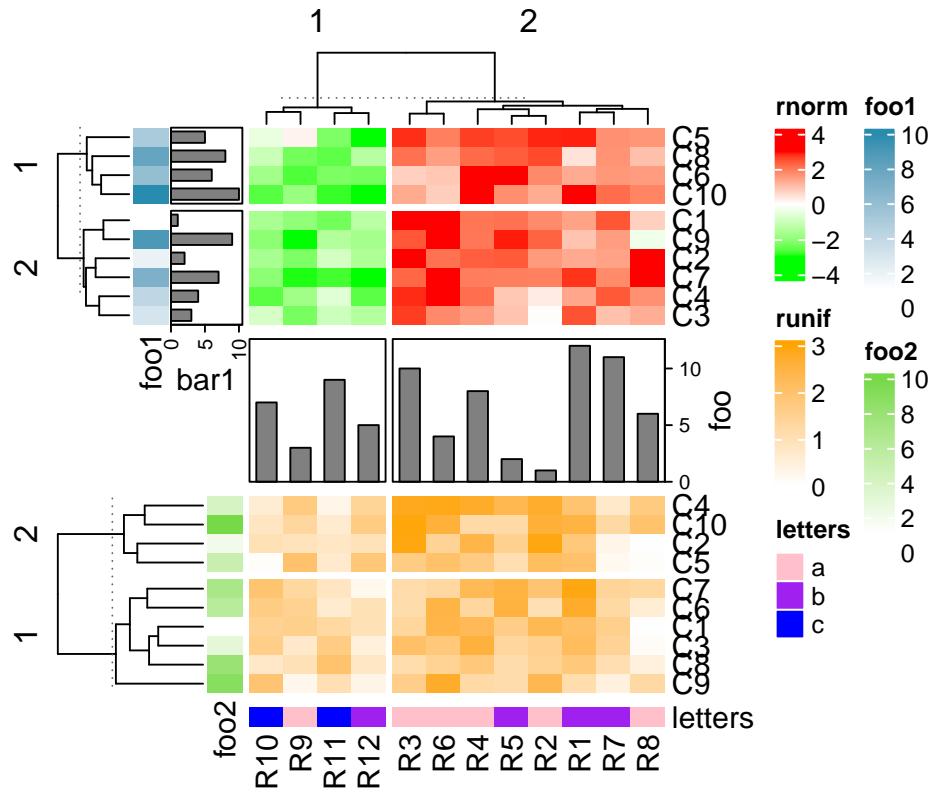
ht1 = Heatmap(mat1t, name = "rnorm", col = col_rnorm, row_km = 2)
ht2 = Heatmap(mat2t, name = "runif", col = col_runif, row_km = 2)
ht3 = Heatmap(rbind(letters = le), name = "letters", col = col_letters)
ha = HeatmapAnnotation(foo = anno_barplot(1:12, height = unit(2, "cm")))
ht_list = ht1 %v% ha %v% ht2 %v% ht3
draw(ht_list, column_km = 2)

```



For the vertical heatmap list, now row annotations should be the heatmap components (by `right_annotation` and `left_annotation`) and they are adjusted just like column annotations for the horizontal heatmap list.

```
ht1 = Heatmap(mat1t, name = "rnorm", col = col_rnorm, row_km = 2,
    left_annotation = rowAnnotation(foo1 = 1:10, bar1 = anno_barplot(1:10)))
ha = HeatmapAnnotation(foo = anno_barplot(1:12, height = unit(2, "cm"),
    axis_param = list(side = "right")))
ht2 = Heatmap(mat2t, name = "runif", col = col_runif, row_km = 2,
    left_annotation = rowAnnotation(foo2 = 1:10))
ht3 = Heatmap(rbind(letters = le), name = "letters", col = col_letters)
ht_list = ht1 %v% ha %v% ht2 %v% ht3
draw(ht_list, column_km = 2)
```



Since `rowAnnotation()` allows arbitrary number of annotations, the way showed above is the only way to expand the heatmap list horizontally and vertically at the same time.

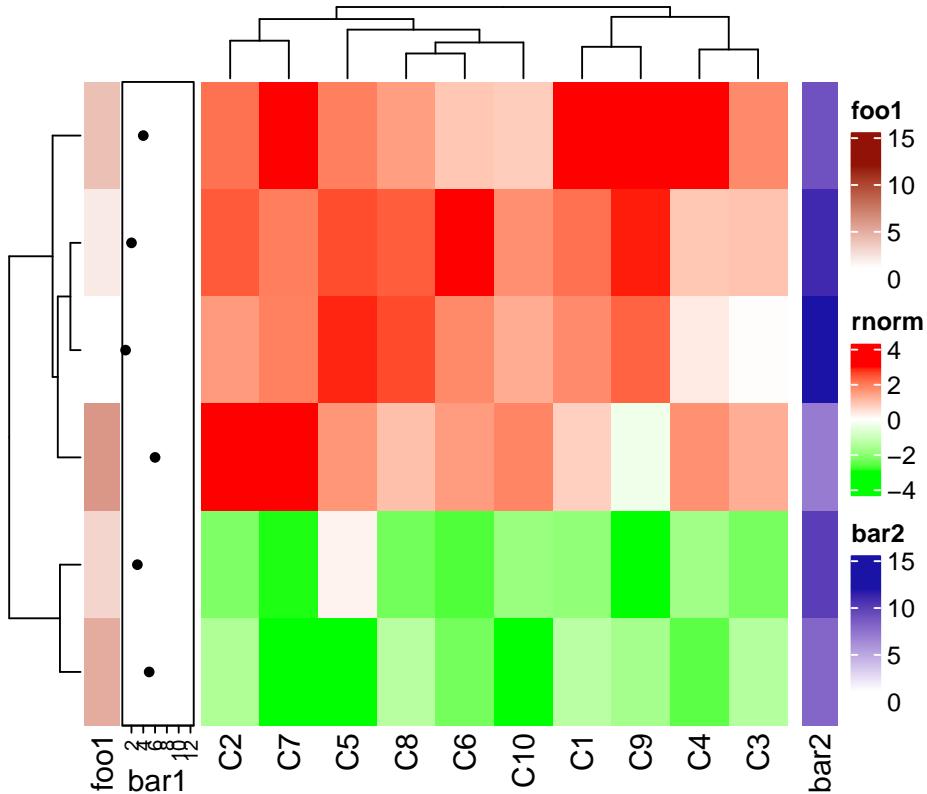
4.10 Subset the heatmap list

Similar as subsetting the `Heatmap` object (Section 2.13), the heatmap list can also be subsetted by providing row index and column index. For horizontal heatmap list, row index correspond to rows in all heatmaps and annotations, while column index only corresponds to a subset of heatmaps and annotations. For vertical heatmap list, it's the other way around.

In following we use horizontal heatmap list as example.

```
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm,
              left_annotation = rowAnnotation(foo1 = 1:12, bar1 = anno_points(1:12)))
ht2 = Heatmap(mat2, name = "runif", col = col_runif)
ha = rowAnnotation(foo2 = anno_barplot(1:12), bar2 = 12:1)
ht_list = ht1 + ht2 + ha
names(ht_list)
```

```
## [1] "rnorm" "runif" "foo2"   "bar2"
ht_list[1:6, c("rnorm", "bar2")]
```



`foo1` and `bar` are components of heatmap `rnorm`, so they can not be selected in the subset function, while `foo2` and `bar2` are independent row annotations and they can be selected to take subset of them.

4.11 Plot the heatmap list

Similar as described in Section 2.11, directly entering the HeatmapList object in interactive R session calls the `show()` method which calls the `draw()` method internally. When there is no plot after you entering the object, you should use `draw()` explicitly:

```
# code only for demonstration
draw(ht_list, ...)
```

4.12 Get orders and dendograms

`row_order()`, `column_order()`, `row_dend()` and `column_dend()` can be used to retrieve corresponding information from the heatmap list. The usage is straightforward by following examples. But remember you need to apply these functions on the object returned by `draw()`.

```
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm)
ht2 = Heatmap(mat2, name = "runif", col = col_runif)
ht_list = ht1 + ht2
ht_list = draw(ht_list)
row_order(ht_list)

## [1] 10 4 8 2 1 12 11 6 7 3 9 5
column_order(ht_list)

## $rnorm
## [1] 5 2 7 6 10 1 9 8 4 3
##
## $runif
## [1] 4 10 2 5 7 6 1 3 8 9
```

If rows or columns are split, the returned values will also be a list.

```
ht1 = Heatmap(mat1, name = "rnorm", col = col_rnorm)
ht2 = Heatmap(mat2, name = "runif", col = col_runif, column_km = 2)
ht_list = ht1 + ht2
ht_list = draw(ht_list, row_km = 2)
row_order(ht_list)

## $`1`
## [1] 7 3 9 5
##
## $`2`
## [1] 10 4 8 2 1 12 11 6
column_order(ht_list)

## $rnorm
## $rnorm[[1]]
## [1] 5 2 7 6 10 1 9 8 4 3
##
## $runif
## $runif$`2`
## [1] 4 10 2 5
##
## $runif$`1`
```

```
## [1] 7 6 1 3 8 9
```

You can specify a certain heatmap for the column order.

```
column_order(ht_list, name = "runif")

## $`2`
## [1] 4 10 2 5
##
## $`1`
## [1] 7 6 1 3 8 9
```

The logic is the same for extracting dendrograms and also the same for vertical heatmap list, so we don't show more examples here.

4.13 Change parameters globally

`ht_opt()` is an option function which controls some parameters globally. You can set some parameters for all heatmaps/annotations simultaneously by this global function. Please note you should put it before your heatmap code and reset all option values after drawing the heatmaps to get rid of affecting next heatmap.

```
ht_opt
```

## Option	Value
## heatmap_row_names_gp	NULL
## heatmap_column_names_gp	NULL
## heatmap_row_title_gp	NULL
## heatmap_column_title_gp	NULL
## legend_title_gp	NULL
## legend_title_position	NULL
## legend_labels_gp	NULL
## legend_grid_height	NULL
## legend_grid_width	NULL
## legend_border	NULL
## legend_gap	4mm, 4mm
## heatmap_border	NULL
## annotation_border	NULL
## fast_hclust	FALSE
## show_parent_dend_line	TRUE
## verbose	FALSE
## message	TRUE
## show_vp	FALSE
## simple_anno_size	5mm
## DENDROGRAM_PADDING	0.5mm
## DIMNAME_PADDING	1mm

```

##  TITLE_PADDING           NULL
##  COLUMN_ANNO_PADDING    1mm
##  ROW_ANNO_PADDING        1mm
##  HEATMAP_LEGEND_PADDING 2mm
##  ANNOTATION_LEGEND_PADDING 2mm
##  save_last               FALSE
##  validate_names          TRUE
##  raster_temp_image_max_width 30000
##  raster_temp_image_max_height 30000
##  COLOR                   blue, #EEEEEE, red

```

There are following parameters to control all heatmaps:

- `heatmap_row_names_gp`: set `row_names_gp` in all `Heatmap()`.
- `heatmap_column_names_gp`: set `column_names_gp` in all `Heatmap()`.
- `heatmap_row_title_gp`: set `row_title_gp` in all `Heatmap()`.
- `heatmap_column_title_gp`: set `column_title_gp` in all `Heatmap()`.
- `heatmap_border`: set `border` in all `Heatmap()`.

Following parameters control the legends:

- `legend_title_gp`: set `title_gp` in all heatmap legends and annotation legends.
- `legend_title_position`: set `title_position` in all heatmap legends and annotation legends.
- `legend_labels_gp`: set `labels_gp` in all heatmap legends and annotation legends.
- `legend_grid_width`: set `grid_width` in all heatmap legends and annotation legends.
- `legend_grid_height`: set `grid_height` in all heatmap legends and annotation legends.
- `legend_border`: set `border` in all heatmap legends and annotation legends.

Following parameters control heatmap annotations:

- `annotation_border`: set `border` in all `HeatmapAnnotation()`.
- `anno_simple_size`: set size for the simple annotation.

Following parameters control the space between heatmap components:

- `DENDROGRAM_PADDING`: space bewteen dendograms and heatmap body.
- `DIMNAME_PADDING`: space between row/column names and heatmap body.
- `TITLE_PADDING`: space between row/column titles and heatmap body.
- `COLUMN_ANNO_PADDING`: space between column annotations and heatmap body.
- `ROW_ANNO_PADDING`: space between row annotations and heatmap body.

Other parameters:

- `fast_hclust`: whether use `fastcluster::hclust()` to speed up clustering?

- `show_parent_dend_line`: when heatmap is split, whether to add a dashed line to mark parent dendrogram and children dendrograms?

You can get or set option values by the traditional way (like `base::options()`) or by \$ operator:

```
ht_opt("heatmap_row_names_gp")
ht_opt$heatmap_row_names_gp

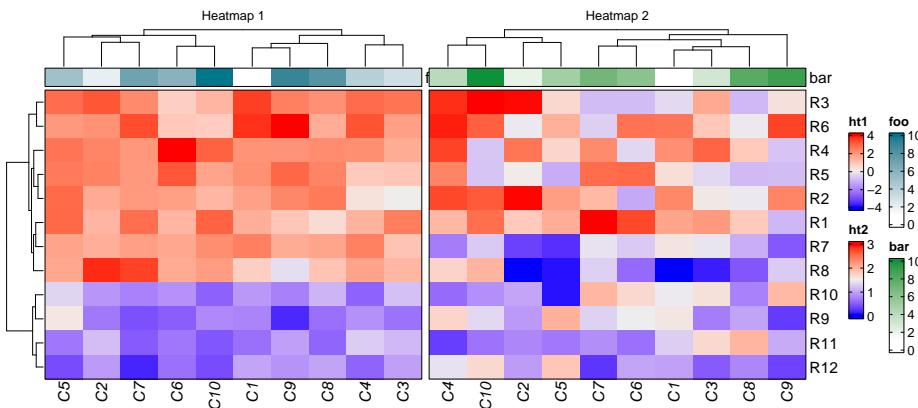
# to set option values
ht_opt("heatmap_row_names_gp" = gpar(fontsize = 8))
ht_opt$heatmap_row_names_gp = gpar(fontsize = 8)
```

Reset to the default values by:

```
ht_opt(RESET = TRUE)
```

Following example shows to control some graphic parameters globally.

```
ht_opt(heatmap_column_names_gp = gpar(fontface = "italic"),
       heatmap_column_title_gp = gpar(fontsize = 10),
       legend_border = "black",
       heatmap_border = TRUE,
       annotation_border = TRUE
)
ht1 = Heatmap(mat1, name = "ht1", column_title = "Heatmap 1",
              top_annotation = HeatmapAnnotation(foo = 1:10))
ht2 = Heatmap(mat2, name = "ht2", column_title = "Heatmap 2",
              top_annotation = HeatmapAnnotation(bar = 1:10))
ht1 + ht2
```



```
ht_opt(RESET = TRUE)
```

These global parameters can also be set in the `draw()` function to temporarily change the global parameters, and they are reset back after the plot is made.

Please check the help page of `drawHeatmapList-method`.

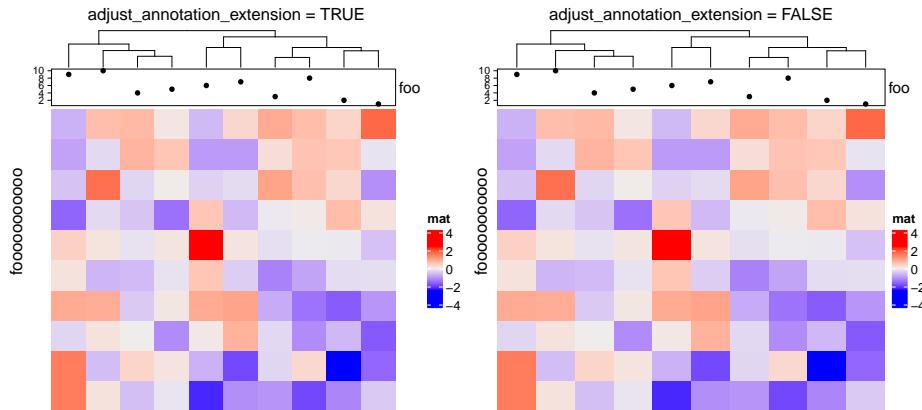
4.14 Adjust blank space caused by annotations

Heatmap annotations may have annotation names and axes, for which the spaces are also taken into account when arranging heatmap components in the final layout. Sometimes, this adjustment is not smart that you may see blank areas in the plot that are not necessary.

One scenario is for a matrix with no row names, the space to the right of the heatmap is determined by the size of annotation name, which results in blank space between the heatmap and the legend. Also the heatmap list level row title is plotted to the left of the annotation axis, which gives blank area if there is no row dendrogram.

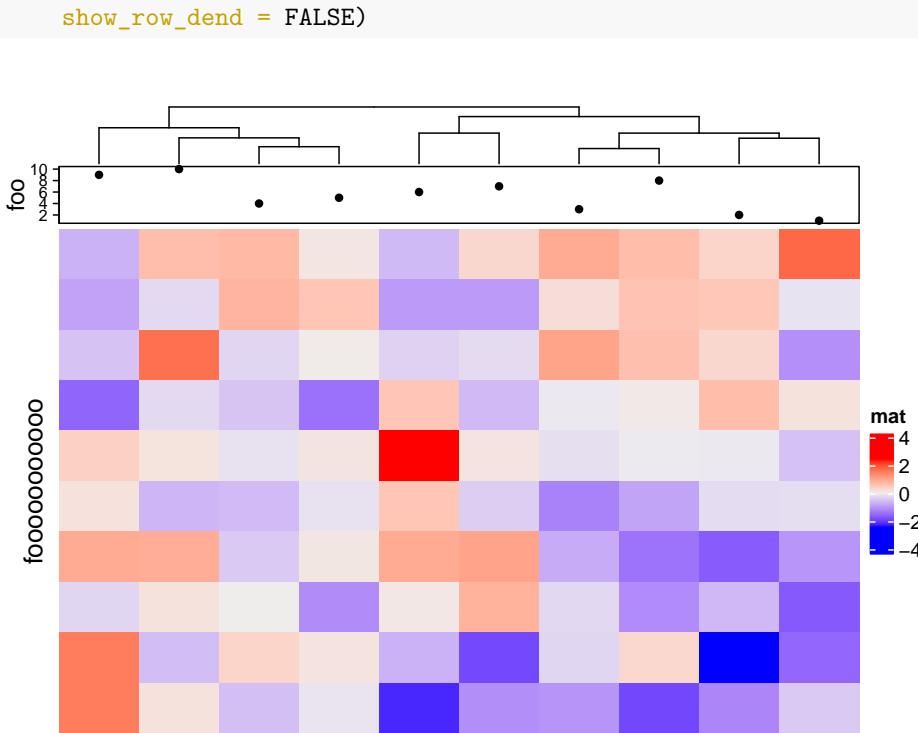
`adjust_annotation_extension` controls whether to take account of the space of annotation names and axes for the layout. Compare following two plots.

```
m = matrix(rnorm(100), 10)
ht = Heatmap(m, name = "mat",
             top_annotation = HeatmapAnnotation(foo = anno_points(1:10)),
             show_row_dend = FALSE)
draw(ht, row_title = "oooooooooooo", adjust_annotation_extension = TRUE, # default
      column_title = "adjust_annotation_extension = TRUE")
draw(ht, row_title = "oooooooooooo", adjust_annotation_extension = FALSE,
      column_title = "adjust_annotation_extension = FALSE")
```



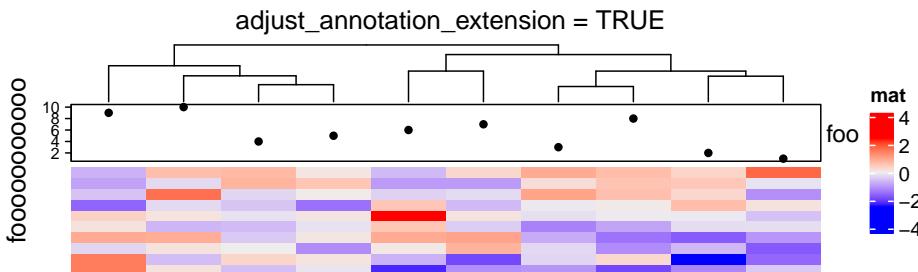
Another way to partially solve the space problem is to move the annotation name to the left and use heatmap-level row title.

```
Heatmap(m, name = "mat",
        top_annotation = HeatmapAnnotation(foo = anno_points(1:10),
                                           annotation_name_side = "left"),
        row_title = "oooooooooooo",
```



However, this adjustment for annotations sometimes is also necessary, e.g. when the heatmap is very short:

```
ht = Heatmap(m, name = "mat",
             top_annotation = HeatmapAnnotation(foo = anno_points(1:10)),
             show_row_dend = FALSE)
draw(ht, row_title = "oooooooooooo", adjust_annotation_extension = TRUE,
      column_title = "adjust_annotation_extension = TRUE")
```



Therefore, we set `TRUE` as the default of `adjust_annotation_extension` and users can configure it based on specific scenarios.

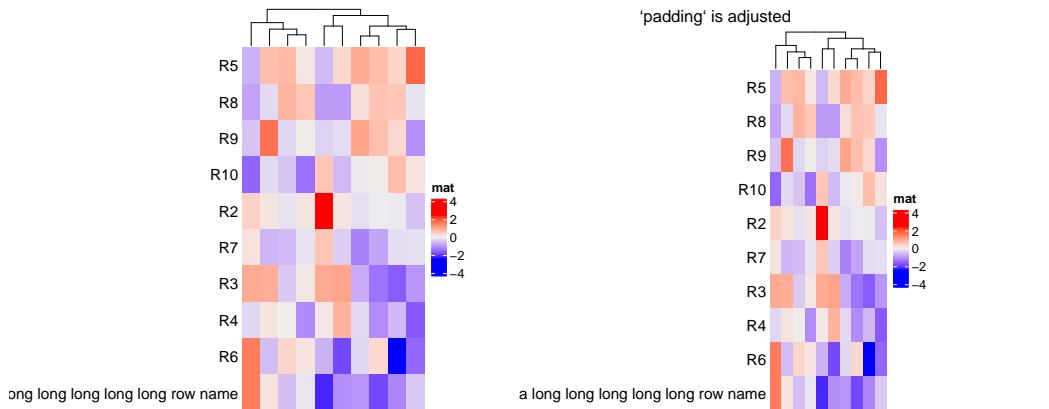
4.15 Manually increase space around the plot

The layout of the **ComplexHeatmap** is not perfect that it is still possible some of the text are drawn out of the plotting region. In this case, you can manually set the `padding` argument in `draw()` function to increase the blank areas around the final plot.

The value of `padding` should be a unit vector with length of four. The four values correspond to the space at the bottom, left, top and right sides.

The following example is not a perfect example because the maximal width for row names can be controlled by `max_row_name_width` argument, but we can still use it to demonstrate the use of `padding`.

```
m2 = m
rownames(m2) = paste0("R", 1:10)
rownames(m2)[1] = "a long long long long long row name"
ht = Heatmap(m2, name = "mat", row_names_side = "left", show_row_dend = FALSE)
draw(ht, padding = unit(c(2, 20, 2, 2), "mm")) ## see right heatmap in following
```



Chapter 5

Legends

The heatmaps and simple annotations automatically generate legends which are put one the right side of the heatmap. By default there is no legend for complex annotations, but they can be constructed and added manually (Section 5.5). All legends are internally constructed by `Legend()` constructor. In later sections, we first introduce the settings for continuous legends and discrete legends, then we will discuss how to configure the legends associated with the heatmaps and annotations, and how to add new legends to the plot.

All the legends (no matter a single legend or a pack of legends) all belong to the `Legends` class. The class only has one slot `grob` which is the real `grid::grob` object or the `grid::gTree` object that records how to draw the graphics. The wrapping of the `Legends` class and the methods designed for the class make legends as single objects and can be drawn like points with specifying the positions on the viewport.

The legends for heatmaps and annotations can be controlled by `heatmap_legend_param` argument in `Heatmap()`, or `annotation_legend_param` argument in `HeatmapAnnotation()`. **Most of the parameters in `Legend()` function can be directly set in the two arguments with the same parameter name.** The details of setting heatmap legends and annotation legends parameters are introduced in Section 5.4.

5.1 Continuous legends

Since most of heatmaps contain continuous values, we first introduce the settings for the continuous legend.

Continuous legend needs a color mapping function which should be generated by `circlize::colorRamp2()`. In the heatmap legends and annotation legends that are automatically generated, the color mapping functions are passed by the

`col` argument from `Heatmap()` or `HeatmapAnnotation()` function, while if you construct a self-defined legend, you need to provide the color mapping function.

The break values provided in the color mapping function (e.g. `c(0, 0.5, 1)` in following example) will not exactly be the same as the break values in the legends). The finally break values presented in the legend are internally adjusted to make the numbers of labels close to 5 or 6.

First we show the default style of a vertical continuous legend:

```
library(circlize)
col_fun = colorRamp2(c(0, 0.5, 1), c("blue", "white", "red"))
lgd = Legend(col_fun = col_fun, title = "foo")
```



`lgd` is a `Legends` class object. The size of the legend can be obtained by `ComplexHeatmap:::width()` and `ComplexHeatmap:::height()` function.

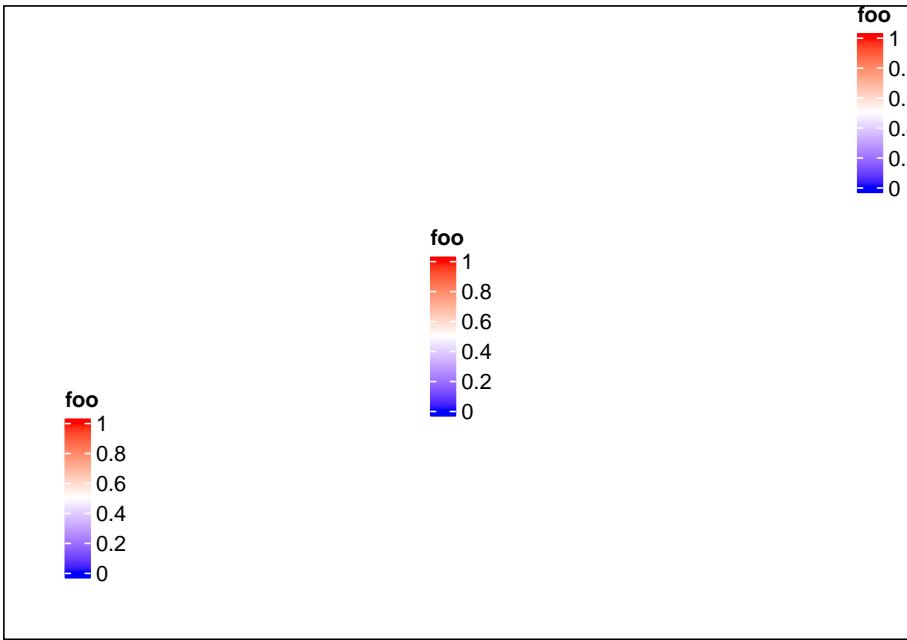
```
ComplexHeatmap:::width(lgd)
```

```
## [1] 9.9036111111111mm
ComplexHeatmap:::height(lgd)
```

```
## [1] 30.2744052165491mm
```

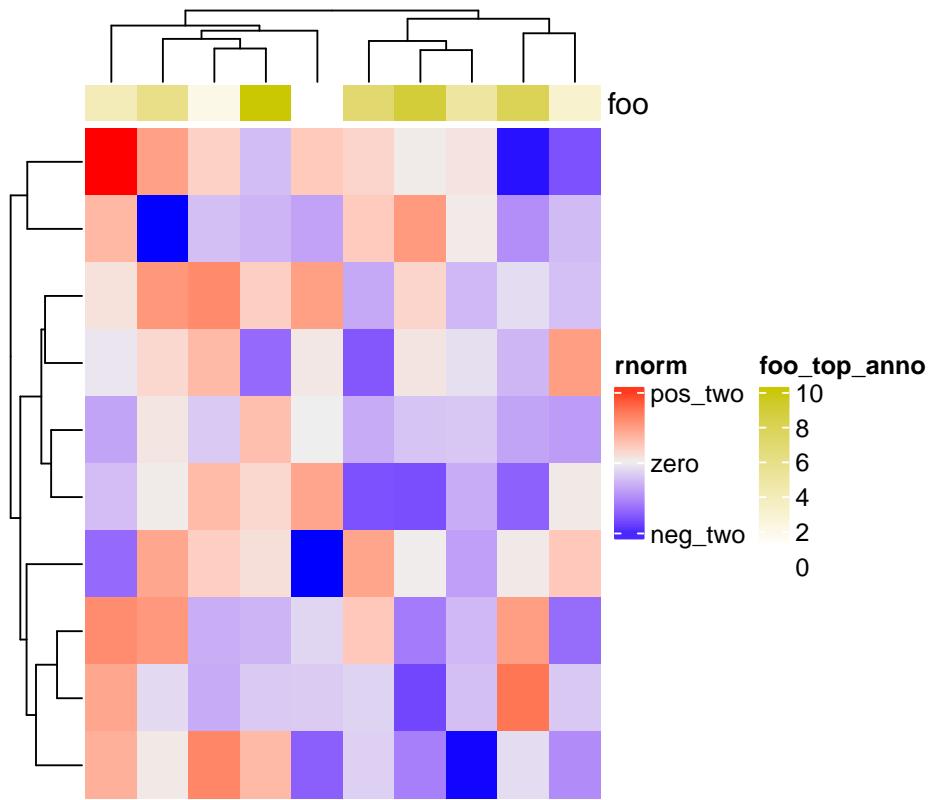
The legend is actually a packed graphic object composed of rectangles, lines and texts. It can be added to the plot by `draw()` function. In `ComplexHeatmap` pacakge, you don't need to use `draw()` directly on legend objects, but it might be useful if you use the legend objects in other places.

```
pushViewport(viewport(width = 0.9, height = 0.9))
grid.rect() # border
draw(lgd, x = unit(1, "cm"), y = unit(1, "cm"), just = c("left", "bottom"))
draw(lgd, x = unit(0.5, "npc"), y = unit(0.5, "npc"))
draw(lgd, x = unit(1, "npc"), y = unit(1, "npc"), just = c("right", "top"))
popViewport()
```



If you only want to configure the legends generated by heatmaps or annotations, you don't need to construct the `Legends` object by your own. The parameters introduced later can be directly used to customize the legends by `heatmap_legend_param` argument in `Heatmap()` and `annotation_legend_param` argument in `HeatmapAnnotation()` (introduced in Section 5.4). It is still nice to see how these parameters change the styles of the legend in following examples. Following is a simple example showing how to configure legends in the heatmap and heatmap annotation.

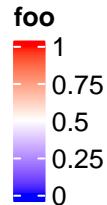
```
Heatmap(matrix(rnorm(100), 10),
       heatmap_legend_param = list(
         title = "rnorm", at = c(-2, 0, 2),
         labels = c("neg_two", "zero", "pos_two")
       ),
       top_annotation = HeatmapAnnotation(
         foo = 1:10,
         annotation_legend_param = list(foo = list(title = "foo_top_anno")))
     ))
```



In following examples, we only show how to construct the legend object, while not show the code which draws the legends. Only remember you can use `draw()` function on the `Legends` object to draw the single legend on the plot.

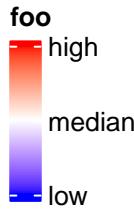
For continuous legend, you can manually adjust the break values in the legend by setting `at`. Note the height is automatically adjusted.

```
lgd = Legend(col_fun = col_fun, title = "foo", at = c(0, 0.25, 0.5, 0.75, 1))
```



The labels corresponding to the break values are set by `labels`.

```
lgd = Legend(col_fun = col_fun, title = "foo", at = c(0, 0.5, 1),
             labels = c("low", "median", "high"))
```



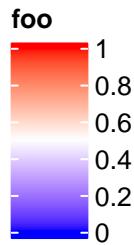
The height of the vertical continuous legend is set by `legend_height`. `legend_height` can only be set for the vertical continuous legend and the value is the height of the legend body (excluding the legend title).

```
lgd = Legend(col_fun = col_fun, title = "foo", legend_height = unit(6, "cm"))
```



If it is a vertical legend, `grid_width` controls the widths of the legend body. `grid_width` is originally designed for the discrete legends where each level in the legend is a grid, but here we use the same name for the parameter that controls the width of the legend.

```
lgd = Legend(col_fun = col_fun, title = "foo", grid_width = unit(1, "cm"))
```



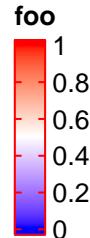
The graphic parameters for the labels are controlled by `labels_gp`.

```
lgd = Legend(col_fun = col_fun, title = "foo", labels_gp = gpar(col = "red", font = 3))
```



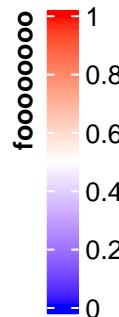
The border of the legend as well as the ticks for the break values are controlled by `border`. The value of `border` can be logical or a string of color.

```
lgd = Legend(col_fun = col_fun, title = "foo", border = "red")
```

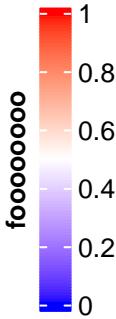


`title_position` controls the position of titles. For vertical legends, the value should be one of `topleft`, `topcenter`, `lefttop-rot` and `leftcenter-rot`. Following two plots show the effect of `lefttop-rot` title and `leftcenter-rot` title.

```
lgd = Legend(col_fun = col_fun, title = "fooooooooo", title_position = "lefttop-rot",
             legend_height = unit(4, "cm"))
```

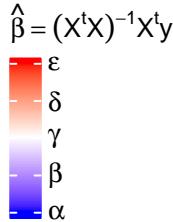


```
lgd = Legend(col_fun = col_fun, title = "fooooooooo", title_position = "leftcenter-rot",
             legend_height = unit(4, "cm"))
```



Legend titles and labels can be set as mathematical formulas.

```
lgd = Legend(col_fun = col_fun, title = expression(hat(beta) == (X^t * X)^{-1} * X^t * y),
             at = c(0, 0.25, 0.5, 0.75, 1), labels = expression(alpha, beta, gamma, delta, epsilon))
```



More complicated texts can be added by using the `gridtext` package (Section 10.3.5).

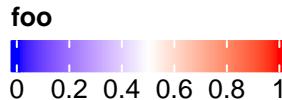
```
lgd = Legend(col_fun = col_fun,
             title = gt_render("<span style='color:orange'>**Legend title**</span>"),
             title_gp = gpar(box_fill = "grey"),
             at = c(-3, 0, 3),
             labels = gt_render(c("<span style='color:blue'>*negative*</span> three", "zero",
                                 "<span style='color:red'>*positive*</span> three"))
)
```



Settings for horizontal continuous legends are almost the same as vertical legends, except that now `legend_width` controls the width of the legend, and the title position can only be one of `topcenter`, `topleft`, `lefttop` and `leftcenter`.

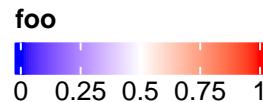
The default style for horizontal legend:

```
lgd = Legend(col_fun = col_fun, title = "foo", direction = "horizontal")
```



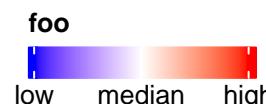
Manually set at:

```
lgd = Legend(col_fun = col_fun, title = "foo", at = c(0, 0.25, 0.5, 0.75, 1),  
direction = "horizontal")
```



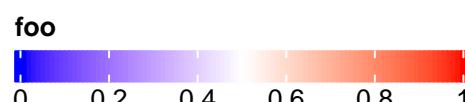
Manually set labels:

```
lgd = Legend(col_fun = col_fun, title = "foo", at = c(0, 0.5, 1),  
labels = c("low", "median", "high"), direction = "horizontal")
```



Set legend_width:

```
lgd = Legend(col_fun = col_fun, title = "foo", legend_width = unit(6, "cm"),  
direction = "horizontal")
```



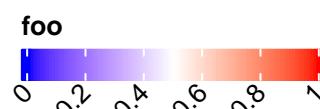
Set graphic parameters for labels:

```
lgd = Legend(col_fun = col_fun, title = "foo", labels_gp = gpar(col = "red", font = 3),  
direction = "horizontal")
```



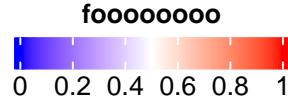
Set rotations of labels:

```
lgd = Legend(col_fun = col_fun, title = "foo", labels_rot = 45,  
direction = "horizontal")
```

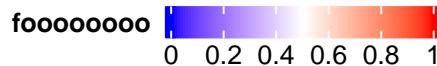


Title can be set as `topleft`, `topcenter` or `lefttop` and `leftcenter`.

```
lgd = Legend(col_fun = col_fun, title = "fooooooooo", direction = "horizontal",
             title_position = "topcenter")
```

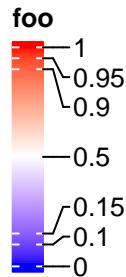


```
lgd = Legend(col_fun = col_fun, title = "fooooooooo", direction = "horizontal",
             title_position = "lefttop")
```



In examples we showed above, the intervals between every two break values are equal. Actually `at` can also be set as break values with unequal intervals. In this scenario, the ticks on the legend are still at the original places while the corresponding texts are shifted to get rid of overlapping. Then, there are lines connecting the ticks and the labels.

```
lgd = Legend(col_fun = col_fun, title = "foo", at = c(0, 0.1, 0.15, 0.5, 0.9, 0.95, 1))
```



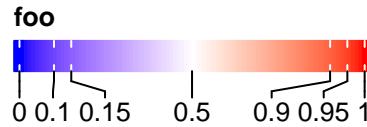
If the labels do not need to be adjusted, they are still at the original places.

```
lgd = Legend(col_fun = col_fun, title = "foo", at = c(0, 0.3, 1),
             legend_height = unit(4, "cm"))
```



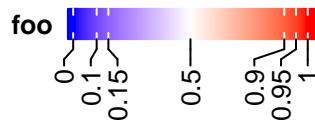
It is similar for the horizontal legends:

```
lgd = Legend(col_fun = col_fun, title = "foo", at = c(0, 0.1, 0.15, 0.5, 0.9, 0.95, 1),
             direction = "horizontal")
```



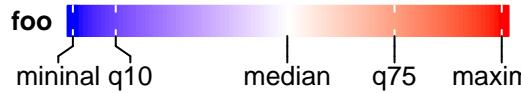
Set rotations of labels to 90 degree.

```
lgd = Legend(col_fun = col_fun, title = "foo", at = c(0, 0.1, 0.15, 0.5, 0.9, 0.95, 1),
             direction = "horizontal", title_position = "lefttop", labels_rot = 90)
```



When the position of title is set to `lefttop`, the area below the title will also be taken into account when calculating the adjusted positions of labels.

```
lgd = Legend(col_fun = col_fun, title = "foo", at = c(0, 0.1, 0.5, 0.75, 1),
             labels = c("mininal", "q10", "median", "q75", "maxin"),
             direction = "horizontal", title_position = "lefttop")
```



If `at` is set in the decreasing order, the legend is reversed, *i.e.* the smallest value is on the top of the legend.

```
lgd = Legend(col_fun = col_fun, title = "foo", at = c(1, 0.8, 0.6, 0.4, 0.2, 0))
```



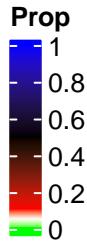
Most continuous legends have legend breaks with equal distance, which I mean, *e.g.* the distance between the first and the second breaks are the same as the distance between the second and the third breaks. However, there are still special cases where users want to set legend breaks with unequal distances.

In the following example, the color mapping function `col_fun_prop` visualizes proportion values with breaks in `c(0, 0.05, 0.1, 0.5, 1)`. The legend breaks with unequal distance might reflect the different importance of the values in

`c(0, 1)`. For example, maybe we want to see more details in the interval `c(0, 0.1)`.

Following is the default style of the legend where the breaks are selected from 0 to 1 with equal distance.

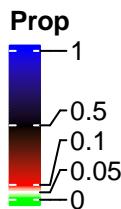
```
col_fun_prop = colorRamp2(c(0, 0.05, 0.1, 0.5, 1),
                           c("green", "white", "red", "black", "blue"))
lgd = Legend(col_fun = col_fun_prop, title = "Prop")
```



You can't see the details in the interval `c(0, 0.1)`, right? This also reminds us that the breaks set in `colorRamp2()` only defines the color mapping while not determine the breaks in the legend.

If we manually select the break values, the color bar keeps the same. The labels are shifted and lines connect them to the original positions. In this case, the distance in the color bar is still proportional to the real difference in the break values, *i.e.*, the distance between 0.5 and 1 is five times longer than 0 and 0.1.

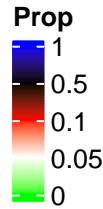
```
col_fun_prop = colorRamp2(c(0, 0.05, 0.1, 0.5, 1),
                           c("green", "white", "red", "black", "blue"))
lgd = Legend(col_fun = col_fun_prop, title = "Prop",
             at = c(0, 0.05, 0.1, 0.5, 1))
```



From version 2.7.1, `Legend()` function has a new argument `break_dist` that controls the distance between two neighbouring break values in the legend. **It might be confusing, but from here, when I mention “break distance,” it always means the visual distance in the legend.**

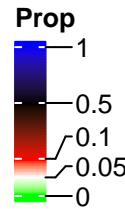
The value of `break_dist` should have length either one which means all break values have equal distance in the legend, or `length(at) - 1`.

```
lgd = Legend(col_fun = col_fun_prop, title = "Prop", break_dist = 1)
```



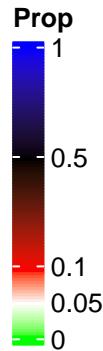
And in the following example, the top two break intervals are three times longer than the bottom two intervals.

```
lgd = Legend(col_fun = col_fun_prop, title = "Prop", break_dist = c(1, 1, 3, 3))
```



If we increase the legend height by `legend_height` argument, there will be enough space for the labels and their positions are not adjusted any more.

```
lgd = Legend(col_fun = col_fun_prop, title = "Prop", break_dist = c(1, 1, 3, 3),
             legend_height = unit(4, "cm"))
```

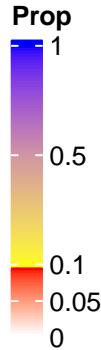


Imaging following user case, we want to use one color scheme for the values in `c(0, 0.1)` and a second color schema for the values in `c(0.1, 1)`, maybe for the reason that we want to emphasize the two intervals are very different. The color mapping can be defined as:

```
col_fun2 = colorRamp2(c(0, 0.1, 0.1+1e-6, 1), c("white", "red", "yellow", "blue"))
```

So here I just added a tiny shift (`1e-6`) to 0.1 and set it as the lower bound for the second color scheme. The legend looks like:

```
lgd = Legend(col_fun = col_fun2, title = "Prop", at = c(0, 0.05, 0.1, 0.5, 1),
             break_dist = c(1, 1, 3, 3), legend_height = unit(4, "cm"))
```



Now you can see the colors are not changed smoothly from 0 to 1 and there are two distinct color schemes.

5.2 Discrete legends

Discrete legends are used for discrete color mappings. The continuous color mapping can also be degenerated as discrete color mapping by only providing the colors and the break values.

You can either specify `at` or `labels`, but most probably you specify `labels`. The colors should be specified by `legend_gp`.

```
lgd = Legend(at = 1:6, title = "foo", legend_gp = gpar(fill = 1:6))
```

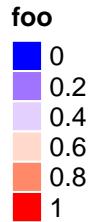


```
lgd = Legend(labels = month.name[1:6], title = "foo", legend_gp = gpar(fill = 1:6))
```



The discrete legend for continuous color mapping:

```
at = seq(0, 1, by = 0.2)
lgd = Legend(at = at, title = "foo", legend_gp = gpar(fill = col_fun(at)))
```



The position of title:

```
lgd = Legend(labels = month.name[1:6], title = "foo", legend_gp = gpar(fill = 1:6),
             title_position = "lefttop")
```

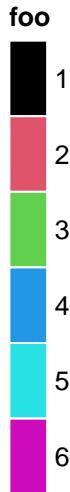


```
lgd = Legend(labels = month.name[1:6], title = "foo", legend_gp = gpar(fill = 1:6),
             title_position = "leftcenter-rot")
```



The size of grids are controlled by `grid_width` and `grid_height`.

```
lgd = Legend(at = 1:6, legend_gp = gpar(fill = 1:6), title = "foo",
             grid_height = unit(1, "cm"), grid_width = unit(5, "mm"))
```



The graphic parameters of labels are controlled by `labels_gp`.

```
lgd = Legend(labels = month.name[1:6], legend_gp = gpar(fill = 1:6), title = "foo",
             labels_gp = gpar(col = "red", fontsize = 14))
```



The graphic parameters of the title are controlled by `title_gp`.

```
lgd = Legend(labels = month.name[1:6], legend_gp = gpar(fill = 1:6), title = "foo",
             title_gp = gpar(col = "red", fontsize = 14))
```



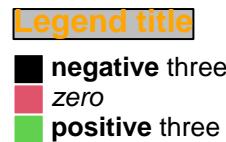
Title and labels are be complicated texts by integrating `gridtext` package (Section 10.3.5):

```
lgd = Legend(
  title = gt_render("<span style='color:orange'>**Legend title**</span>"),
```

```

    title_gp = gpar(box_fill = "grey"),
    at = c(-3, 0, 3),
    labels = gt_render(c("**negative** three", "*zero*", "**positive** three")),
    legend_gp = gpar(fill = 1:3)
)

```



Borders of grids are controlled by `border`.

```
lgd = Legend(labels = month.name[1:6], legend_gp = gpar(fill = 1:6), title = "foo",
             border = "red")
```



One important thing for the discrete legend is you can arrange the grids into multiple rows or/and columns. If `ncol` is set to a number, the grids are arranged into `ncol` columns.

```
lgd = Legend(labels = month.name[1:10], legend_gp = gpar(fill = 1:10),
              title = "foo", ncol = 3)
```



Still the title position is calculated based on the multiplt-column legend.

```
lgd = Legend(labels = month.name[1:10], legend_gp = gpar(fill = 1:10), title = "foo",
              ncol = 3, title_position = "topcenter")
```



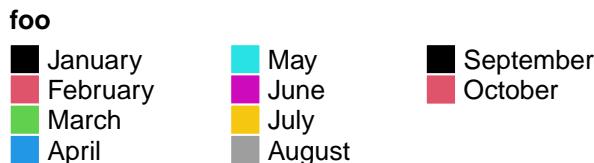
You can choose to list the legend levels by rows by setting `by_row = TRUE`.

```
lgd = Legend(labels = month.name[1:10], legend_gp = gpar(fill = 1:10), title = "foo",
  ncol = 3, by_row = TRUE)
```



The gaps between two columns are controlled by `gap` or `column_gap`. These two arguments are treated the same.

```
lgd = Legend(labels = month.name[1:10], legend_gp = gpar(fill = 1:10), title = "foo",
  ncol = 3, gap = unit(1, "cm"))
```



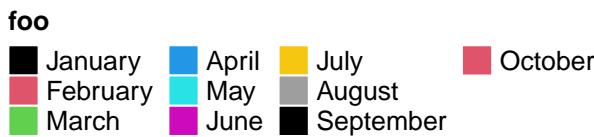
The gaps between rows are controlled by `row_gap`.

```
lgd = Legend(labels = month.name[1:10], legend_gp = gpar(fill = 1:10), title = "foo",
  ncol = 3, row_gap = unit(5, "mm"))
```



Instead of `ncol`, you can also specify the layout by `nrow`. Note you cannot use `ncol` and `nrow` at a same time.

```
lgd = Legend(labels = month.name[1:10], legend_gp = gpar(fill = 1:10),
  title = "foo", nrow = 3)
```



One extreme case is when all levels are put in one row and the title are rotated by 90 degree. The height of the legend will be the height of the rotated title.

```
lgd = Legend(labels = month.name[1:6], legend_gp = gpar(fill = 1:6), title = "foooooo"
             nrow = 1, title_position = "lefttop-rot")
```



Following style a lot of people might like:

```
lgd = Legend(labels = month.name[1:6], legend_gp = gpar(fill = 1:6), title = "foooooo"
             nrow = 1, title_position = "leftcenter")
```



`Legend()` also supports to use simple graphics (e.g. points, lines, boxplots) as legends. `type` argument can be specified as `points` or `p` that you can use number for `pch` or single-letter for `pch`.

```
lgd = Legend(labels = month.name[1:6], title = "foo", type = "points",
             pch = 1:6, legend_gp = gpar(col = 1:6), background = "#FF8080")
```

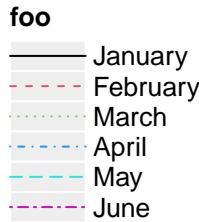


```
lgd = Legend(labels = month.name[1:6], title = "foo", type = "points",
             pch = letters[1:6], legend_gp = gpar(col = 1:6), background = "white")
```



Or set `type = "lines"`/`type = "l"` to use lines as legend:

```
lgd = Legend(labels = month.name[1:6], title = "foo", type = "lines",
             legend_gp = gpar(col = 1:6, lty = 1:6), grid_width = unit(1, "cm"))
```



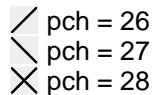
Or set `type = "boxplot"/type = "box"` to use boxes as legends:

```
lgd = Legend(labels = month.name[1:6], title = "foo", type = "boxplot",
  legend_gp = gpar(fill = 1:6))
```



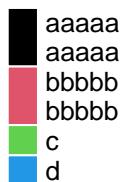
When `pch` is an integer number, the numbers in `26:28` correspond to following symbols:

```
lgd = Legend(labels = paste0("pch = ", 26:28), type = "points", pch = 26:28)
```



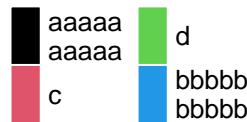
In all examples showed above, the labels are single lines. Multiple-line labels are also supported. As shown in the following example, legend grids for multiple-line labels are automatically elongated.

```
lgd = Legend(labels = c("aaaaaa\naaaaaa", "bbbbbb\nbbbbbb", "c", "d"),
  legend_gp = gpar(fill = 1:4))
```



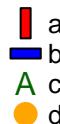
If the legend is arranged in multiple rows or columns, the sizes of legend grids are adjusted to the label with the most number of lines.

```
lgd = Legend(labels = c("aaaaaa\naaaaaa", "c", "d", "bbbbbb\nbbbbbb"),
  legend_gp = gpar(fill = 1:4), nrow = 2)
```



The last useful argument `graphics` can be used to self-define the legend graphics. The value for `graphics` should be a list of functions with four arguments: `x` and `y`: the center of the legend grid, `w` and `h`: the width and height of the legend grid. Length of `graphics` should be the same as `at` or `labels`. If `graphics` is a named list where the names correspond to `labels`, then the order of the list of `graphics` is automatically adjusted.

```
lgd = Legend(labels = letters[1:4],
             graphics = list(
               function(x, y, w, h) grid.rect(x, y, w*0.33, h, gp = gpar(fill = "red")),
               function(x, y, w, h) grid.rect(x, y, w, h*0.33, gp = gpar(fill = "blue")),
               function(x, y, w, h) grid.text("A", x, y, gp = gpar(col = "darkgreen")),
               function(x, y, w, h) grid.points(x, y, gp = gpar(col = "orange"), pch = 16)
             ))
```



5.3 A list of legends

A list of legends can be constructed or packed as a `Legends` object where the individual legends are arranged within a certain layout. The legend list can be sent to `packLegend()` separately or as a list. The legend can be arranged either vertically or horizontally. `ComplexHeatmap` uses `packLegend()` internally to arrange multiple legends. Normally you don't need to manually control the arrangement of multiple legends, but the following section would be useful if you want to manually construct a list of legends and apply to other plots.

```
lgd1 = Legend(at = 1:6, legend_gp = gpar(fill = 1:6), title = "legend1")
lgd2 = Legend(col_fun = col_fun, title = "legend2", at = c(0, 0.25, 0.5, 0.75, 1))
lgd3 = Legend(labels = month.name[1:3], legend_gp = gpar(fill = 7:9), title = "legend3")

pd = packLegend(lgd1, lgd2, lgd3)
# which is same as
pd = packLegend(list = list(lgd1, lgd2, lgd3))
```

legend1**legend2****legend3**

Simillar as single legend, you can draw the packed legends by `draw()` function. Also you can get the size of pd by `ComplexHeatmap:::width()` and `ComplexHeatmap:::height()`.

```
ComplexHeatmap:::width(pd)
```

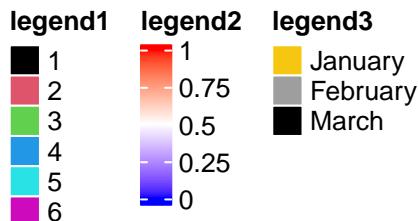
```
## [1] 19.167555555556mm
```

```
ComplexHeatmap:::height(pd)
```

```
## [1] 78.698833333334mm
```

Horizontally arranging the legends simply by setting `direction = "horizontal"`.

```
pd = packLegend(lgd1, lgd2, lgd3, direction = "horizontal")
```



One feature of `packLegend()` is, e.g. if the packing is vertically and the sum of the packed legends exceeds the height specified by `max_height`, it will be rearranged as mutliple column layout. In following example, the maximum height is 10cm.

When all the legends are put into multiple columns, `column_gap` controls the space between two columns.

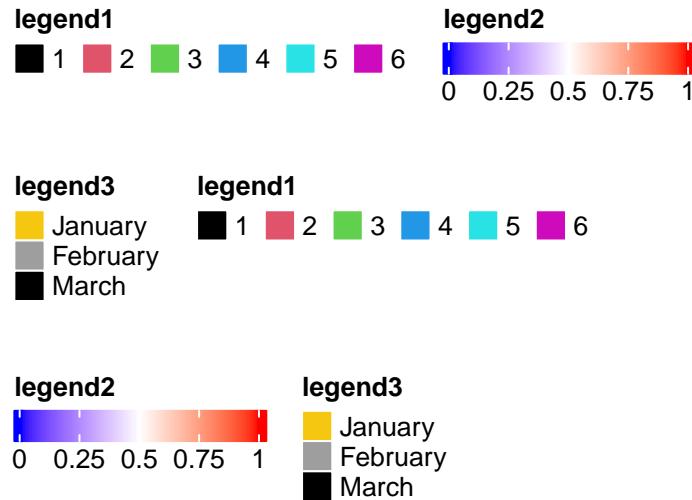
```
pd = packLegend(lgd1, lgd3, lgd2, lgd3, lgd2, lgd1, max_height = unit(10, "cm"),
                column_gap = unit(1, "cm"))
```



Similar for horizontal packing:

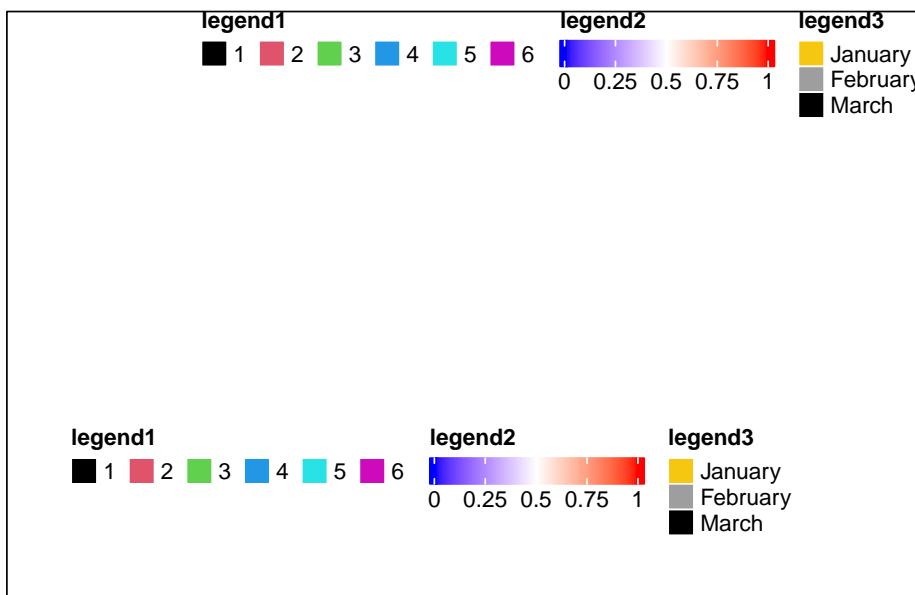
```
lgd1 = Legend(at = 1:6, legend_gp = gpar(fill = 1:6), title = "legend1",
              nr = 1)
lgd2 = Legend(col_fun = col_fun, title = "legend2", at = c(0, 0.25, 0.5, 0.75, 1),
              direction = "horizontal")

pd = packLegend(lgd1, lgd2, lgd3, lgd1, lgd2, lgd3, max_width = unit(10, "cm"),
                direction = "horizontal", column_gap = unit(5, "mm"), row_gap = unit(1, "cm"))
```



The packed legends `pd` is also a `Legends` object, which means you can use `draw()` to draw it by specifying the positions.

```
pd = packLegend(lgd1, lgd2, lgd3, direction = "horizontal")
pushViewport(viewport(width = 0.8, height = 0.8))
grid.rect()
draw(pd, x = unit(1, "cm"), y = unit(1, "cm"), just = c("left", "bottom"))
draw(pd, x = unit(1, "npc"), y = unit(1, "npc"), just = c("right", "top"))
popViewport()
```

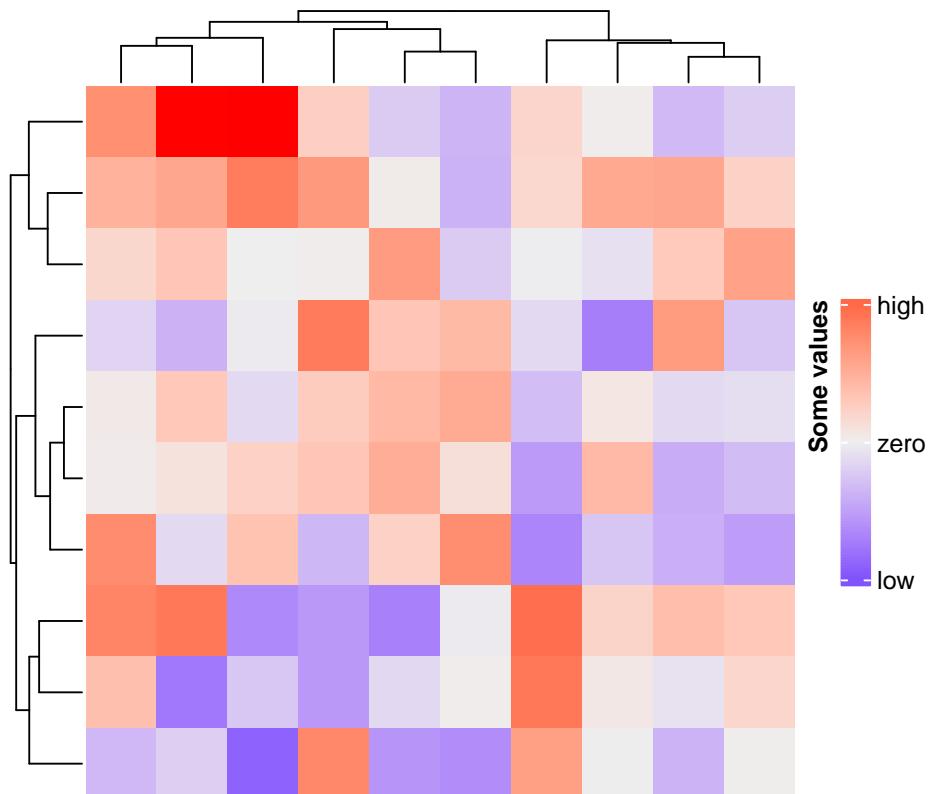


To be mentioned again, `packLegend()` is used internally to manage the list of heatmap and annotation legends.

5.4 Heatmap and annotation legends

Settings for heatmap legend are controlled by `heatmap_legend_param` argument in `Heatmap()`. The value for `heatmap_legend_param` is a list of parameters which are supported in `Legend()`.

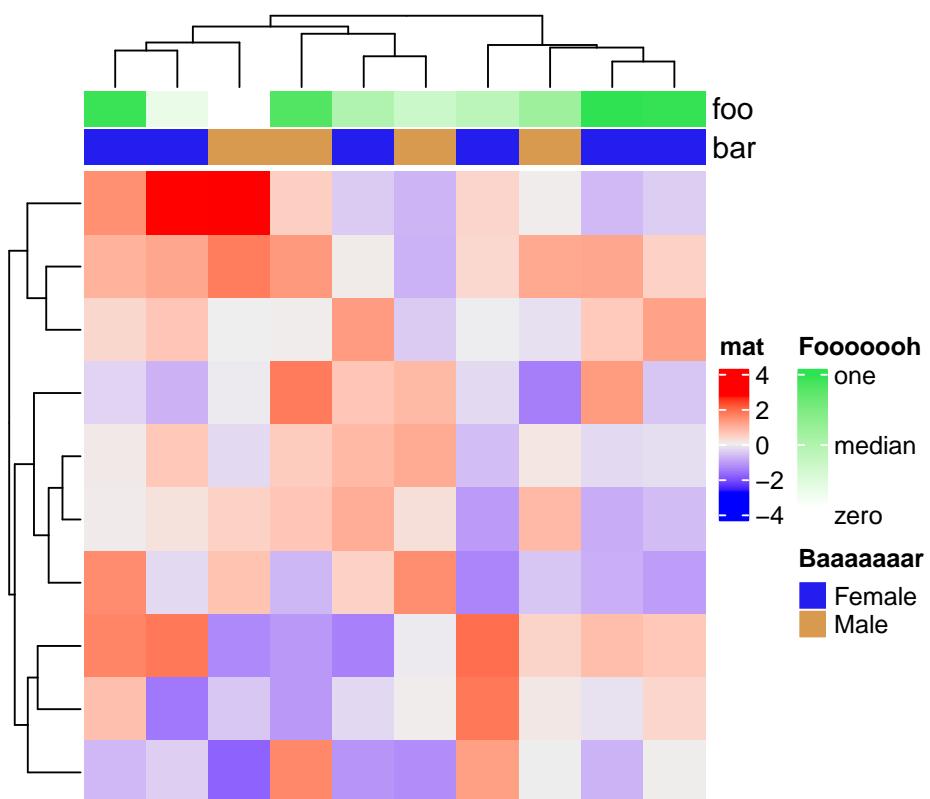
```
m = matrix(rnorm(100), 10)
Heatmap(m, name = "mat", heatmap_legend_param = list(
  at = c(-2, 0, 2),
  labels = c("low", "zero", "high"),
  title = "Some values",
  legend_height = unit(4, "cm"),
  title_position = "lefttop-rot"
))
```



`annotation_legend_param` controls legends for annotations. Since a

HeatmapAnnotation may contain multiple annotations, the value of annotation_legend_param is a list of configurations of each annotation.

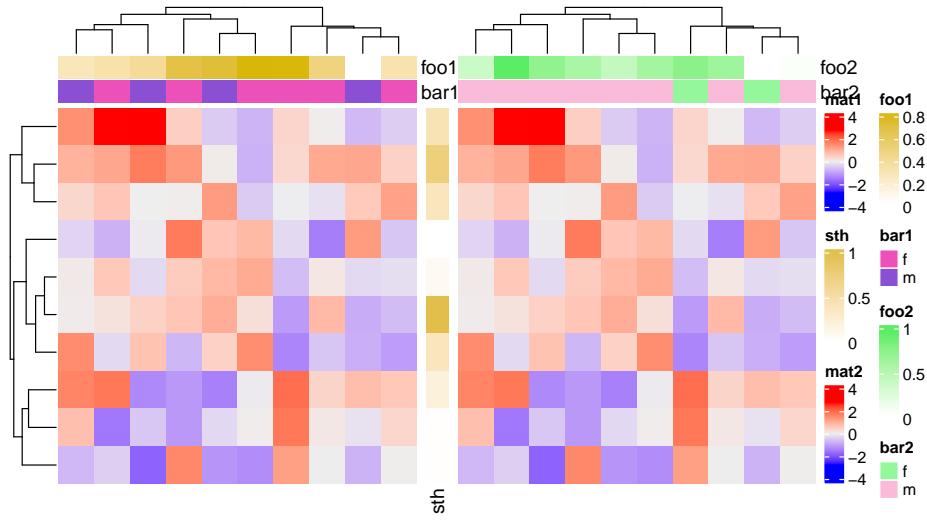
```
ha = HeatmapAnnotation(foo = runif(10), bar = sample(c("f", "m"), 10, replace = TRUE),
annotation_legend_param = list(
  foo = list(
    title = "Fooooooh",
    at = c(0, 0.5, 1),
    labels = c("zero", "median", "one")
  ),
  bar = list(
    title = "Baaaaaaaar",
    at = c("f", "m"),
    labels = c("Female", "Male")
  )
))
Heatmap(m, name = "mat", top_annotation = ha)
```



If the heatmaps are concatenated horizontally, all heatmap and row annotation legends are grouped and all column annotation legends are grouped. The reason

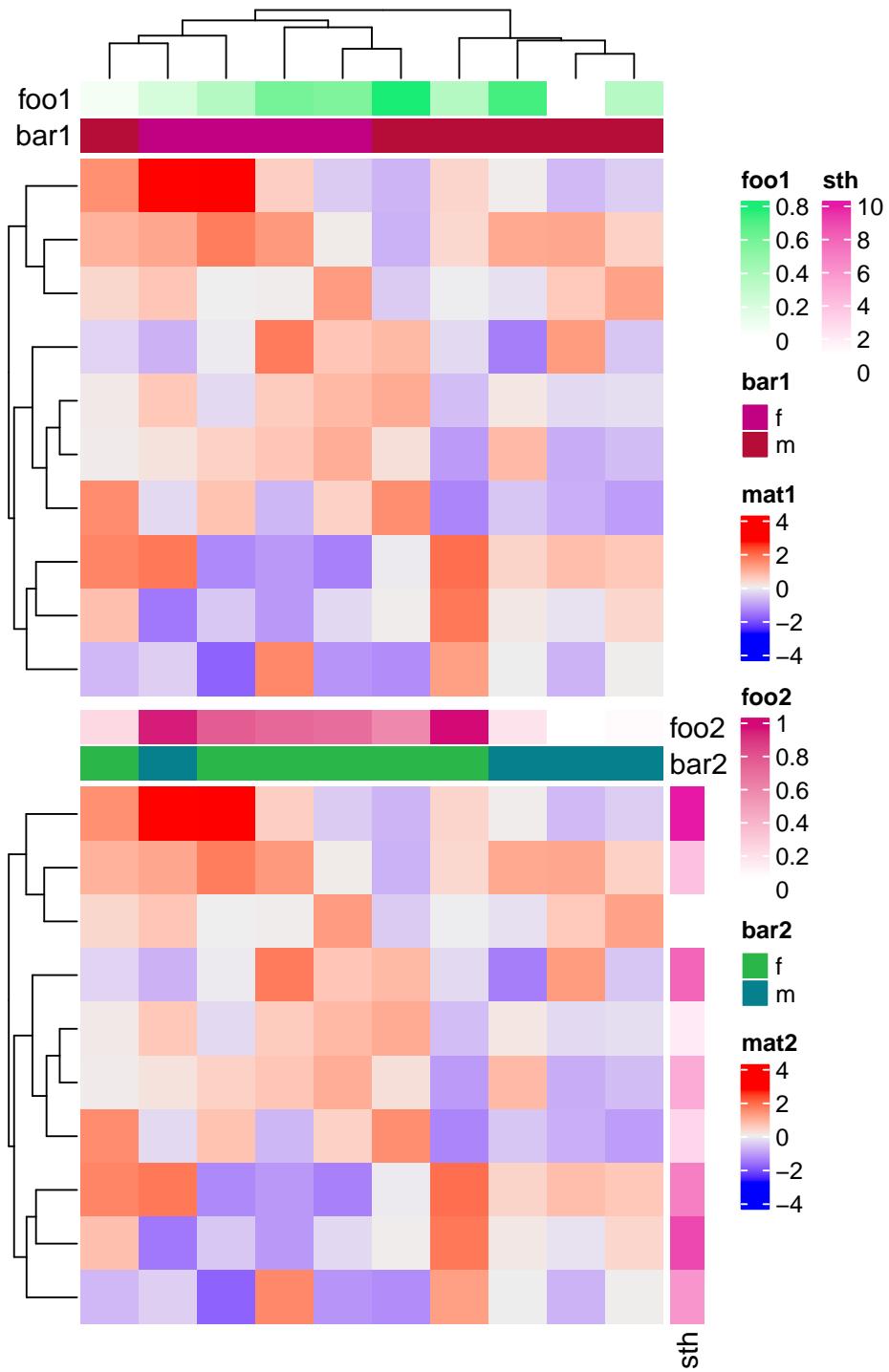
we assume the horizontal direction passes the main message of the plot, while the vertical direction provides secondary information.

```
ha1 = HeatmapAnnotation(foo1 = runif(10), bar1 = sample(c("f", "m"), 10, replace = TRUE)
ha2 = HeatmapAnnotation(foo2 = runif(10), bar2 = sample(c("f", "m"), 10, replace = TRUE)
Heatmap(m, name = "mat1", top_annotation = ha1) +
rowAnnotation(sth = runif(10)) +
Heatmap(m, name = "mat2", top_annotation = ha2)
```



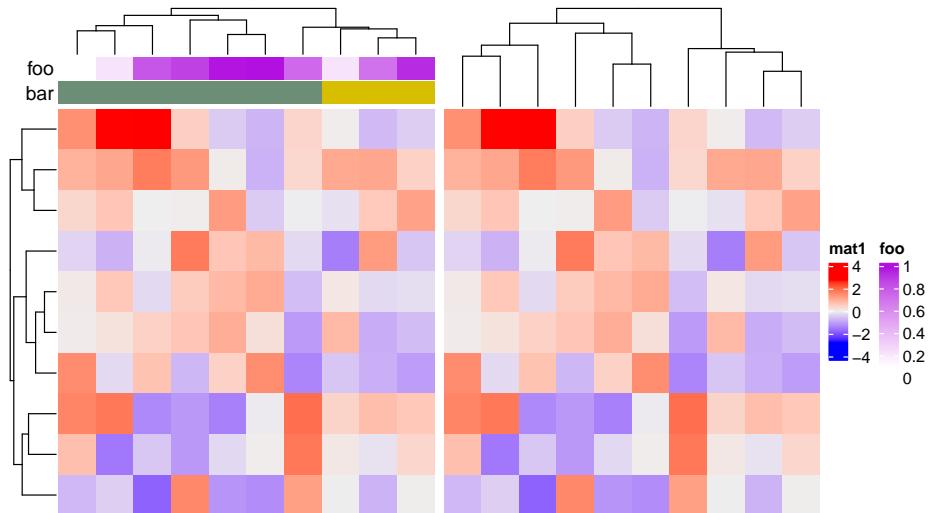
Similarly, if the heatmaps are concatenated vertically, all heatmaps/column annotations are grouped and legends for all row annotations are grouped.

```
ha1 = HeatmapAnnotation(foo1 = runif(10), bar1 = sample(c("f", "m"), 10, replace = TRUE,
annotation_name_side = "left")
ha2 = HeatmapAnnotation(foo2 = runif(10), bar2 = sample(c("f", "m"), 10, replace = TRUE)
Heatmap(m, name = "mat1", top_annotation = ha1) %v%
Heatmap(m, name = "mat2", top_annotation = ha2,
right_annotation = rowAnnotation(sth = 1:10))
```



`show_legend` in `HeatmapAnnotation()` and `show_heatmap_legend` in `Heatmap()` controls whether show the legends. Note `show_legend` can be a single logical value, a logical vector, or a named vector which controls subset of annotations.

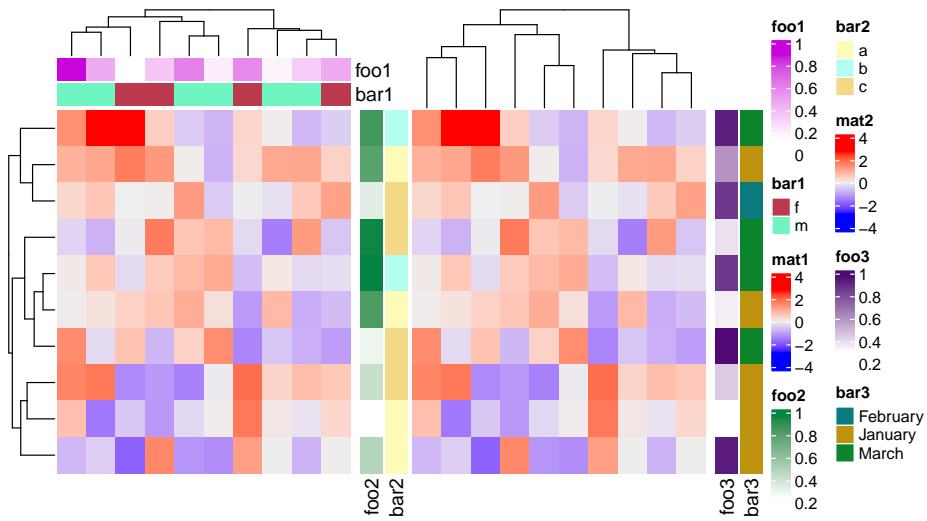
```
ha = HeatmapAnnotation(foo = runif(10),
                      bar = sample(c("f", "m"), 10, replace = TRUE),
                      show_legend = c(TRUE, FALSE), # it can also be show_legend = c(bar = FALSE)
                      annotation_name_side = "left")
Heatmap(m, name = "mat1", top_annotation = ha) +
Heatmap(m, name = "mat2", show_heatmap_legend = FALSE)
```



`merge_legend` in `draw()` function controls whether to merge all the legends into a single group. Normally, when there are many annotations and heatmaps, the number of legends is always large. In this case, the legends are automatically arranged into multiple columns (or multiple rows if they are put at the bottom of the heatmaps) to get rid of being out of the figure page. If a heatmap has heatmap annotations, the order of putting legends are: legends for the left annotations, legends for the top annotations, legend of the heatmap, legends for the bottom annotations and legends for the right annotations.

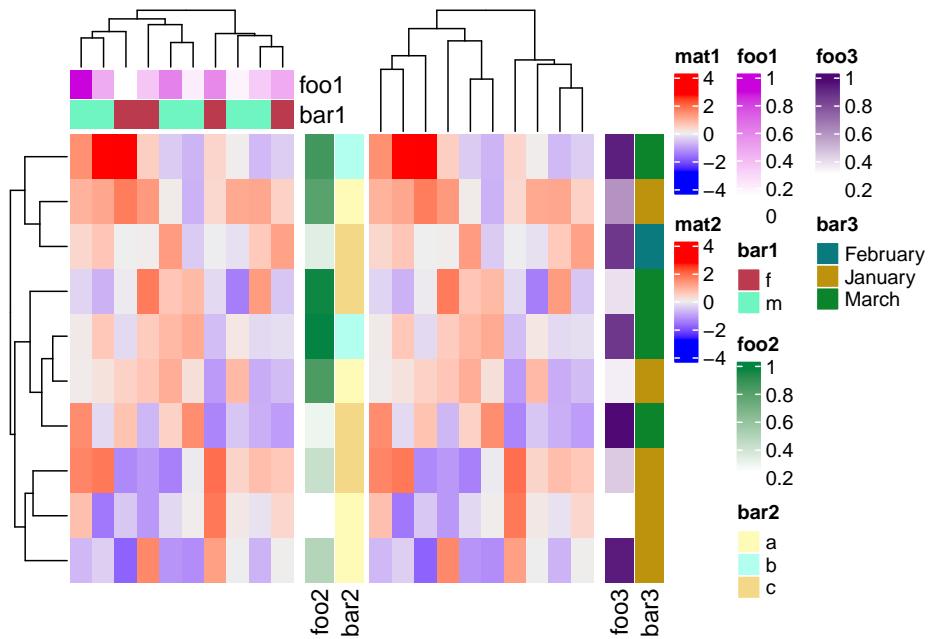
```
ha1 = HeatmapAnnotation(foo1 = runif(10),
                       bar1 = sample(c("f", "m"), 10, replace = TRUE))
ha2 = rowAnnotation(foo2 = runif(10),
                     bar2 = sample(letters[1:3], 10, replace = TRUE))
ha3 = rowAnnotation(foo3 = runif(10),
                     bar3 = sample(month.name[1:3], 10, replace = TRUE))
ht_list = Heatmap(m, name = "mat1", top_annotation = ha1) +
Heatmap(m, name = "mat2", left_annotation = ha2) +
```

```
ha3
draw(ht_list, merge_legend = TRUE)
```

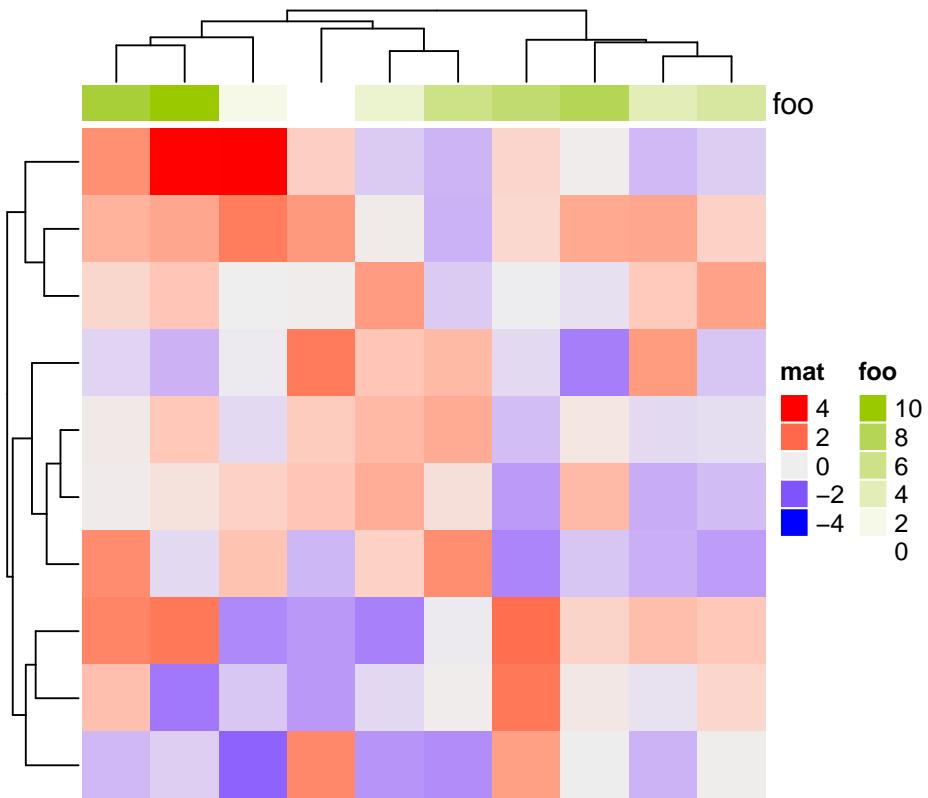


If you want the heatmap legends to be the “pure heatmap legends,” you can set `legend_grouping = "original"` to enforce all annotation legends to be put together, no matter whether they are row annotation legends or column annotation legends.

```
draw(ht_list, legend_grouping = "original")
```



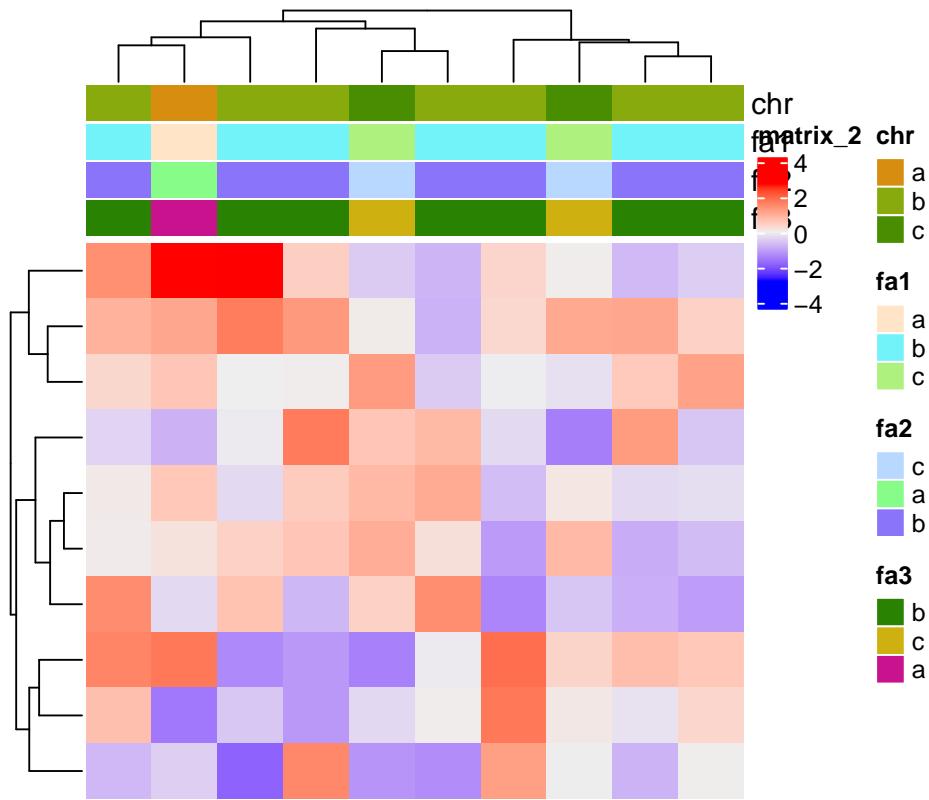
A continuous color mapping can have a discrete legend by setting `color_bar = "discrete"`, both work for heatmap legends and annotation legends.



If the `value` is a character vector, no matter it is an annotation or the one-row/one-column matrix for the heatmap, the default order of legend labels is `sort(unique(value))` and if `value` is a factor, the order of legend labels is `levels(value)`. Always remember the order can be fine-tuned by setting `at` and `labels` parameters in `heatmap_legend_param/annotation_legend_param` in `Heatmap()`/`HeatmapAnnotation()` functions respectively.

```
chr = sample(letters[1:3], 10, replace = TRUE)
chr
```

```
## [1] "b" "b" "c" "b" "b" "b" "b" "c" "b" "a"
fa1 = factor(chr)
fa2 = factor(chr, levels = c("c", "a", "b"))
Heatmap(m, top_annotation = HeatmapAnnotation(chr = chr, fa1 = fa1, fa2 = fa2, fa3 = fa2,
annotation_legend_param = list(fa3 = list(at = c("b", "c", "a")))))
```



5.5 Add customized legends

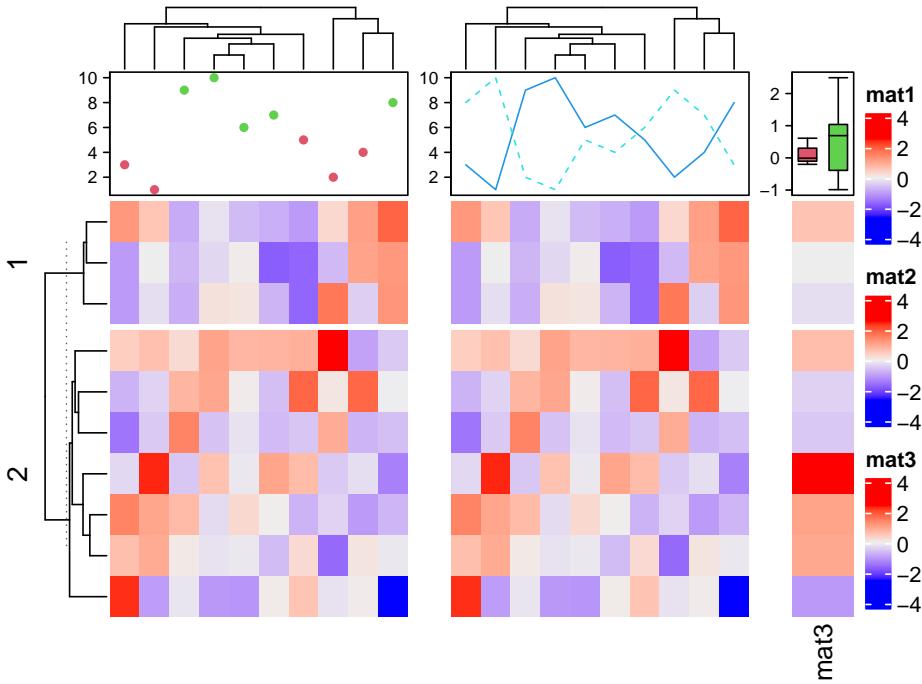
The self-defined legends (constructed by `Legend()`) can be added to the heatmap legend list by `heatmap_legend_list` argument in `draw()` and the legends for annotations can be added to the annotation legend list by `annotation_legend_list` argument.

There is a nice example of adding self-defined legends in Section 11.2, but here we show a simple example.

As mentioned before, only the heatmap and simple annotations can generate legends on the plot. **ComplexHeatmap** provides many annotation functions, but none of them supports generating legends. In following code, we add a point annotation, a line annotation and a summary annotation to the heatmaps.

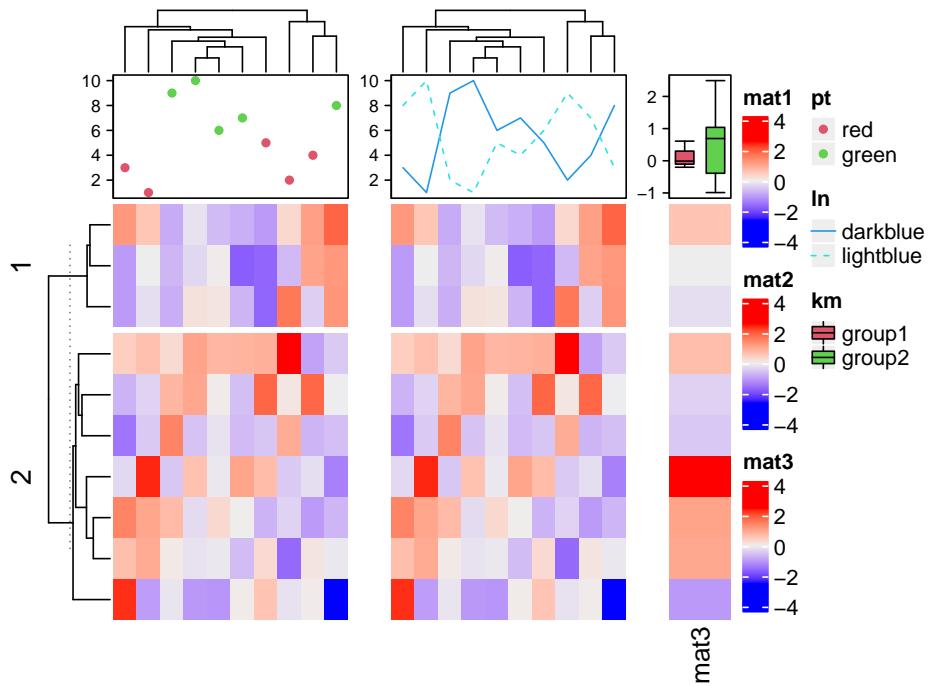
```
ha1 = HeatmapAnnotation(pt = anno_points(1:10, gp = gpar(col = rep(2:3, each = 5)),
height = unit(2, "cm")), show_annotation_name = FALSE)
ha2 = HeatmapAnnotation(ln = anno_lines(cbind(1:10, 10:1), gp = gpar(col = 4:5, lty =
height = unit(2, "cm")), show_annotation_name = FALSE)
m = matrix(rnorm(100), 10)
```

```
ht_list = Heatmap(m, name = "mat1", top_annotation = ha1) +
  Heatmap(m, name = "mat2", top_annotation = ha2) +
  Heatmap(m[, 1], name = "mat3",
    top_annotation = HeatmapAnnotation(
      summary = anno_summary(gp = gpar(fill = 2:3)))
  ), width = unit(1, "cm"))
draw(ht_list, ht_gap = unit(7, "mm"), row_km = 2)
```



Next we construct legends for the points, the lines and the boxplots.

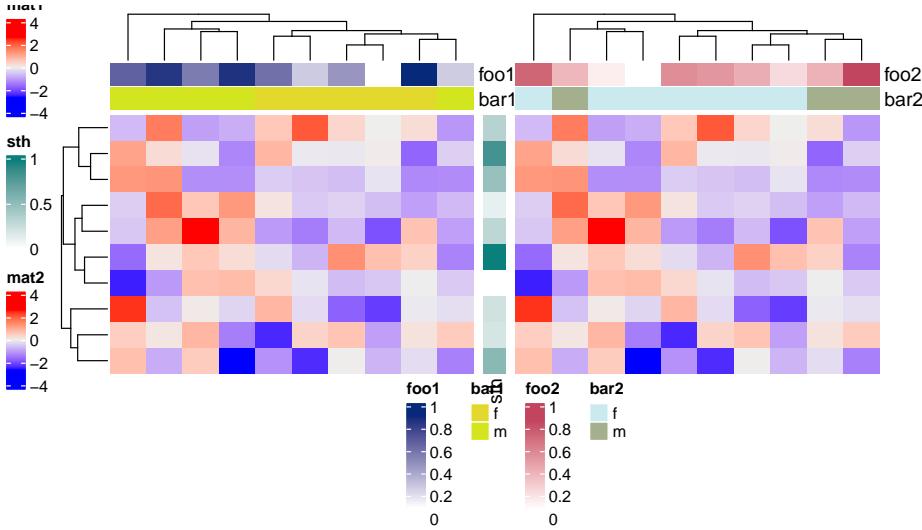
```
lgd_list = list(
  Legend(labels = c("red", "green"), title = "pt", type = "points", pch = 16,
    legend_gp = gpar(col = 2:3)),
  Legend(labels = c("darkblue", "lightblue"), title = "ln", type = "lines",
    legend_gp = gpar(col = 4:5, lty = 1:2)),
  Legend(labels = c("group1", "group2"), title = "km", type = "boxplot",
    legend_gp = gpar(fill = 2:3))
)
draw(ht_list, ht_gap = unit(7, "mm"), row_km = 2, annotation_legend_list = lgd_list)
```



5.6 The side of legends

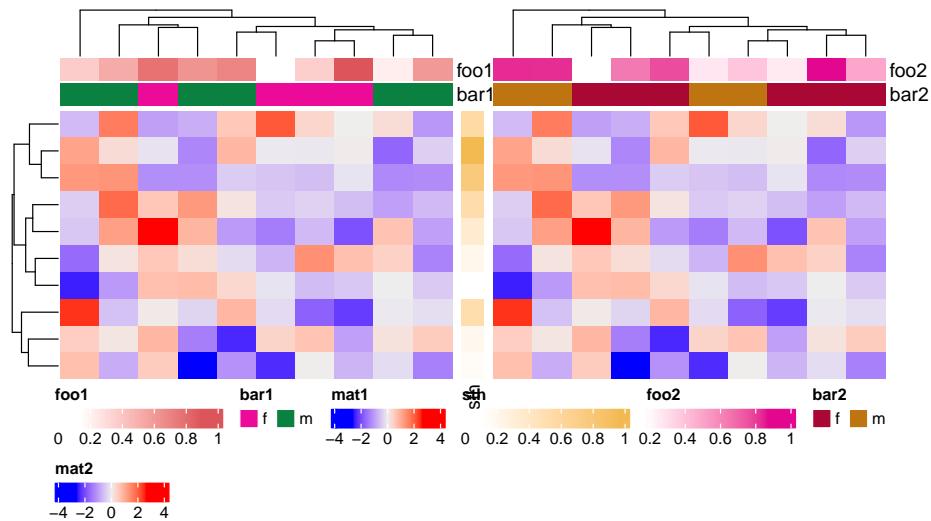
By default, the heatmap legends and annotation legends are put on the right of the plot. The side relative to the heatmaps of the two types of legends can be controlled by `heatmap_legend_side` and `annotation_legend_side` arguments in `draw()` function. The values that can be set for the two arguments are `left`, `right`, `bottom` and `top`.

```
m = matrix(rnorm(100), 10)
ha1 = HeatmapAnnotation(foo1 = runif(10), bar1 = sample(c("f", "m"), 10, replace = TRUE),
ha2 = HeatmapAnnotation(foo2 = runif(10), bar2 = sample(c("f", "m"), 10, replace = TRUE),
ht_list = Heatmap(m, name = "mat1", top_annotation = ha1) +
  rowAnnotation(sth = runif(10)) +
  Heatmap(m, name = "mat2", top_annotation = ha2)
draw(ht_list, heatmap_legend_side = "left", annotation_legend_side = "bottom")
```



When the legends are put at the bottom or on the top, the legends are arranged horizontally. We might also want to set every single legend as horizontal legend, this needs to be set via the `heatmap_legend_param` and `annotation_legend_param` arguments in `Heatmap()` and `HeatmapAnnotation()` functions:

```
ha1 = HeatmapAnnotation(foo1 = runif(10), bar1 = sample(c("f", "m"), 10, replace = TRUE),
  annotation_legend_param = list(
    foo1 = list(direction = "horizontal"),
    bar1 = list(nrow = 1)))
ha2 = HeatmapAnnotation(foo2 = runif(10), bar2 = sample(c("f", "m"), 10, replace = TRUE),
  annotation_legend_param = list(
    foo2 = list(direction = "horizontal"),
    bar2 = list(nrow = 1)))
ht_list = Heatmap(m, name = "mat1", top_annotation = ha1,
  heatmap_legend_param = list(direction = "horizontal")) +
  rowAnnotation(sth = runif(10),
    annotation_legend_param = list(sth = list(list(direction = "horizontal")))) +
  Heatmap(m, name = "mat2", top_annotation = ha2,
    heatmap_legend_param = list(direction = "horizontal"))
draw(ht_list, merge_legend = TRUE, heatmap_legend_side = "bottom",
  annotation_legend_side = "bottom")
```



Chapter 6

Heatmap Decoration

The plotting region for each heatmap component is still kept after the heatmaps are made, so it is possible to go back to the original places to add more graphics there. First let's generate a figure that almost contains all types of heatmap components. `list_components()` lists the names of the heatmap/annotation components (or the name of the viewport).

```
set.seed(123)
mat = matrix(rnorm(80, 2), 8, 10)
mat = rbind(mat, matrix(rnorm(40, -2), 4, 10))
rownames(mat) = paste0("R", 1:12)
colnames(mat) = paste0("C", 1:10)

ha_column1 = HeatmapAnnotation(points = anno_points(rnorm(10)),
  annotation_name_side = "left")
ht1 = Heatmap(mat, name = "ht1", km = 2, column_title = "Heatmap 1",
  top_annotation = ha_column1, row_names_side = "left")

ha_column2 = HeatmapAnnotation(type = c(rep("a", 5), rep("b", 5)),
  col = list(type = c("a" = "red", "b" = "blue")))
ht2 = Heatmap(mat, name = "ht2", row_title = "Heatmap 2", column_title = "Heatmap 2",
  bottom_annotation = ha_column2, column_km = 2)

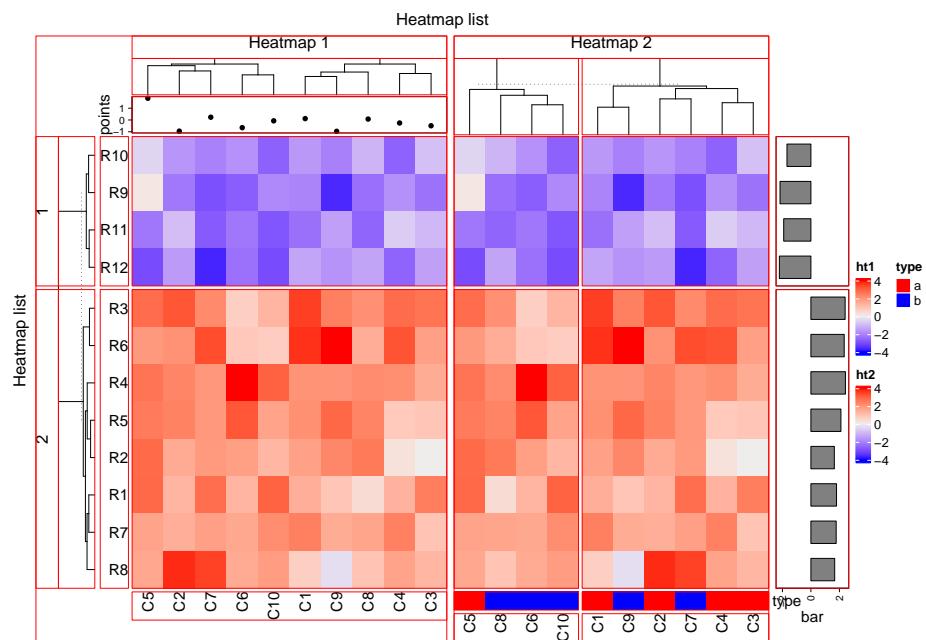
ht_list = ht1 + ht2 +
  rowAnnotation(bar = anno_barplot(rowMeans(mat), width = unit(2, "cm")))
draw(ht_list, row_title = "Heatmap list", column_title = "Heatmap list")
list_components()

## [1] "ROOT"                               "global"
## [3] "global_layout"                      "global-heatmaplist"
## [5] "main_heatmap_list"                  "heatmap_ht1"
```

```

## [7] "ht1_heatmap_body_wrap"
## [9] "ht1_heatmap_body_2_1"
## [11] "ht1_row_title_1"
## [13] "ht1_dend_row_1"
## [15] "ht1_dend_column_1"
## [17] "ht1_row_names_2"
## [19] "annotation_points_1"
## [21] "ht2_heatmap_body_wrap"
## [23] "ht2_heatmap_body_1_2"
## [25] "ht2_heatmap_body_2_2"
## [27] "ht2_dend_column_1"
## [29] "ht2_column_names_1"
## [31] "annotation_type_1"
## [33] "heatmap_annotation_2"
## [35] "annotation_bar_2"
## [37] "global_column_title"
## [39] "global_row_title"
## [41] "heatmap_legend"
## [43] "annotation_legend"
## [45] "ht1_heatmap_body_1_1"
## [47] "ht1_column_title_1"
## [49] "ht1_row_title_2"
## [51] "ht1_dend_row_2"
## [53] "ht1_row_names_1"
## [55] "ht1_column_names_1"
## [57] "heatmap_ht2"
## [59] "ht2_heatmap_body_1_1"
## [61] "ht2_heatmap_body_2_1"
## [63] "ht2_column_title_1"
## [65] "ht2_dend_column_2"
## [67] "ht2_column_names_2"
## [69] "annotation_type_2"
## [71] "annotation_bar_1"
## [73] "global-column_title_top"
## [75] "global-row_title_left"
## [77] "global-heatmap_legend_right"
## [79] "global-annotation_legend_right"

```



Basically the red regions in above plot can be revisited by `decorate_*`() functions.

6.1 Decoration functions

Since you can get the viewport name by `list_components()`, actually you can directly go to the viewport by `seekViewport()`. To get rid of the complicated viewport names, the `decorate_*`() functions provide a more friendly way to do it.

There are following decoration functions in **ComplexHeatmap** package:

- `decorate_heatmap_body()`
- `decorate_annotation()`
- `decorate_dend()`
- `decorate_title()`
- `decorate_dimnames()`
- `decorate_row_names()`, identical to `decorate_dimnames(..., which = "row")`.
- `decorate_column_names()`, identical to `decorate_dimnames(..., which = "column")`.
- `decorate_row_dend()`, identical to `decorate_dend(..., which = "row")`.
- `decorate_column_dend()`, identical to `decorate_dend(..., which = "column")`.
- `decorate_row_title()`, identical to `decorate_title(..., which = "row")`.
- `decorate_column_title()`, identical to `decorate_title(..., which = "column")`.

Among them, `decorate_heatmap_body()` and `decorate_annotation()` are more often used.

For all these functions, they need a heatmap or annotation name, index for the row/column slices if the heatmap is split and a code block which defines how to add graphics. Check following example.

```
ht_list = draw(ht_list, row_title = "Heatmap list", column_title = "Heatmap list",
               heatmap_legend_side = "right", annotation_legend_side = "left")

decorate_heatmap_body("ht1", {
  grid.text("outlier", 1.5/10, 2.5/4, default.units = "npc")
  grid.lines(c(0.5, 0.5), c(0, 1), gp = gpar(lty = 2, lwd = 2))
}, slice = 2)

decorate_column_dend("ht1", {
  tree = column_dend(ht_list)$ht1[[1]]
  ind = cutree(as.hclust(tree), k = 2)[order.dendrogram(tree)]

  first_index = function(l) which(l)[1]
  last_index = function(l) { x = which(l); x[length(x)] }
})
```

```

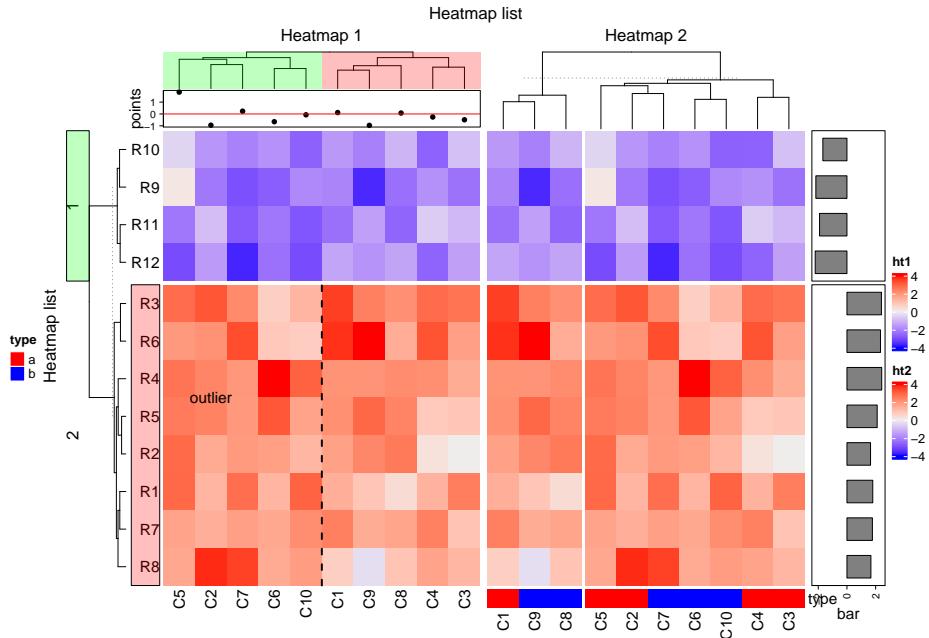
x1 = c(first_index(ind == 1), first_index(ind == 2)) - 1
x2 = c(last_index(ind == 1), last_index(ind == 2))
grid.rect(x = x1/length(ind), width = (x2 - x1)/length(ind), just = "left",
          default.units = "npc", gp = gpar(fill = c("#FF000040", "#00FF0040"), col = NA))
})

decorate_row_names("ht1", {
  grid.rect(gp = gpar(fill = "#FF000040"))
}, slice = 2)

decorate_row_title("ht1", {
  grid.rect(gp = gpar(fill = "#00FF0040"))
}, slice = 1)

decorate_annotation("points", {
  grid.lines(c(0, 1), unit(c(0, 0), "native"), gp = gpar(col = "red"))
})

```



For annotations which are created by `anno_points()`, `anno_barplot()` and `anno_boxplot()`, “native” unit can be used in the decoration code.

6.2 Examples

6.2.1 Barplot for single-column heatmap

In Section 3.20, we introduced adding barplots as annotations for single-column heatmap. In that case the heatmap contains discrete values where the barplots show the frequency of each level. In following example, we show another scenario of using barplot as annotation but for a continuous heatmap.

Imagining we are analyzing a set of genomic regions (e.g. differentially methylated regions, DMRs) and we have a single-column heatmap which shows the overlap to e.g. genes (measured by the fraction of a DMR covered by genes, value between 0 and 1, e.g. a value of 0.5 means 50% of this DMR overlaps to the genes). If we denote the width of DMRs as w and the fraction as p , on top of the fraction heatmap, we want to add barplots to show, on average, how much of the DMRs are covered by genes. In this case, we need to calcualte the mean fraction weighted by the width of DMRs ($\sum (w \cdot p) / \sum w$).

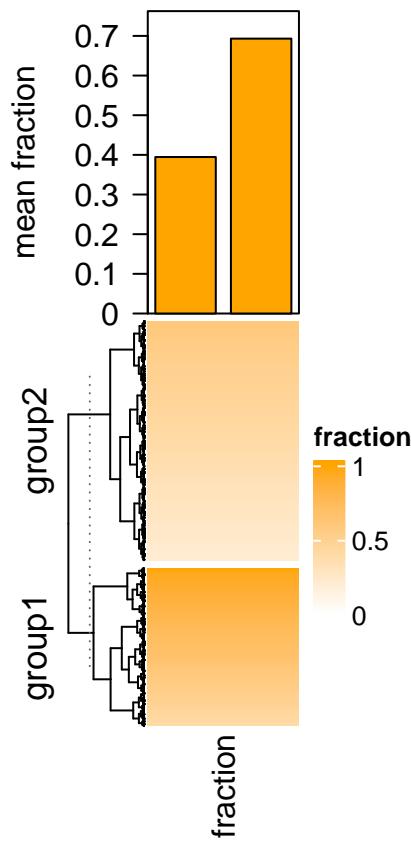
In following code, we randomly generated a fraction vector and split it into two groups. We first use `anno_empty()` to allocate empty plotting region on top of the heatmap and later we use `decorate_annotation()` to add the barplots into it.

```
library(circlize)
# DMRs
bed = generateRandomBed(nr = 1000)
# fractions
frac = c(runif(400, min = 0.4, max = 1), runif(nrow(bed) - 400, min = 0.2, max = 0.6))
col_fun = colorRamp2(c(0, 1), c("white", "orange"))
# two groups
split = c(rep("group1", 400), rep("group2", nrow(bed) - 400))
# draw the fraction heatmap with an empty annotation
ht = Heatmap(frac, name = "fraction", col = col_fun, width = unit(2, "cm"),
             top_annotation = HeatmapAnnotation(barplot = anno_empty(height = unit(4, "cm"))))
ht = draw(ht, row_split = split)
# get the row indices in the two row-groups
ro = row_order(ht)
w = bed[, 3] - bed[, 2]
# the mean weighted fraction in the two groups
p = sapply(ro, function(index) {
  sum(w[index]*frac[index])/sum(w[index])
})
# add two bars of `p`
decorate_annotation("barplot", {
  pushViewport(viewport(xscale = c(0.5, 2.5), yscale = c(0, max(p)*1.1)))
  grid.rect(x = 1:2, y = 0, width = 0.8, height = p, just = "bottom",
            gp = gpar(fill = "orange"), default.units = "native")})
```

```

grid.yaxis()
grid.text("mean fraction", x = unit(-1.5, "cm"), rot = 90, just = "bottom")
popViewport()
})

```



6.2.2 Add titles for row annotations

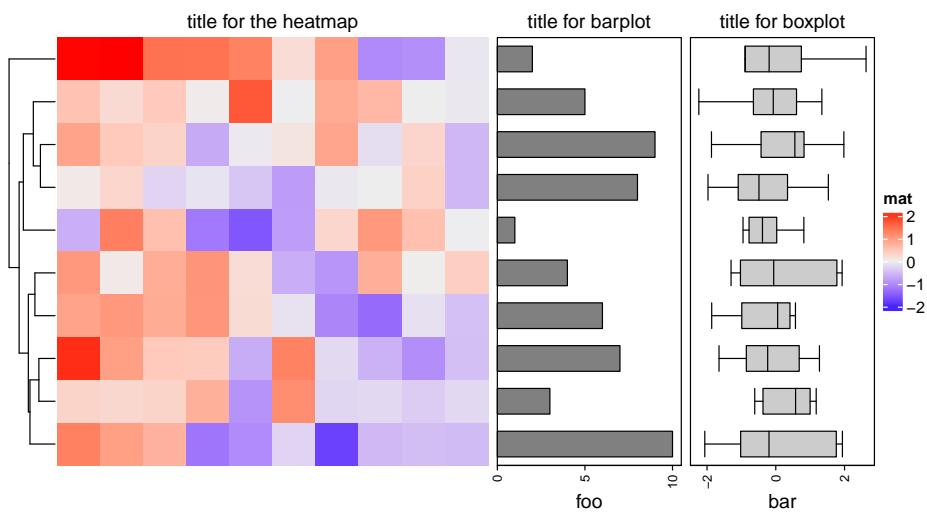
Row annotations can be concatenated to the heatmap list. Sometimes we need a title for the row annotation. It is easy to implement it by decorations.

```

ht_list = Heatmap(matrix(rnorm(100), 10), name = "mat", show_column_dend = FALSE) +
  rowAnnotation(foo = anno_barplot(1:10, width = unit(4, "cm"))) +
  rowAnnotation(bar = anno_boxplot(matrix(rnorm(100), 10)), width = unit(4, "cm"))
draw(ht_list, padding = unit(c(2, 2, 10, 2), "mm")) # add space for titles
decorate_annotation("foo", {
  grid.text("title for barplot", y = unit(1, "npc") + unit(2, "mm"), just = "bottom")
})
decorate_annotation("bar", {
  grid.text("title for boxplot", y = unit(1, "npc") + unit(2, "mm"), just = "bottom")
})

```

```
})
decorate_heatmap_body("mat", {
  grid.text("title for the heatmap", y = unit(1, "npc") + unit(2, "mm"), just = "bottom")
})
```



This is basically the way we use in Section 11.2.

6.2.3 Other possible use of decorations

There are some other examples where decoration is helpful:

- The quantile lines are added in `densityHeatmap()`, Section 11.1.
- Heatmap annotations are grouped by lines, Section 14.5.
- Texts are added to heatmaps, Section 14.2.

Chapter 7

OncoPrint

OncoPrint is a way to visualize multiple genomic alteration events by heatmap. Here the **ComplexHeatmap** package provides a `oncoPrint()` function which makes oncoPrints. Besides the default style which is provided by cBioPortal, there are additional barplots at both sides of the heatmap which show numbers of different alterations for each sample and for each gene. Also with the functionality of **ComplexHeatmap**, you can concatenate oncoPrints with additional heatmaps and annotations to correspond more types of information.

7.1 General settings

7.1.1 Input data format

There are two different formats of input data. The first is represented as a matrix in which each value can include multiple alterations in a form of a complicated string. In follow example, ‘g1’ in ‘s1’ has two types of alterations which are ‘snv’ and ‘indel’.

```
mat = read.table(textConnection(
  "s1,s2,s3
  g1,snv;indel,snv,indel
  g2,,snv;indel,snv
  g3,snv,,indel;snv"), row.names = 1, header = TRUE, sep = ",",
  stringsAsFactors = FALSE)
mat = as.matrix(mat)
mat

##      s1          s2          s3
## g1 "snv;indel" "snv"      "indel"
## g2 ""          "snv;indel" "snv"
## g3 "snv"       ""         "indel;snv"
```

In this case, we need to define a function to extract different alteration types from these long strings. The definition of such function is always simple, it accepts the complicated string and returns a vector of alteration types.

For `mat`, we can define the function as:

```
get_type_fun = function(x) strsplit(x, ";")[[1]]
get_type_fun(mat[1, 1])

## [1] "snv"    "indel"
get_type_fun(mat[1, 2])

## [1] "snv"
```

So, if the alterations are encoded as `snv|indel`, you can define the function as `function(x) strsplit(x, "|")[[1]]`. This self-defined function is assigned to the `get_type` argument in `oncoPrint()`.

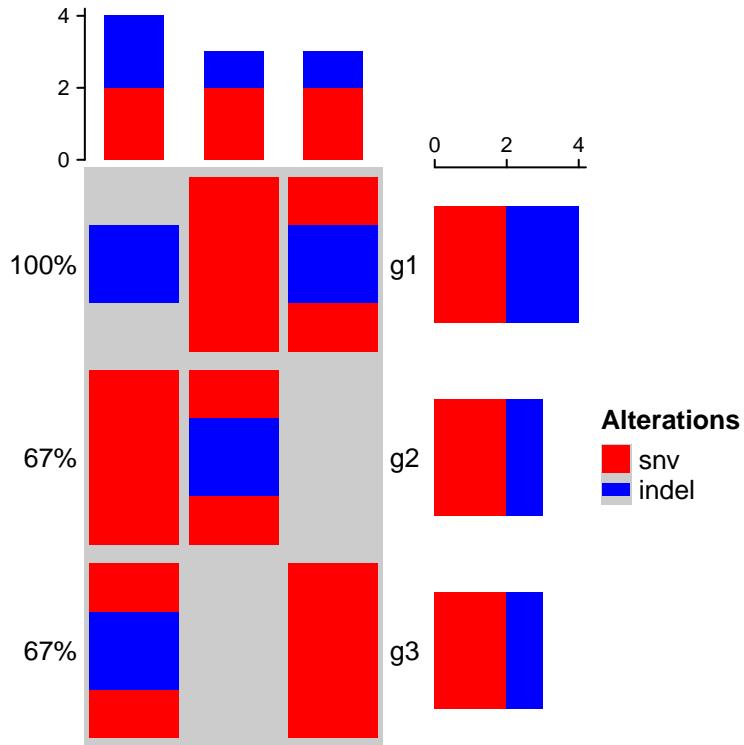
Since in most cases, the separators are only single characters, If the separators are in `;:,|`, `oncoPrint()` automatically spit the alteration strings so that you don't need to explicitly specify `get_type` in `oncoPrint()` function.

For one gene in one sample, since different alteration types may be drawn into one same grid in the heatmap, we need to define how to add the graphics by providing a list of self-defined functions to `alter_fun` argument. Here if the graphics have no transparency, order of adding graphics matters. In following example, snv are first drawn and then the indel. You can see rectangles for indels are actually smaller ($0.4*h$) than that for snvs ($0.9*h$) so that you can visualize both snvs and indels if they are in a same grid. Names of the function list should correspond to the alteration types (here, `snv` and `indel`).

For the self-defined graphic function (the functions in `alter_fun`, there should be four arguments which are positions of the grids on the `oncoPrint` (`x` and `y`), and widths and heights of the grids (`w` and `h`, which is measured in `npc` unit). Proper values for the four arguments are sent to these functions automatically from `oncoPrint()`.

Colors for different alterations are defined in `col`. It should be a named vector for which names correspond to alteration types. It is used to generate the barplots.

```
col = c(snv = "red", indel = "blue")
oncoPrint(mat,
          alter_fun = list(
            snv = function(x, y, w, h) grid.rect(x, y, w*0.9, h*0.9,
                                                gp = gpar(fill = col["snv"], col = NA)),
            indel = function(x, y, w, h) grid.rect(x, y, w*0.9, h*0.4,
                                                 gp = gpar(fill = col["indel"], col = NA))
          ), col = col)
```



You can see the order in barplots also correspond to the order defined in `alter_fun`. The graphics in legend are based on the functions defined in `alter_fun`.

If you are confused of how to generated the matrix, there is a second way. The second type of input data is a list of matrix for which each matrix contains binary value representing whether the alteration is absent or present. The list should have names which correspond to the alteration types.

```
mat_list = list(snv = matrix(c(1, 0, 1, 1, 1, 0, 0, 1, 1), nrow = 3),
                indel = matrix(c(1, 0, 0, 0, 1, 0, 1, 0, 0), nrow = 3))
rownames(mat_list$snv) = rownames(mat_list$indel) = c("g1", "g2", "g3")
colnames(mat_list$snv) = colnames(mat_list$indel) = c("s1", "s2", "s3")
mat_list

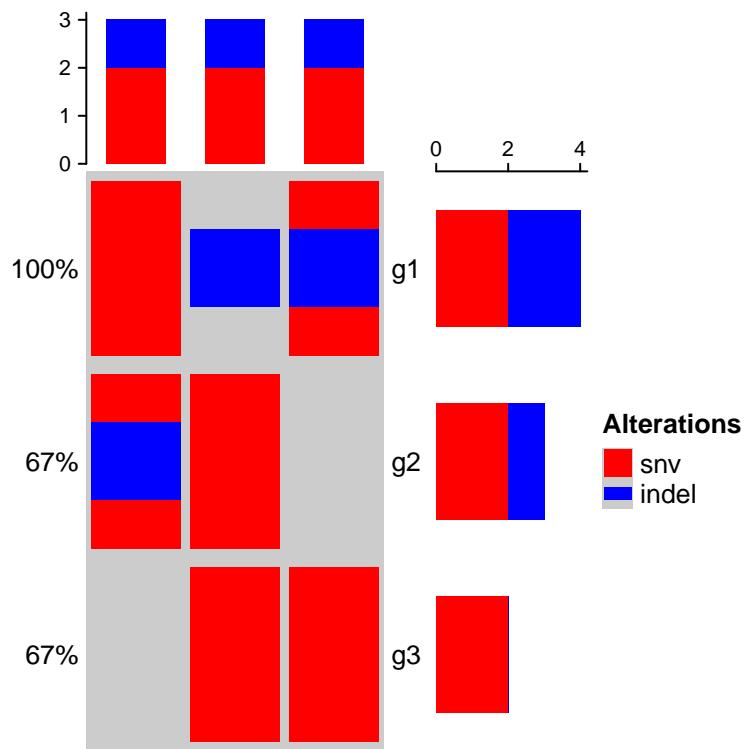
## $snv
##   s1 s2 s3
## g1  1  1  0
## g2  0  1  1
## g3  1  0  1
##
## $indel
##   s1 s2 s3
```

```
## g1  1  0  1
## g2  0  1  0
## g3  0  0  0
```

`oncoPrint()` expects all matrices in `mat_list` having same row names and column names.

Pass `mat_list` to `oncoPrint()`:

```
# now you don't need `get_type`
oncoPrint(mat_list,
          alter_fun = list(
            snv = function(x, y, w, h) grid.rect(x, y, w*0.9, h*0.9,
                                                gp = gpar(fill = col["snv"], col = NA)),
            indel = function(x, y, w, h) grid.rect(x, y, w*0.9, h*0.4,
                                                gp = gpar(fill = col["indel"], col = NA))
          ), col = col)
```



In following parts of this chapter, we still use the single matrix form `mat` to specify the input data.

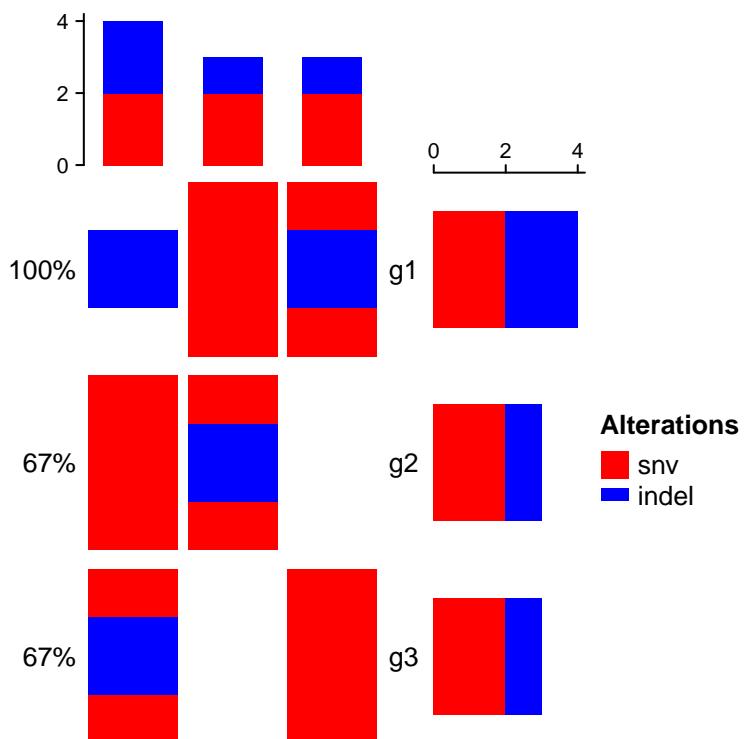
7.1.2 Define the alter_fun()

`alter_fun` is a list of functions which add graphics layer by layer (i.e. first draw for `snv`, then for `indel`). Graphics can also be added in a grid-by-grid style by specifying `alter_fun` as a single function. The difference from the function list is now `alter_fun` should accept a fifth argument which is a logical vector. This logical vector shows whether different alterations exist for current gene in current sample.

Let's assume in a grid there is only `snv` event, then `v` for this grid is:

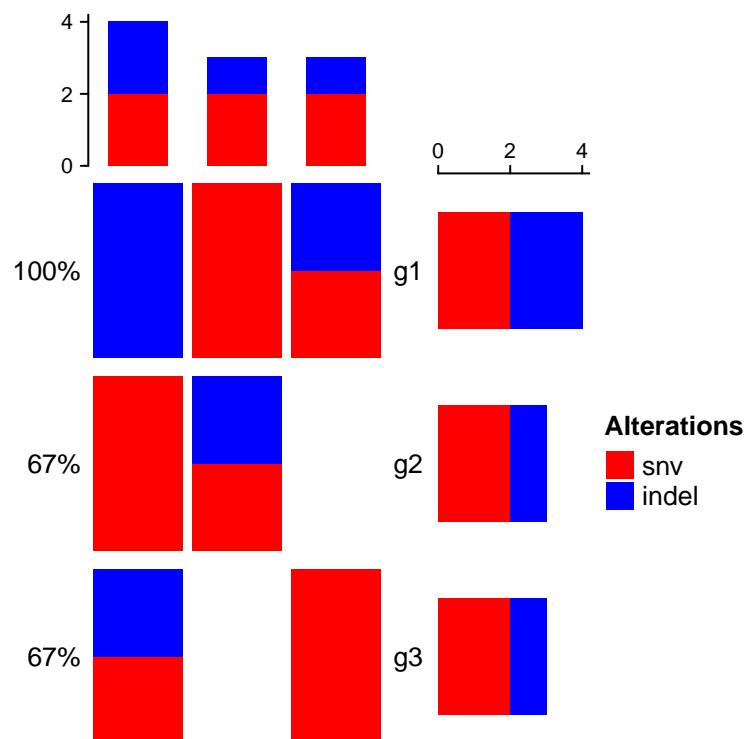
```
## snv indel
## TRUE FALSE

oncoPrint(mat,
  alter_fun = function(x, y, w, h, v) {
    if(v["snv"]) grid.rect(x, y, w*0.9, h*0.9, # v["snv"] is a logical value
      gp = gpar(fill = col["snv"], col = NA))
    if(v["indel"]) grid.rect(x, y, w*0.9, h*0.4, # v["indel"] is a logical value
      gp = gpar(fill = col["indel"], col = NA))
  }, col = col)
```



If `alter_fun` is set as a single function, customization can be more flexible. In following example, the blue rectangles can have different height in different grid.

```
oncoPrint(mat,
  alter_fun = function(x, y, w, h, v) {
    n = sum(v) # how many alterations for current gene in current sample
    h = h*0.9
    # use `names(which(v))` to correctly map between `v` and `col`
    if(n) grid.rect(x, y - h*0.5 + 1:n/n*h, w*0.9, 1/n*h,
      gp = gpar(fill = col[names(which(v))], col = NA), just = "top")
  }, col = col)
```



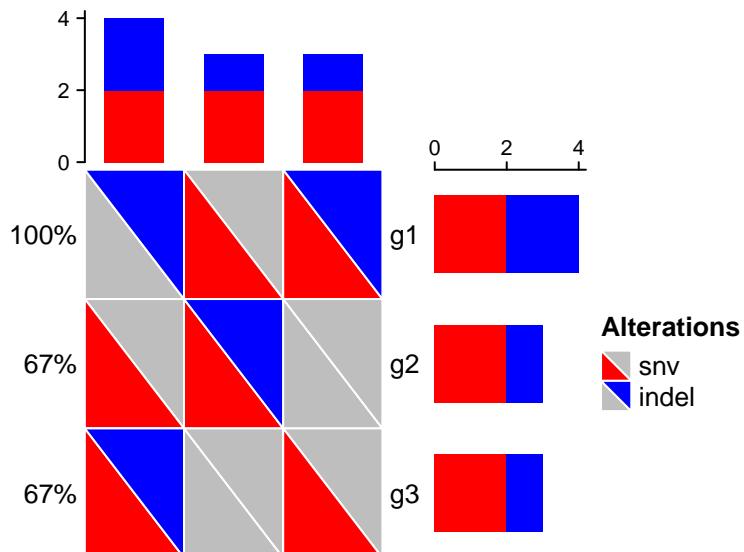
Following is a complicated example for `alter_fun` where triangles are used:

```
oncoPrint(mat,
  alter_fun = list(
    background = function(x, y, w, h) {
      grid.polygon(
        unit.c(x - 0.5*w, x - 0.5*w, x + 0.5*w),
        unit.c(y - 0.5*h, y + 0.5*h, y - 0.5*h),
        gp = gpar(fill = "grey", col = "white"))
      grid.polygon(
        unit.c(x + 0.5*w, x + 0.5*w, x - 0.5*w),
        unit.c(y + 0.5*h, y - 0.5*h, y + 0.5*h),
        gp = gpar(fill = "grey", col = "white"))
```

```

},
snv = function(x, y, w, h) {
  grid.polygon(
    unit.c(x - 0.5*w, x - 0.5*w, x + 0.5*w),
    unit.c(y - 0.5*h, y + 0.5*h, y - 0.5*h),
    gp = gpar(fill = col["snv"], col = "white"))
},
indel = function(x, y, w, h) {
  grid.polygon(
    unit.c(x + 0.5*w, x + 0.5*w, x - 0.5*w),
    unit.c(y + 0.5*h, y - 0.5*h, y + 0.5*h),
    gp = gpar(fill = col["indel"], col = "white"))
}
), col = col)

```



In some cases, you might need to define `alter_fun` for many alteration types. If you are not sure about the visual effect of your `alter_fun`, you can use `test_alter_fun()` to test your `alter_fun`. In following example, we defined seven alteration functions:

```

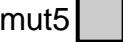
alter_fun = list(
  mut1 = function(x, y, w, h)
    grid.rect(x, y, w, h, gp = gpar(fill = "red", col = NA)),
  mut2 = function(x, y, w, h)
    grid.rect(x, y, w, h, gp = gpar(fill = "blue", col = NA)),
  mut3 = function(x, y, w, h)
    grid.rect(x, y, w, h, gp = gpar(fill = "yellow", col = NA)),
  mut4 = function(x, y, w, h)

```

```

grid.rect(x, y, w, h, gp = gpar(fill = "purple", col = NA)),
mut5 = function(x, y, w, h)
  grid.rect(x, y, w, h, gp = gpar(fill = NA, lwd = 2)),
mut6 = function(x, y, w, h)
  grid.points(x, y, pch = 16),
mut7 = function(x, y, w, h)
  grid.segments(x - w*0.5, y - h*0.5, x + w*0.5, y + h*0.5, gp = gpar(lwd = 2))
)
test.Alter_fun(alter_fun)

## `alter_fun` is defined as a list of functions.
## Functions are defined for following alteration types:
##   mut1, mut2, mut3, mut4, mut5, mut6, mut7

  mut1  mut2+mut6+mut7 
  mut2  mut1+mut6+mut7 
  mut3  mut1+mut2+mut4 
  mut4  mut2+mut4+mut7 
  mut5  mut1+mut2+mut5 
  mut6  mut2+mut4+mut5 
  mut7  mut5+mut6+mut7 

```

For the combination of alteration types, `test.Alter_fun()` randomly samples some of them.

`test.Alter_fun()` works both for `alter_fun` as a list and as a single function.

7.1.3 Background

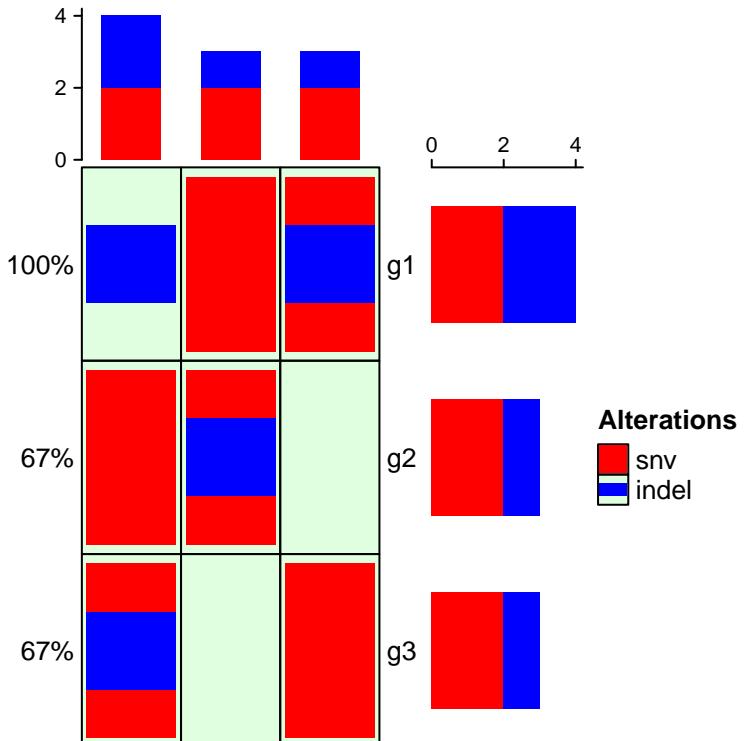
If `alter_fun` is specified as a list, the order of the elements controls the order of adding graphics. There is a special element called `background` which defines how to draw background and it should be always put as the first element in the `alter_fun` list. In following example, background color is changed to light green with borders.

```

oncoPrint(mat,
  alter_fun = list(
    background = function(x, y, w, h) grid.rect(x, y, w, h,
      gp = gpar(fill = "#00FF0020")),
    snv = function(x, y, w, h) grid.rect(x, y, w*0.9, h*0.9,
      gp = gpar(fill = col["snv"], col = NA)),

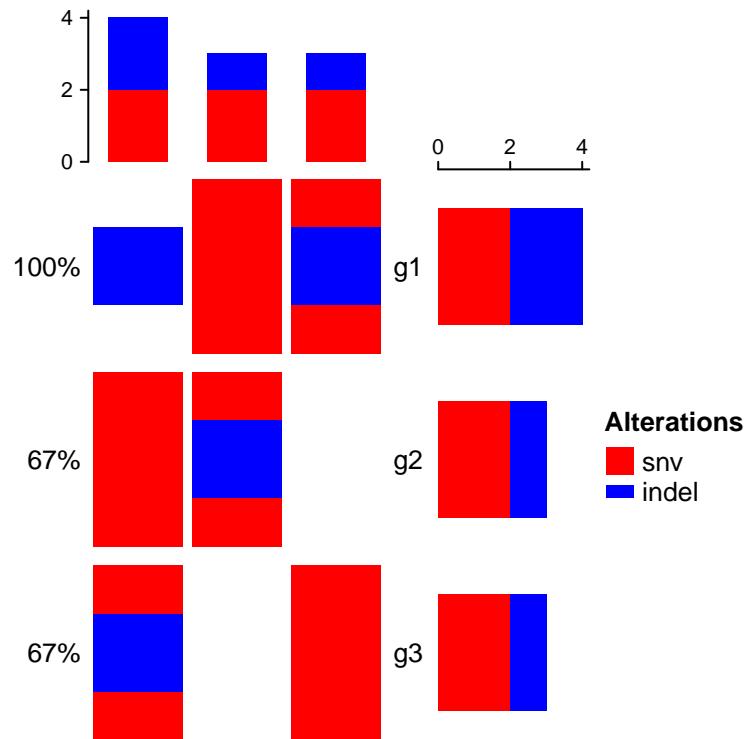
```

```
indel = function(x, y, w, h) grid.rect(x, y, w*0.9, h*0.4,
  gp = gpar(fill = col["indel"], col = NA))
), col = col)
```



Or just remove the background (don't set it to NULL. Setting `background` directly to NULL means to use the default style of background which is in grey):

```
oncoPrint(mat,
  alter_fun = list(
    background = function(...) NULL,
    snv = function(x, y, w, h) grid.rect(x, y, w*0.9, h*0.9,
      gp = gpar(fill = col["snv"], col = NA)),
    indel = function(x, y, w, h) grid.rect(x, y, w*0.9, h*0.4,
      gp = gpar(fill = col["indel"], col = NA))
  ), col = col)
```



7.1.4 Complex alteration types

It is very easy to have many more different alteration types when integrating information from multiple analysis results. It is sometimes difficult to design graphics and assign different colors for them (e.g. see plot in this link. On the other hand, in these alteration types, there are primary classes of alteration types which is more important to distinguish, while there are secondary classes which is less important. For example, we may have alteration types of “intronic snv,” “exonic snv,” “intronic indel” and “exonic indel.” Actually we can classify them into two classes where “snv/indel” is more important and they belong to the primary class, and “intronic/exonic” is less important and they belong to the secondary class. Reflecting on the oncoPrint, for the “intronic snv” and “exonic snv,” we want to use similar graphics because they are snvs and we want them visually similar, and we add slightly different symbols to represent “intronic” and “exonic.” E.g. we can use red rectangle for snv and above the red rectangles, we use dots to represent “intronic” and cross lines to represent “exonic.” On the barplot annotations which summarize the number of different alteration types, we don’t want to separate “intronic snv” and “exonic snv” while we prefer to simply get the total number of snv to get rid of too many categories in the barplots.

Let’s demonstrate this scenario by following simulated data. To simplify the

example, we assume for a single gene in a single sample, it only has either snv or indel and it can only be either intronic or exonic. If there is no “intronic” or “exonic” attached to the gene, it basically means we don’t have this gene-related information (maybe it is an intergenic snv/indel).

```
set.seed(123)
x1 = sample(c("", "snv"), 100, replace = TRUE, prob = c(8, 2))
x2 = sample(c("", "indel"), 100, replace = TRUE, prob = c(8, 2))
x2[x1 == "snv"] = ""
x3 = sample(c("", "intronic"), 100, replace = TRUE, prob = c(5, 5))
x4 = sample(c("", "exonic"), 100, replace = TRUE, prob = c(5, 5))
x3[x1 == "" & x2 == "") = ""
x4[x1 == "" & x2 == "") = ""
x4[x3 == "intronic"] = ""
x = apply(cbind(x1, x2, x3, x4), 1, function(x) {
  x = x[x != ""]
  paste(x, collapse = ";")
})
m = matrix(x, nrow = 10, ncol = 10, dimnames = list(paste0("g", 1:10), paste0("s", 1:10)))
m[1:4, 1:4]

##      s1      s2          s3          s4
## g1 ""    "snv;intronic" "snv;intronic" "snv"
## g2 ""        ""           ""           "snv;intronic"
## g3 ""        ""           ""           ""
## g4 "snv" "indel;exonic" "snv"         "
```

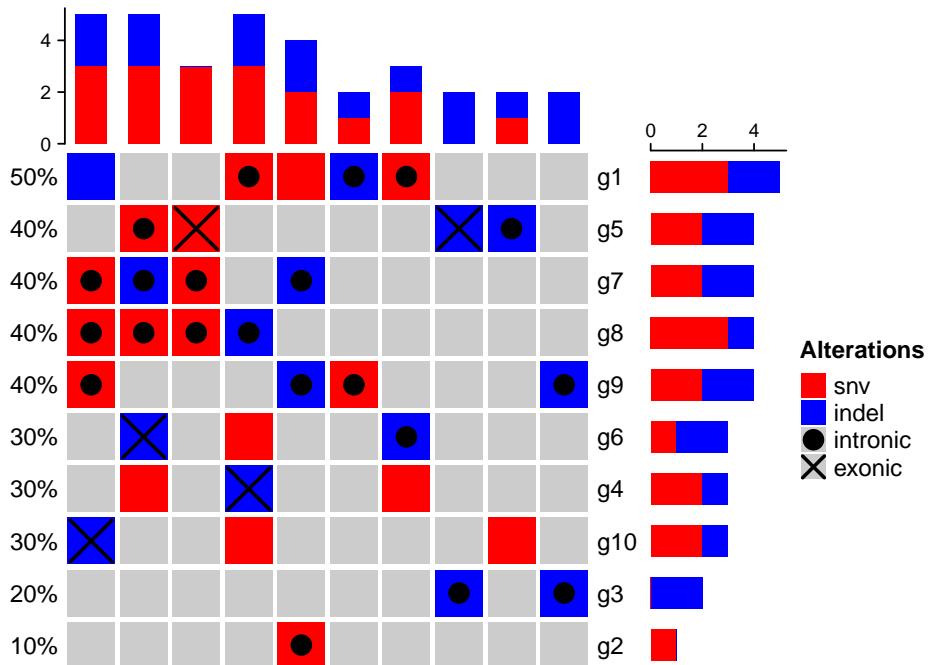
Now in `m`, there are four different alteration types: `snv`, `indel`, `intronic` and `exonic`. Next we define `alter_fun` for the four alterations.

```
alter_fun = list(
  background = function(x, y, w, h)
    grid.rect(x, y, w*0.9, h*0.9, gp = gpar(fill = "#CCCCCC", col = NA)),
  # red rectangles
  snv = function(x, y, w, h)
    grid.rect(x, y, w*0.9, h*0.9, gp = gpar(fill = "red", col = NA)),
  # blue rectangles
  indel = function(x, y, w, h)
    grid.rect(x, y, w*0.9, h*0.9, gp = gpar(fill = "blue", col = NA)),
  # dots
  intronic = function(x, y, w, h)
    grid.points(x, y, pch = 16),
  # crossed lines
  exonic = function(x, y, w, h) {
    grid.segments(x - w*0.4, y - h*0.4, x + w*0.4, y + h*0.4, gp = gpar(lwd = 2))
    grid.segments(x + w*0.4, y - h*0.4, x - w*0.4, y + h*0.4, gp = gpar(lwd = 2))
  }
}
```

)

For the alteration types in the primary class (`snv` and `indel`), we use colored rectangles to represent them because the rectangles are visually obvious, while for the alteration types in the secondary class (`intronic` and `exonic`), we only use simple symbols (dots for `intronic` and crossed diagonal lines for `exonic`). Since there is no color corresponding to `intronic` and `exonic`, we don't need to define colors for these two types, and on the barplot annotation for genes and samples, only `snv` and `indel` are visualized (so the height for `snv` in the barplot corresponds to the number of intronic snv plus exonic snv).

```
# we only define color for snv and indel, so barplot annotations only show snv and indel
oncoPrint(m, alter_fun = alter_fun, col = c(snv = "red", indel = "blue"))
```

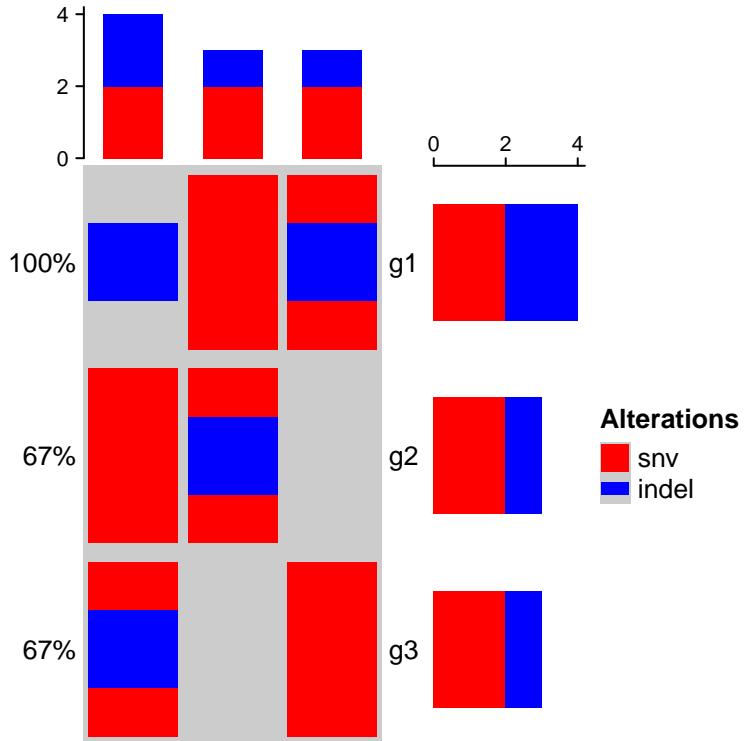


7.1.5 Simplify alter_fun

If the graphics are only simple graphics, e.g., rectangles, points, the graphic functions can be automatically generated by `alter_graphic()` function. One of previous example can be simplified as:

```
oncoPrint(mat,
         alter_fun = list(
           snv = alter_graphic("rect", width = 0.9, height = 0.9, fill = col["snv"]),
           indel = alter_graphic("rect", width = 0.9, height = 0.4, fill = col["indel"]))
```

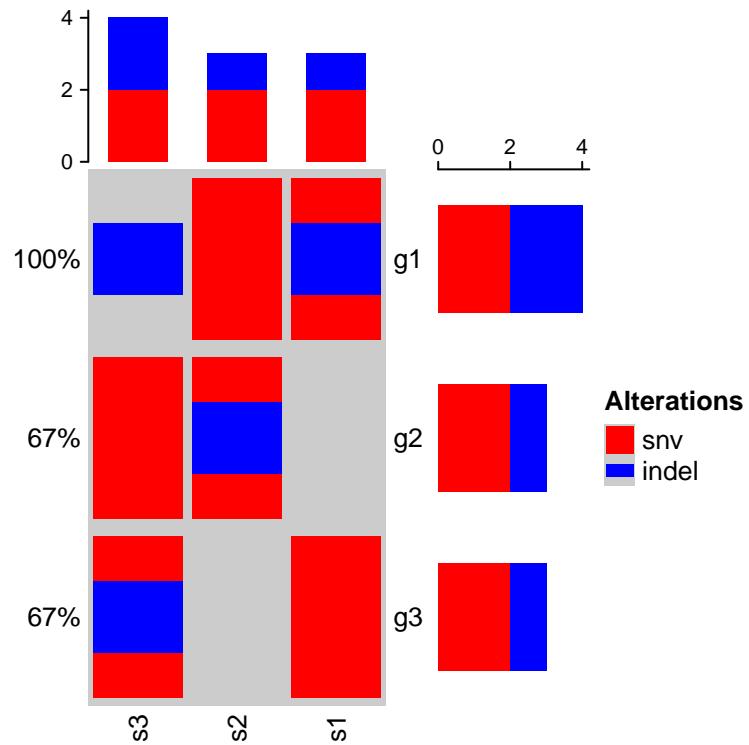
```
), col = col)
```



7.1.6 Other heatmap-related settings

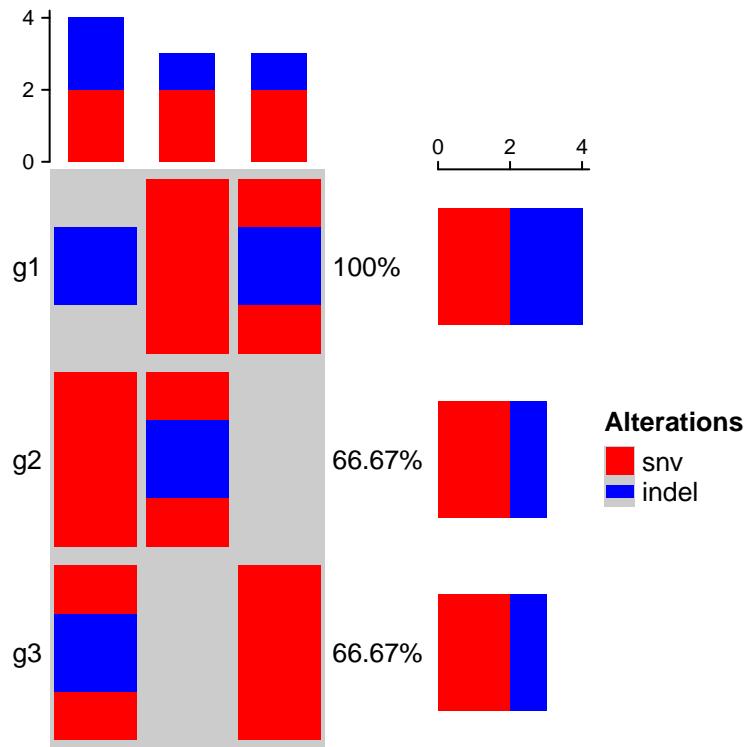
Column names are by default not drawn in the plot. It is can be turned on by setting `show_column_names = TRUE`.

```
alter_fun = list(
  snv = function(x, y, w, h) grid.rect(x, y, w*0.9, h*0.9,
    gp = gpar(fill = col["snv"], col = NA)),
  indel = function(x, y, w, h) grid.rect(x, y, w*0.9, h*0.4,
    gp = gpar(fill = col["indel"], col = NA))
)
oncoPrint(mat, alter_fun = alter_fun, col = col, show_column_names = TRUE)
```



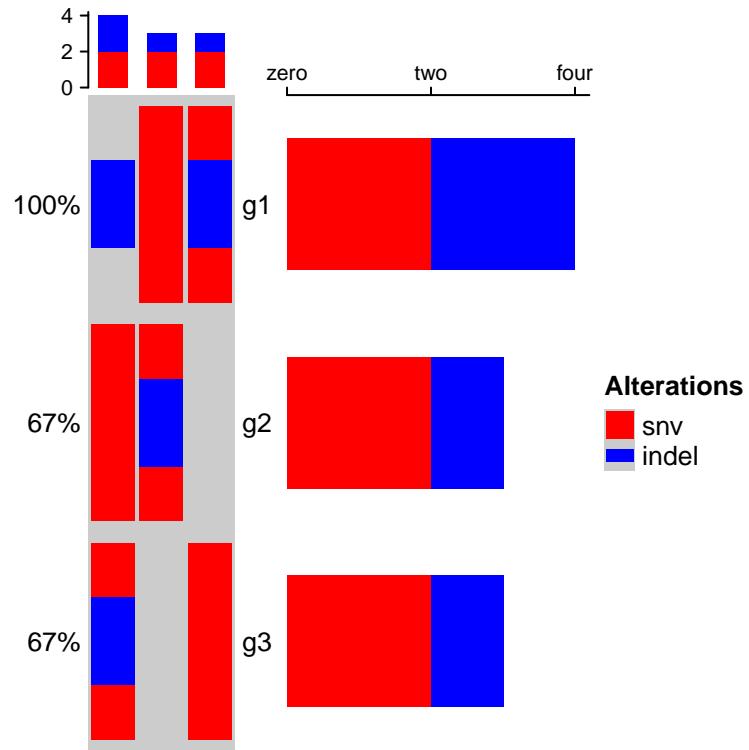
Row names and percent texts can be turned on/off by setting `show_pct` and `show_row_names`. The side of both according to the oncoPrint is controlled by `pct_side` and `row_names_side`. Digits of the percent values are controlled by `pct_digits`.

```
oncoPrint(mat, alter_fun = alter_fun, col = col,
          row_names_side = "left", pct_side = "right", pct_digits = 2)
```



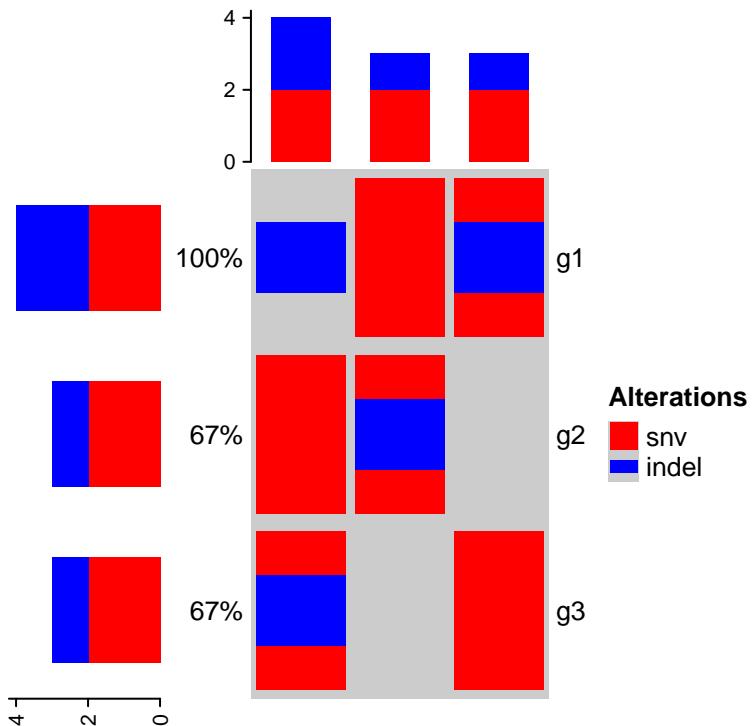
The barplot annotations on the both side are controlled by `anno_oncoprint_barplot()` annotation function. Customization such as the size and the axes can be set directly in `anno_oncoprint_barplot()`. More examples of setting `anno_oncoprint_barplot()` can be found in Section 7.2.3.

```
oncoPrint(mat, alter_fun = alter_fun, col = col,
          top_annotation = HeatmapAnnotation(
            cbar = anno_oncoprint_barplot(height = unit(1, "cm"))),
          right_annotation = rowAnnotation(
            rbar = anno_oncoprint_barplot(
              width = unit(4, "cm"),
              axis_param = list(at = c(0, 2, 4),
                labels = c("zero", "two", "four")),
              side = "top",
              labels_rot = 0))),
        )
```



Some people might want to move the right barplots to the left of the oncoPrint:

```
oncoPrint(mat, alter_fun = alter_fun, col = col,
          left_annotation = rowAnnotation(
            rbar = anno_oncoprint_barplot(
              axis_param = list(direction = "reverse"))
          ),
          right_annotation = NULL)
```



OncoPrints essentially are heatmaps, thus, there are many arguments set in `Heatmap()` can also be set in `oncoPrint()`. In following section, we use a real-world dataset to demonstrate more use of `oncoPrint()` function.

7.2 Apply to cBioPortal dataset

We use a real-world dataset to demonstrate advanced usage of `oncoPrint()`. The data is retrieved from cBioPortal. Steps for getting the data are as follows:

1. go to <http://www.cbioportal.org>,
2. search Cancer Study: “*Lung Adenocarcinoma Carcinoma*” and select: “*Lung Adenocarcinoma Carcinoma (TCGA, Provisional)*,”
3. in “Enter Gene Set” field, select: “*General: Ras-Raf-MEK-Erk/JNK signaling (26 genes)*,”
4. submit the form.

In the result page,

5. go to “Download” tab, download text in “*Type of Genetic alterations across all cases*.”

The order of samples can also be downloaded from the results page,

6. go to “*OncoPrint*” tab, move the mouse above the plot, click “*download*”

icon and click “*Sample order*.”

The data is already in **ComplexHeatmap** package. First we read the data and make some pre-processing.

```
mat = read.table(system.file("extdata", package = "ComplexHeatmap",
  "tcga_lung_adenocarcinoma_provisional_ras Raf_mek_jnk_signalling.txt"),
  header = TRUE, stringsAsFactors = FALSE, sep = "\t")
mat[is.na(mat)] = ""
rownames(mat) = mat[, 1]
mat = mat[, -1]
mat = mat[, -ncol(mat)]
mat = t(as.matrix(mat))
mat[1:3, 1:3]

##      TCGA-05-4384-01 TCGA-05-4390-01 TCGA-05-4425-01
## KRAS " "           "MUT;"           " "
## HRAS " "           " "               " "
## BRAF " "           " "               " "
```

There are three different alterations in `mat`: HOMDEL, AMP and MUT. We first define how to add graphics for different alterations.

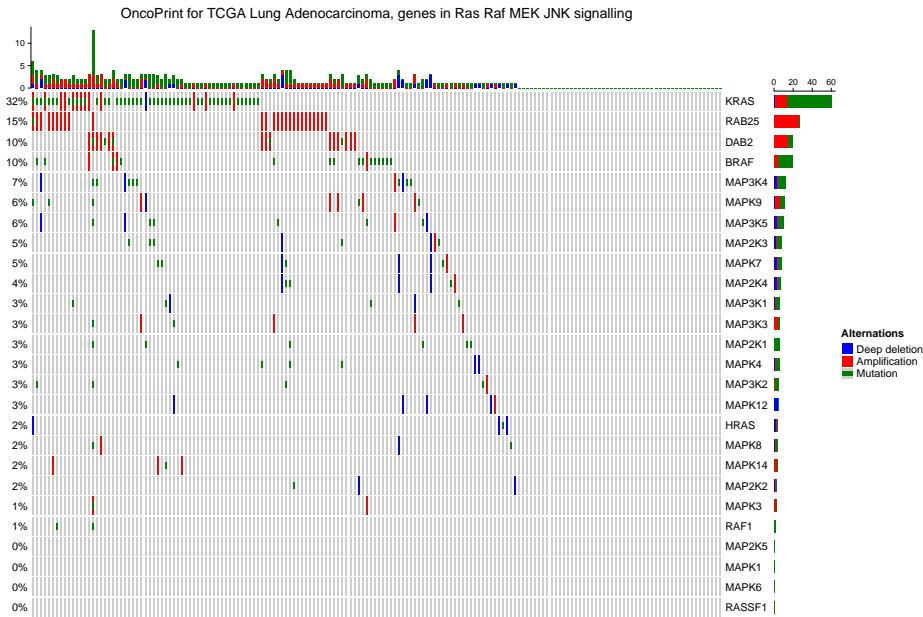
```
col = c("HOMDEL" = "blue", "AMP" = "red", "MUT" = "#008000")
alter_fun = list(
  background = function(x, y, w, h) {
    grid.rect(x, y, w-unit(2, "pt"), h-unit(2, "pt"),
    gp = gpar(fill = "#CCCCCC", col = NA))
  },
  # big blue
  HOMDEL = function(x, y, w, h) {
    grid.rect(x, y, w-unit(2, "pt"), h-unit(2, "pt"),
    gp = gpar(fill = col["HOMDEL"], col = NA))
  },
  # big red
  AMP = function(x, y, w, h) {
    grid.rect(x, y, w-unit(2, "pt"), h-unit(2, "pt"),
    gp = gpar(fill = col["AMP"], col = NA))
  },
  # small green
  MUT = function(x, y, w, h) {
    grid.rect(x, y, w-unit(2, "pt"), h*0.33,
    gp = gpar(fill = col["MUT"], col = NA))
  }
)
```

Just a note, since the graphics are all rectangles, they can be simplified by generating by `alter_graphic()`:

```
# just for demonstration
alter_fun = list(
  background = alter_graphic("rect", fill = "#CCCCCC"),
  HOMDEL = alter_graphic("rect", fill = col["HOMDEL"]),
  AMP = alter_graphic("rect", fill = col["AMP"]),
  MUT = alter_graphic("rect", height = 0.33, fill = col["MUT"])
)
```

Now we make the oncoPrint. We save `column_title` and `heatmap_legend_param` as variables because they are used multiple times in following code chunks.

```
column_title = "OncoPrint for TCGA Lung Adenocarcinoma, genes in Ras Raf MEK JNK signalling"
heatmap_legend_param = list(title = "Alternations", at = c("HOMDEL", "AMP", "MUT"),
                             labels = c("Deep deletion", "Amplification", "Mutation"))
oncoPrint(mat,
          alter_fun = alter_fun, col = col,
          column_title = column_title, heatmap_legend_param = heatmap_legend_param)
```

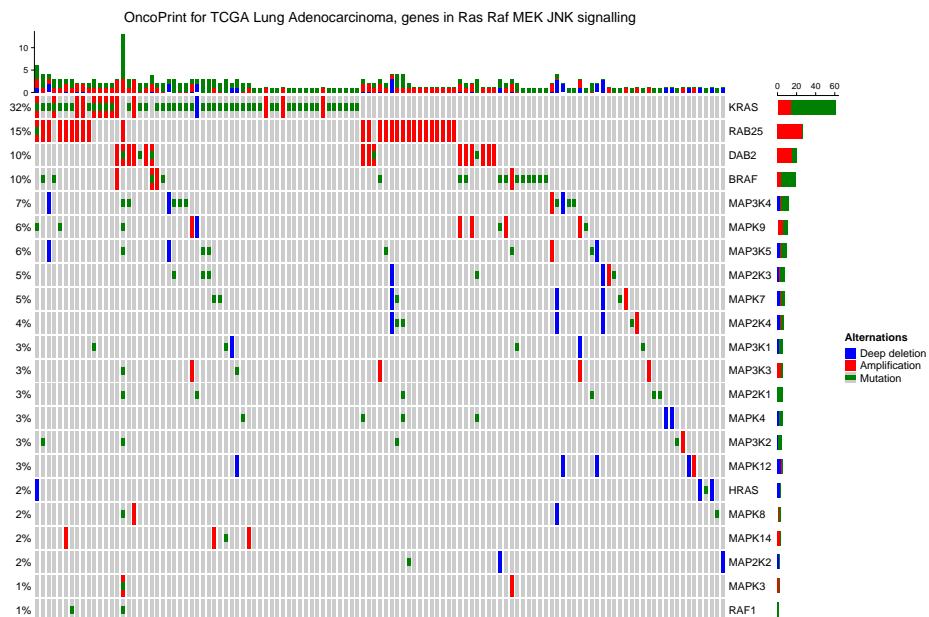


As you see, the genes and samples are reordered automatically. Rows are sorted based on the frequency of the alterations in all samples and columns are reordered to visualize the mutual exclusivity between samples. The column reordering is based on the “memo sort” method, adapted from <https://gist.github.com/armish/564a65ab874a770e2c26>.

7.2.1 Remove empty rows and columns

By default, if samples or genes have no alterations, they will still remain in the heatmap, but you can set `remove_empty_columns` and `remove_empty_rows` to `TRUE` to remove them:

```
oncoPrint(mat,
          alter_fun = alter_fun, col = col,
          remove_empty_columns = TRUE, remove_empty_rows = TRUE,
          column_title = column_title, heatmap_legend_param = heatmap_legend_param)
```

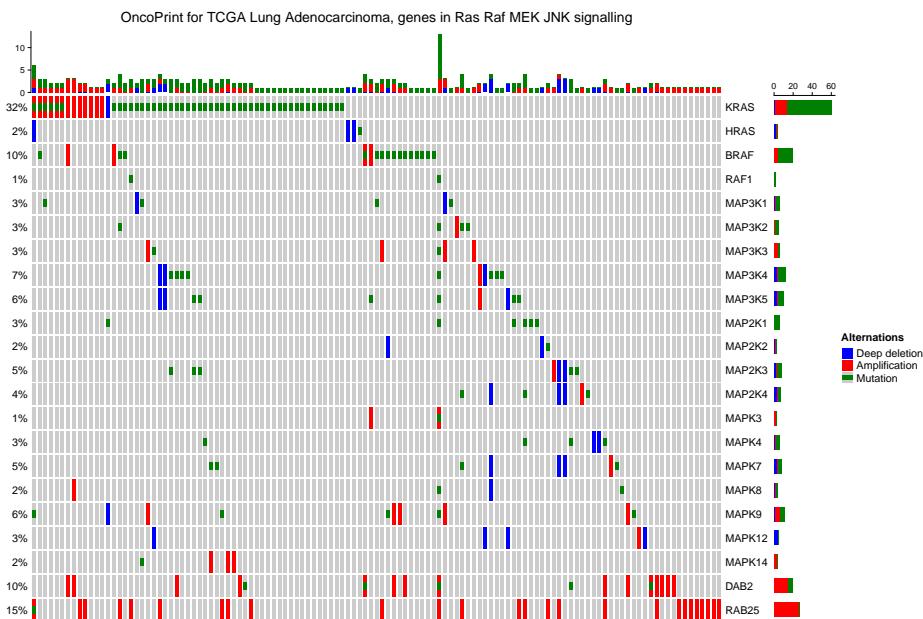


The number of rows and columns may be reduced after empty rows and columns are removed. All the components of the oncoPrint are adjusted accordingly. When the oncoPrint is concatenated with other heatmaps and annotations, this may cause a problem that the number of rows or columns are not all identical in the heatmap list. So, if you put oncoPrint into a heatmap list and you don't want to see empty rows or columns, you need to remove them manually before sending to `oncoPrint()` function (this preprocess should be very easy for you!).

7.2.2 Reorder the oncoPrint

As the normal `Heatmap()` function, `row_order` or `column_order` can be assigned with a vector of orders (either numeric or character). In following example, the order of samples are gathered from cBio as well. You can see the difference for the sample order between ‘memo sort’ and the method used by cBio.

```
sample_order = scan(paste0(system.file("extdata", package = "ComplexHeatmap"),
  "/sample_order.txt"), what = "character")
oncoPrint(mat,
  alter_fun = alter_fun, col = col,
  row_order = 1:nrow(mat), column_order = sample_order,
  remove_empty_columns = TRUE, remove_empty_rows = TRUE,
  column_title = column_title, heatmap_legend_param = heatmap_legend_param)
```



Again, `row_order` and `column_order` are automatically adjusted if `remove_empty_rows` and `remove_empty_columns` are set to TRUE.

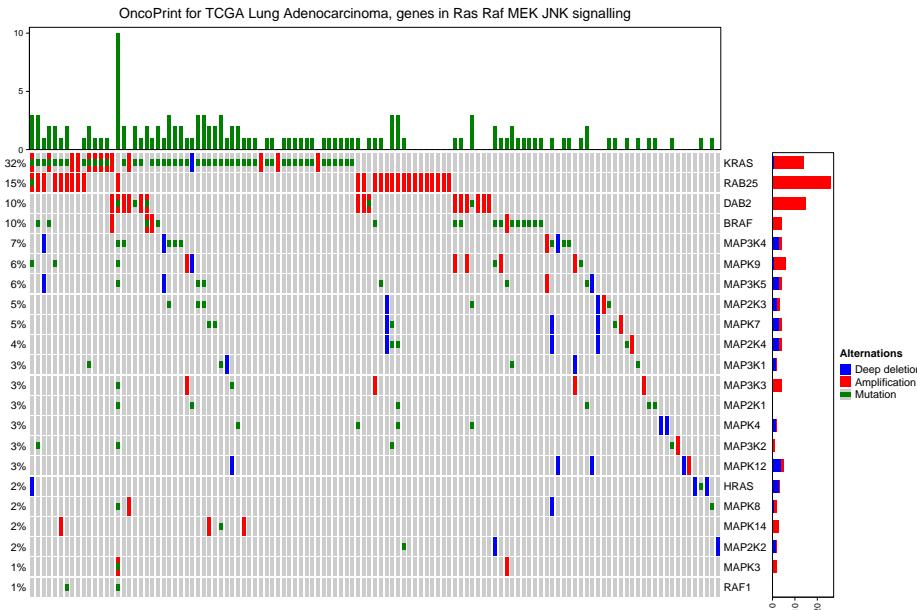
7.2.3 OncoPrint annotations

The oncoPrint has several pre-defined annotations.

On top and right of the oncoPrint, there are barplots showing the number of different alterations for each gene or for each sample, and on the left of the oncoPrint is a text annotation showing the percent of samples that have alterations for every gene.

The barplot annotation is implemented by `anno_oncoprint_barplot()` where you can set the the annotation there. Barplots by default draw for all alteration types, but you can also select subset of alterations to put on barplots by setting in `anno_oncoprint_barplot()`. `anno_oncoprint_barplot()` is a simple wrapper around `anno_barplot()` where the frequency matrix is just interanally calculated. See following example:

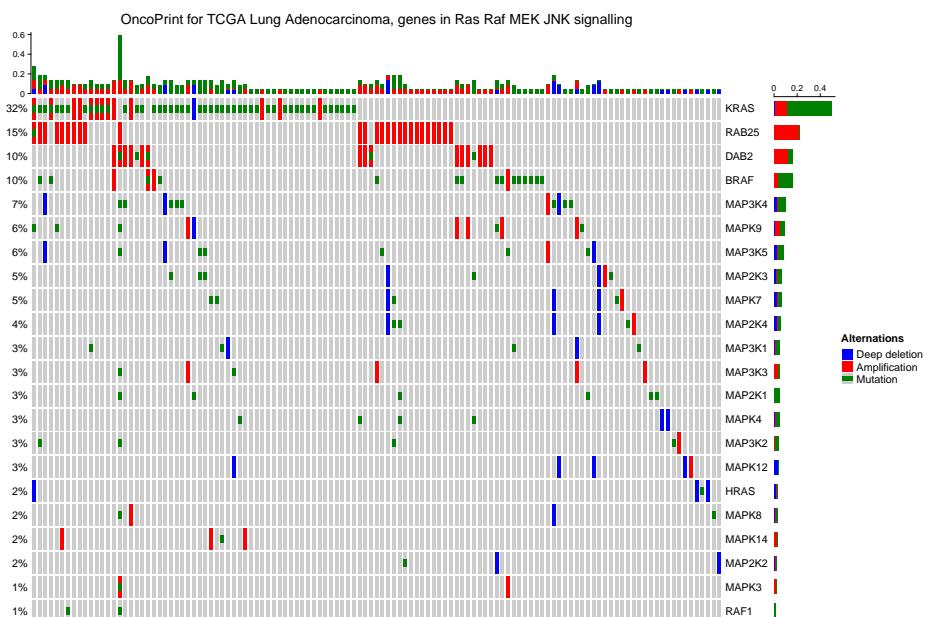
```
oncoPrint(mat,
    alter_fun = alter_fun, col = col,
    top_annotation = HeatmapAnnotation(
        column_barplot = anno_oncoprint_barplot("MUT", border = TRUE, # only MUT
                                                height = unit(4, "cm"))
    ),
    right_annotation = rowAnnotation(
        row_barplot = anno_oncoprint_barplot(c("AMP", "HOMDEL"), # only AMP and HOMDEL
                                              border = TRUE, height = unit(4, "cm"),
                                              axis_param = list(side = "bottom", labels_rot = 90))
    ),
    remove_empty_columns = TRUE, remove_empty_rows = TRUE,
    column_title = column_title, heatmap_legend_param = heatmap_legend_param)
```



By default the barplot annotation shows the frequencies. The values can be changed to fraction by setting `show_fraction = TRUE` in `anno_oncoprint_barplot()`:

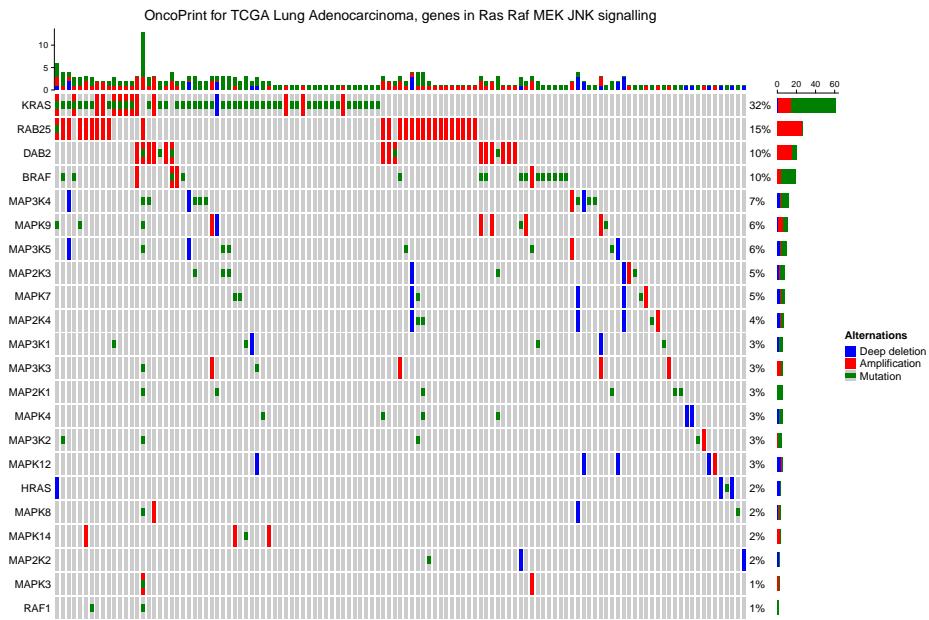
```
oncoPrint(mat,
    alter_fun = alter_fun, col = col,
    top_annotation = HeatmapAnnotation(
        column_barplot = anno_oncoprint_barplot(show_fraction = TRUE)
    ),
    right_annotation = rowAnnotation(
        row_barplot = anno_oncoprint_barplot(show_fraction = TRUE)
```

```
),
remove_empty_columns = TRUE, remove_empty_rows = TRUE,
column_title = column_title, heatmap_legend_param = heatmap_legend_param)
```



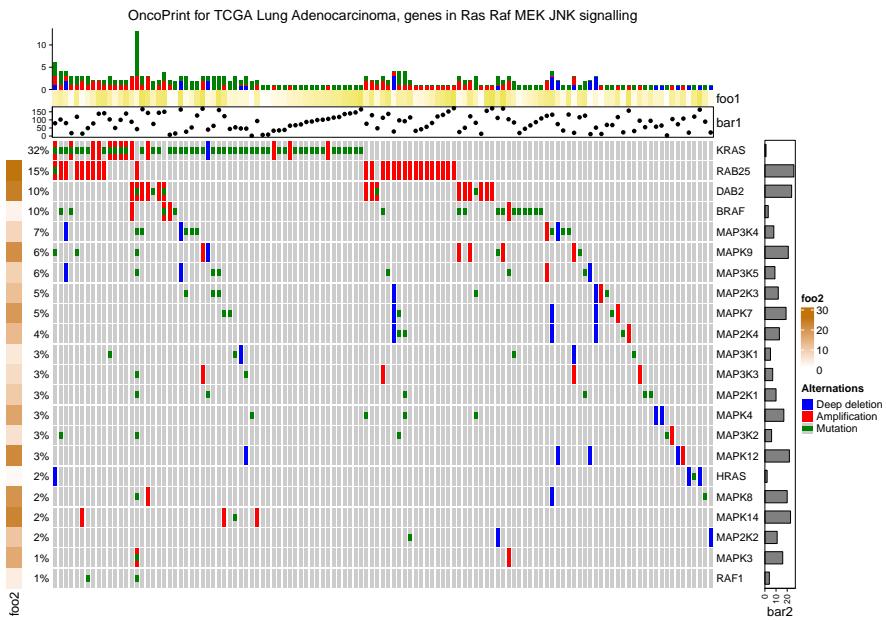
The percent values and row names are internally constructed as text annotations. You can set `show_pct` and `show_row_names` to turn them on or off. `pct_side` and `row_names_side` controls the sides where they are put.

```
oncoPrint(mat,
alter_fun = alter_fun, col = col,
remove_empty_columns = TRUE, remove_empty_rows = TRUE,
pct_side = "right", row_names_side = "left",
column_title = column_title, heatmap_legend_param = heatmap_legend_param)
```



The barplot annotation for oncoPrint are essentially normal annotations, you can add more annotations in `HeatmapAnnotation()` or `rowAnnotation()` in the normal way:

```
oncoPrint(mat,
  alter_fun = alter_fun, col = col,
  remove_empty_columns = TRUE, remove_empty_rows = TRUE,
  top_annotation = HeatmapAnnotation(cbar = anno_oncoprint_barplot(),
    foo1 = 1:172,
    bar1 = anno_points(1:172)
  ),
  left_annotation = rowAnnotation(foo2 = 1:26),
  right_annotation = rowAnnotation(bar2 = anno_barplot(1:26)),
  column_title = column_title, heatmap_legend_param = heatmap_legend_param)
```



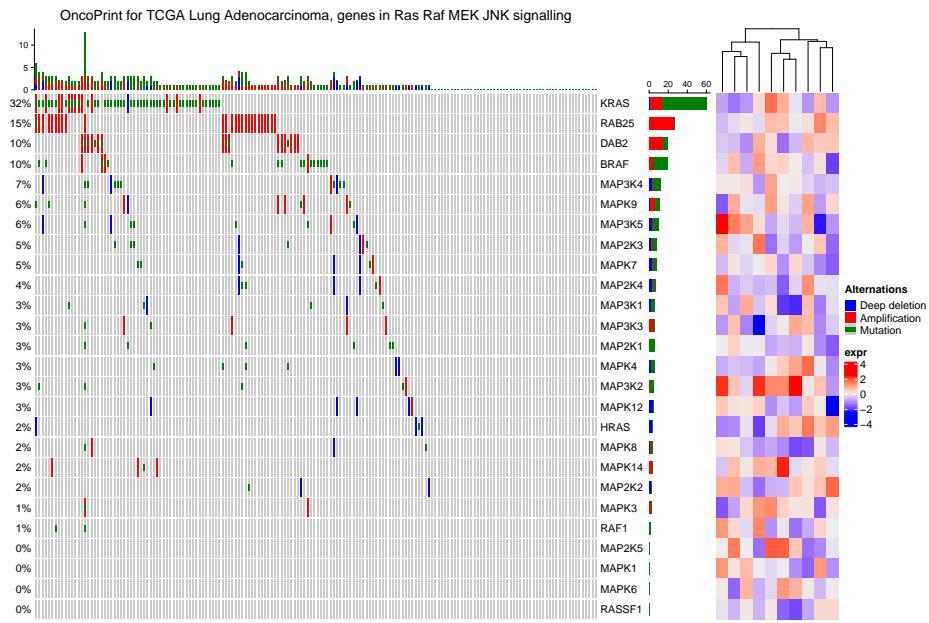
As you see, the percent annotation, the row name annotation and the oncoPrint annotation are appended to the user-specified annotation by default. Also annotations are automatically adjusted if `remove_empty_columns` and `remove_empty_rows` are set to TRUE.

7.2.4 oncoPrint as a Heatmap

`oncoPrint()` actually returns a `Heatmap` object, so you can add more heatmaps and annotations horizontally or vertically to visualize more complicated associations.

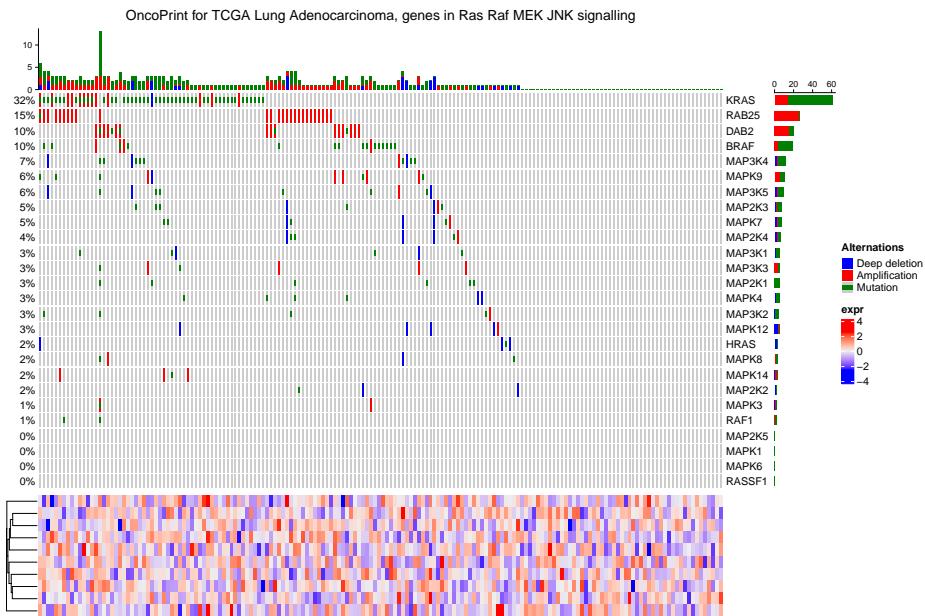
Following example adds a heatmap horizontally. Remember you can always add row annotations to the heatmap list.

```
ht_list = oncoPrint(mat,
    alter_fun = alter_fun, col = col,
    column_title = column_title, heatmap_legend_param = heatmap_legend_param) +
Heatmap(matrix(rnorm(nrow(mat)*10), ncol = 10), name = "expr", width = unit(4, "cm"))
draw(ht_list)
```



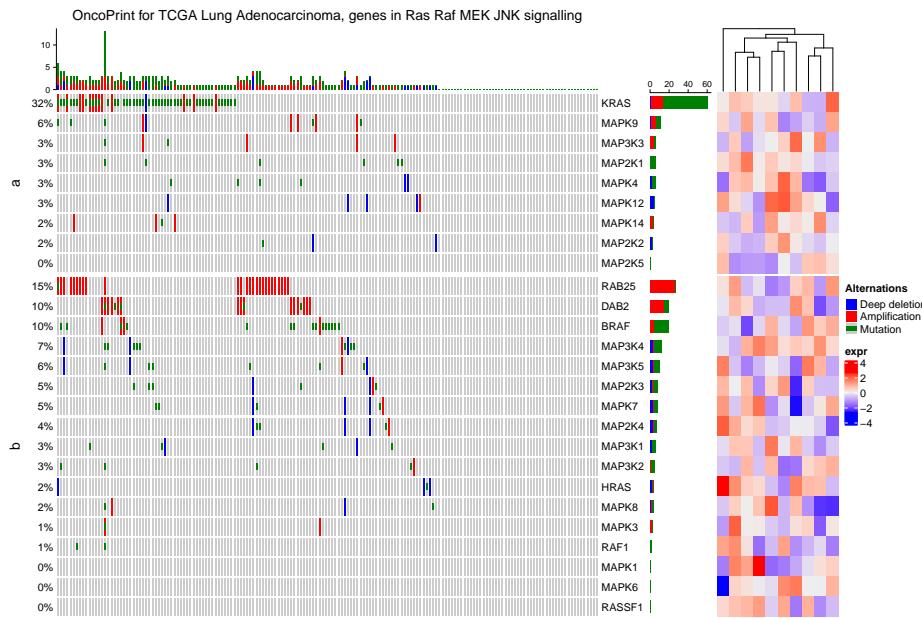
or add it vertically:

```
ht_list = oncoPrint(mat,
  alter_fun = alter_fun, col = col,
  column_title = column_title, heatmap_legend_param = heatmap_legend_param) %v%
Heatmap(matrix(rnorm(ncol(mat)*10), nrow = 10), name = "expr", height = unit(4, "cm"))
draw(ht_list)
```



Similar as normal heatmap list, you can split the heatmap list:

```
ht_list = oncoPrint(mat,
  alter_fun = alter_fun, col = col,
  column_title = column_title, heatmap_legend_param = heatmap_legend_param) +
Heatmap(matrix(rnorm(nrow(mat)*10), ncol = 10), name = "expr", width = unit(4, "cm"))
draw(ht_list, row_split = sample(c("a", "b"), nrow(mat), replace = TRUE))
```



When `remove_empty_columns` or `remove_empty_rows` is set to TRUE, the number of genes or the samples may not be the original number. If the original matrix has row names and column names. The subset of rows and columns can be get as follows:

```
ht = oncoPrint(mat,
  alter_fun = alter_fun, col = col,
  remove_empty_columns = TRUE, remove_empty_rows = TRUE,
  column_title = column_title, heatmap_legend_param = heatmap_legend_param)
rownames(ht@matrix)

## [1] "KRAS"    "HRAS"    "BRAF"    "RAF1"    "MAP3K1"   "MAP3K2"   "MAP3K3"   "MAP3K4"
## [9] "MAP3K5"   "MAP2K1"   "MAP2K2"   "MAP2K3"   "MAP2K4"    "MAPK3"    "MAPK4"    "MAPK7"
## [17] "MAPK8"    "MAPK9"    "MAPK12"   "MAPK14"   "DAB2"     "RAB25"

colnames(ht@matrix)

## [1] "TCGA-05-4390-01" "TCGA-38-4631-01" "TCGA-44-6144-01" "TCGA-44-6145-01"
## [5] "TCGA-44-6146-01" "TCGA-49-4488-01" "TCGA-50-5930-01" "TCGA-50-5931-01"
## [9] "TCGA-50-5932-01" "TCGA-50-5933-01" "TCGA-50-5941-01" "TCGA-50-5942-01"
## [13] "TCGA-50-5944-01" "TCGA-50-5946-01" "TCGA-50-6591-01" "TCGA-50-6592-01"
## [17] "TCGA-50-6594-01" "TCGA-67-4679-01" "TCGA-67-6215-01" "TCGA-73-4658-01"
## [21] "TCGA-73-4676-01" "TCGA-75-5122-01" "TCGA-75-5125-01" "TCGA-75-5126-01"
## [25] "TCGA-75-6206-01" "TCGA-75-6211-01" "TCGA-86-6562-01" "TCGA-05-4396-01"
## [29] "TCGA-05-4405-01" "TCGA-05-4410-01" "TCGA-05-4415-01" "TCGA-05-4417-01"
## [33] "TCGA-05-4424-01" "TCGA-05-4427-01" "TCGA-05-4433-01" "TCGA-44-6774-01"
## [37] "TCGA-44-6775-01" "TCGA-44-6776-01" "TCGA-44-6777-01" "TCGA-44-6778-01"
```

```
## [41] "TCGA-49-4487-01" "TCGA-49-4490-01" "TCGA-49-6744-01" "TCGA-49-6745-01"
## [45] "TCGA-49-6767-01" "TCGA-50-5044-01" "TCGA-50-5051-01" "TCGA-50-5072-01"
## [49] "TCGA-50-6590-01" "TCGA-55-6642-01" "TCGA-55-6712-01" "TCGA-71-6725-01"
## [53] "TCGA-91-6828-01" "TCGA-91-6829-01" "TCGA-91-6835-01" "TCGA-91-6836-01"
## [57] "TCGA-35-3615-01" "TCGA-44-2655-01" "TCGA-44-2656-01" "TCGA-44-2662-01"
## [61] "TCGA-44-2666-01" "TCGA-44-2668-01" "TCGA-55-1592-01" "TCGA-55-1594-01"
## [65] "TCGA-55-1595-01" "TCGA-64-1676-01" "TCGA-64-1677-01" "TCGA-64-1678-01"
## [69] "TCGA-64-1680-01" "TCGA-67-3771-01" "TCGA-67-3773-01" "TCGA-67-3774-01"
## [73] "TCGA-05-4244-01" "TCGA-05-4249-01" "TCGA-05-4250-01" "TCGA-35-4122-01"
## [77] "TCGA-35-4123-01" "TCGA-44-2657-01" "TCGA-44-3398-01" "TCGA-44-3918-01"
## [81] "TCGA-05-4382-01" "TCGA-05-4389-01" "TCGA-05-4395-01" "TCGA-05-4397-01"
## [85] "TCGA-05-4398-01" "TCGA-05-4402-01" "TCGA-05-4403-01" "TCGA-05-4418-01"
## [89] "TCGA-05-4420-01" "TCGA-05-4422-01" "TCGA-05-4426-01" "TCGA-05-4430-01"
## [93] "TCGA-05-4434-01" "TCGA-38-4625-01" "TCGA-38-4626-01" "TCGA-38-4628-01"
## [97] "TCGA-38-4630-01" "TCGA-44-3396-01" "TCGA-49-4486-01" "TCGA-49-4505-01"
## [101] "TCGA-49-4506-01" "TCGA-49-4507-01" "TCGA-49-4510-01" "TCGA-73-4659-01"
## [105] "TCGA-73-4662-01" "TCGA-73-4668-01" "TCGA-73-4670-01" "TCGA-73-4677-01"
## [109] "TCGA-05-5428-01" "TCGA-05-5715-01" "TCGA-50-5045-01" "TCGA-50-5049-01"
## [113] "TCGA-50-5936-01" "TCGA-55-5899-01" "TCGA-64-5774-01" "TCGA-64-5775-01"
## [117] "TCGA-64-5778-01" "TCGA-64-5815-01" "TCGA-75-5146-01" "TCGA-75-5147-01"
## [121] "TCGA-80-5611-01"
```


Chapter 8

UpSet plot

UpSet plot provides an efficient way to visualize intersections of multiple sets compared to the traditional approaches, i.e. the Venn Diagram. It is implemented in the UpSetR package in R. Here we re-implemented UpSet plots with the **ComplexHeatmap** package with some improvements.

8.1 Input data

To represent multiple sets, the variable can be represented as:

1. A list of sets where each set is a vector, e.g.:

```
list(set1 = c("a", "b", "c"),
     set2 = c("b", "c", "d", "e"),
     ...)
```

2. A binary matrix/data frame where rows are elements and columns are sets, e.g.:

```
set1 set2 set3
h    1    1    1
t    1    0    1
j    1    0    0
u    1    0    1
w    1    0    0
...
.
```

In the matrix, e.g., for row **t**, it means, **t** is in set **set1**, not in set **set2**, and in set **set3**. Note the matrix is also valid if it is a logical matrix.

If the variable is a data frame, the binary columns (only contain 0 and 1) and the logical columns are only used.

Both formats can be used for making UpSet plots, users can still use `list_to_matrix()` to convert from list to the binary matrix.

```
lt = list(set1 = c("a", "b", "c"),
          set2 = c("b", "c", "d", "e"))
list_to_matrix(lt)
```

```
##   set1 set2
## a    1    0
## b    1    1
## c    1    1
## d    0    1
## e    0    1
```

You can also set the universal set in `list_to_matrix()`:

```
list_to_matrix(lt, universal = letters[1:10])
```

```
##   set1 set2
## a    1    0
## b    1    1
## c    1    1
## d    0    1
## e    0    1
## f    0    0
## g    0    0
## h    0    0
## i    0    0
## j    0    0
```

If the universal set does not completely cover the input sets, those elements are not in the universal set are removed:

```
list_to_matrix(lt, universal = letters[1:4])
```

```
##   set1 set2
## a    1    0
## b    1    1
## c    1    1
## d    0    1
```

3. The set can be genomic intervals, then it can only be represented as a list of `GRanges/IRanges` objects.

```
list(set1 = GRanges(...),
     set2 = GRanges(...),
     ...)
```

8.2 Mode

E.g. for three sets (**A**, **B**, **C**), all combinations of selecting elements in the set or not in the set are encoded as following:

```

A B C
1 1 1
1 1 0
1 0 1
0 1 1
1 0 0
0 1 0
0 0 1

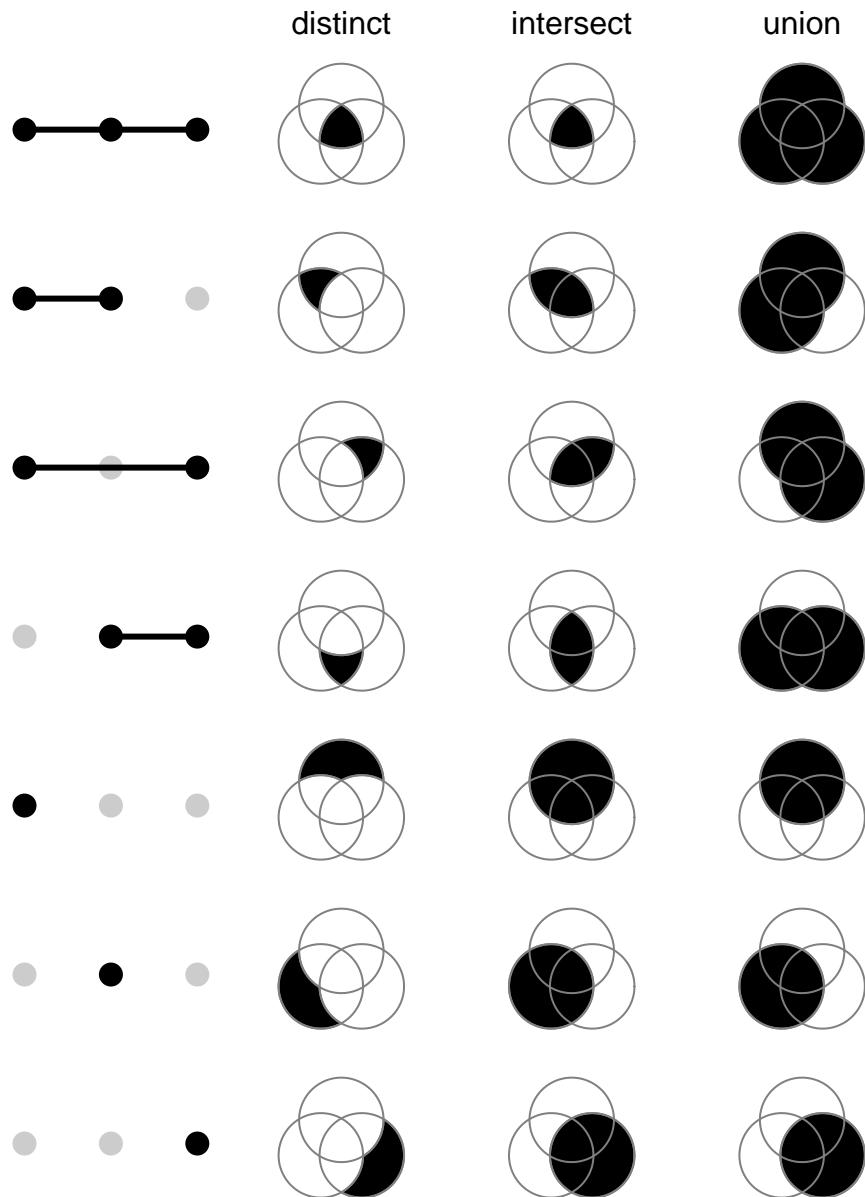
```

A value of 1 means to select that set and 0 means not to select that set. E.g., 1 1 0 means to select set A, B while not set C. Note there is no 0 0 0, because the background set is not of interest here. In following part of this section, we refer **A**, **B** and **C** as **sets** and each combination as **combination set**. The whole binary matrix is called **combination matrix**.

The UpSet plot visualizes the **size** of each combination set. With the binary code of each combination set, next we need to define how to calculate the size of that combination set. There are three modes:

1. **distinct** mode: 1 means in that set and 0 means not in that set, then 1 1 0 means a set of elements both in set **A** and **B**, while not in **C** (`setdiff(intersect(A, B), C)`). Under this mode, the seven combination sets are the seven partitions in the Venn diagram and they are mutually exclusive.
2. **intersect** mode: 1 means in that set and 0 is not taken into account, then, 1 1 0 means a set of elements in set **A** and **B**, and they can also in **C** or not in **C** (`intersect(A, B)`). Under this mode, the seven combination sets can overlap.
3. **union** mode: 1 means in that set and 0 is not taken into account. When there are multiple 1, the relationship is **OR**. Then, 1 1 0 means a set of elements in set **A** or **B**, and they can also in **C** or not in **C** (`union(A, B)`). Under this mode, the seven combination sets can overlap.

The three modes are illustrated in following figure:



8.3 Make the combination matrix

The function `make_comb_mat()` generates the combination matrix as well as calculates the size of the sets and the combination sets. The input can be one single variable or name-value pairs:

```

set.seed(123)
lt = list(a = sample(letters, 5),
          b = sample(letters, 10),
          c = sample(letters, 15))
m1 = make_comb_mat(lt)
m1

## A combination matrix with 3 sets and 7 combinations.
## ranges of combination set size: c(1, 8).
## mode for the combination size: distinct.
## sets are on rows.
##
## Combination sets are:
##   a b c code size
##   x x x  111    2
##   x x     110    1
##   x   x  101    1
##       x x  011    4
##       x   100    1
##           x  010    3
##               x  001    8
##
## Sets are:
##   set size
##   a      5
##   b     10
##   c     15
m2 = make_comb_mat(a = lt$a, b = lt$b, c = lt$c)
m3 = make_comb_mat(list_to_matrix(lt))

```

`m1`, `m2` and `m3` are identical.

The mode is controlled by the `mode` argument:

```

m1 = make_comb_mat(lt) # the default mode is `distinct`
m2 = make_comb_mat(lt, mode = "intersect")
m3 = make_comb_mat(lt, mode = "union")

```

The UpSet plots under different modes will be demonstrated in later sections.

When there are too many sets, the sets can be pre-filtered by the set sizes. The `min_set_size` and `top_n_sets` are for this purpose. `min_set_size` controls the minimal size for the sets and `top_n_sets` controls the number of top sets with the largest sizes.

```

m1 = make_comb_mat(lt, min_set_size = 6)
m2 = make_comb_mat(lt, top_n_sets = 2)

```

The subsetting of the sets affects the calculation of the sizes of the combination sets, that is why it needs to be controlled at the combination matrix generation step. The subsetting of combination sets can be directly performed by subsetting the matrix:

```
m = make_comb_mat(lt)
m[1:4]

## A combination matrix with 3 sets and 4 combinations.
## ranges of combination set size: c(1, 4).
## mode for the combination size: distinct.
## sets are on rows.
##
## Combination sets are:
##   a b c code size
##   x x x  111    2
##   x x     110    1
##   x   x  101    1
##       x x  011    4
##
## Sets are:
##   set size
##   a      5
##   b      10
##   c     15
```

`make_comb_mat()` also allows to specify the universal set so that the complement set which contains elements not belonging to any set is also considered.

```
m = make_comb_mat(lt, universal_set = letters)
m

## A combination matrix with 3 sets and 8 combinations.
## ranges of combination set size: c(1, 8).
## mode for the combination size: distinct.
## sets are on rows.
##
## Combination sets are:
##   a b c code size
##   x x x  111    2
##   x x     110    1
##   x   x  101    1
##       x x  011    4
##   x     100    1
##       x  010    3
##           x  001    8
##               000    6
##
```

```

## Sets are:
##      set size
##      a    5
##      b   10
##      c   15
## complement   6

## A combination matrix with 3 sets and 5 combinations.
## ranges of combination set size: c(1, 3).
## mode for the combination size: distinct.
## sets are on rows.
##
## Combination sets are:
##   a b c code size
##   x x    110    1
##   x   x   101    1
##   x   x   011    2
##       x   001    3
##           000    3
##
## Sets are:
##      set size
##      a    2
##      b    3
##      c    6
## complement   3

```

If you already know the size of the complement size, you can directly set `complement_size` argument.

```

m = make_comb_mat(lt, complement_size = 5)
m

## A combination matrix with 3 sets and 8 combinations.
## ranges of combination set size: c(1, 8).
## mode for the combination size: distinct.
## sets are on rows.
##
## Combination sets are:
##   a b c code size
##   x x x   111    2
##   x   x   110    1

```

```

##   x   x  101    1
##   x x  011    4
##   x     100    1
##   x     010    3
##   x     001    8
##     000    5
##
## Sets are:
##           set size
##           a     5
##           b    10
##           c    15
## complement    5

```

When the input is matrix and it contains elements that do not belong to any of the set, these elements are treated as complement set.

```

x = list_to_matrix(lt, universal_set = letters)
m = make_comb_mat(x)
m

## A combination matrix with 3 sets and 8 combinations.
## ranges of combination set size: c(1, 8).
## mode for the combination size: distinct.
## sets are on rows.
##
## Combination sets are:
##   a b c code size
##   x x x  111    2
##   x x     110    1
##   x   x  101    1
##   x   x  011    4
##   x     100    1
##   x     010    3
##   x     001    8
##     000    6
##
## Sets are:
##           set size
##           a     5
##           b    10
##           c    15
## complement    6

```

Next we demonstrate a second example, where the sets are genomic regions. **When the sets are genomic regions, the size is calculated as the sum of the width of regions in each set (or in other words, the total number**

of base pairs).

```
library(circlize)
library(GenomicRanges)
lt2 = lapply(1:4, function(i) generateRandomBed())
lt2 = lapply(lt2, function(df) GRanges(seqnames = df[, 1],
  ranges = IRanges(df[, 2], df[, 3])))
names(lt2) = letters[1:4]
m2 = make_comb_mat(lt2)
m2

## A combination matrix with 4 sets and 15 combinations.
##   ranges of combination set size: c(184941701, 199900416).
##   mode for the combination size: distinct.
##   sets are on rows.
##
## Top 8 combination sets are:
##   a b c d code      size
##   x x x 0011 199900416
##   x       1000 199756519
##   x   x x 1011 198735008
##   x x x x 1111 197341532
##   x x x   1110 197137160
##   x x   x 1101 194569926
##   x       x 1001 194462988
##   x   x   1010 192670258
##
## Sets are:
##   set      size
##   a 1566783009
##   b 1535968265
##   c 1560549760
##   d 1552480645
```

We don't recommend to use **the number of regions** for the intersection of two sets of genomic regions. There are two reasons: 1. the value is not symmetric, i.e. the number of intersected regions measured in set1 is not always identical to the number of intersected regions measured in set2, thus, it is difficult to assign a value for the intersection between set1 and set2; 2. if one long region in set1 overlaps to another long region in set2, but only in a few base pairs, does it make sense to say these two regions are common in the two sets?

The universal set also works for sets as genomic regions.

8.4 Utility functions

`make_comb_mat()` returns a matrix, also in `comb_mat` class. There are some utility functions that can be applied to this `comb_mat` object:

- `set_name()`: The set names.
- `comb_name()`: The combination set names. The names of the combination sets are formatted as a string of binary bits. E.g. for three sets of **A**, **B**, **C**, the combination set with name “101” corresponds to selecting set **A**, not selecting set **B** and selecting set **C**.
- `set_size()`: The set sizes.
- `comb_size()`: The combination set sizes.
- `comb_degree()`: The degree for a combination set is the number of sets that are selected.
- `t()`: Transpose the combination matrix. By default `make_comb_mat()` generates a matrix where sets are on rows and combination sets are on columns, and so are they on the UpSet plots. By transposing the combination matrix, the position of sets and combination sets can be switched on the UpSet plot.
- `extract_comb()`: Extract the elements in a specified combination set. The usage will be explained later.
- Functions for subsetting the matrix.

Quick examples are:

```
m = make_comb_mat(lt)
set_name(m)

## [1] "a" "b" "c"
comb_name(m)

## [1] "111" "110" "101" "011" "100" "010" "001"
set_size(m)

##   a   b   c
##  5 10 15
comb_size(m)

## 111 110 101 011 100 010 001
##  2    1    1    4    1    3    8
comb_degree(m)

## 111 110 101 011 100 010 001
##  3    2    2    2    1    1    1
t(m)
```

```

## A combination matrix with 3 sets and 7 combinations.
## ranges of combination set size: c(1, 8).
## mode for the combination size: distinct.
## sets are on columns
##
## Combination sets are:
##   a b c code size
##   x x x  111    2
##   x x     110    1
##   x   x  101    1
##   x   x  011    4
##   x     100    1
##       x  010    3
##           x  001    8
##
## Sets are:
##   set size
##   a      5
##   b      10
##   c     15

```

For using `extract_comb()`, the valid combination set name should be from `comb_name()`. Note the elements in the combination sets depends on the “mode” set in `make_comb_mat()`.

```
extract_comb(m, "101")
```

```
## [1] "j"
```

And the example for sets that are the genomic regions:

```
# `lt2` was generated in the previous section
m2 = make_comb_mat(lt2)
set_size(m2)
```

```
##          a          b          c          d
## 1566783009 1535968265 1560549760 1552480645
```

```
comb_size(m2)
```

```
##      1111      1110      1101      1011      0111      1100      1010      1001
## 197341532 197137160 194569926 198735008 191312455 192109618 192670258 194462988
##      0110      0101      0011      1000      0100      0010      0001
## 191359036 184941701 199900416 199756519 187196837 192093895 191216619
```

And now `extract_comb()` returns genomic regions that are in the corresponding combination set.

```
extract_comb(m2, "1010")
```

```

## GRanges object with 5063 ranges and 0 metadata columns:
##           seqnames      ranges strand
##           <Rle>        <IRanges>  <Rle>
## [1]   chr1    255644-258083      *
## [2]   chr1    306114-308971      *
## [3]   chr1    1267493-1360170     *
## [4]   chr1    2661311-2665736     *
## [5]   chr1    3020553-3030645     *
## ...
## [5059] chrY  56286079-56286864     *
## [5060] chrY  57049541-57078332     *
## [5061] chrY  58691055-58699756     *
## [5062] chrY  58705675-58716954     *
## [5063] chrY  58765097-58776696     *
## -----
## seqinfo: 24 sequences from an unspecified genome; no seqlengths

```

With `comb_size()` and `comb_degree()`, we can filter the combination matrix as:

```

m = make_comb_mat(lt)
# combination set size >= 4
m[comb_size(m) >= 4]

## A combination matrix with 3 sets and 2 combinations.
## ranges of combination set size: c(4, 8).
## mode for the combination size: distinct.
## sets are on rows.
##
## Combination sets are:
##   a b c code size
##   x x  011    4
##   x   001    8
##
## Sets are:
##   set size
##   a     5
##   b    10
##   c    15
# combination set degree == 2
m[comb_degree(m) == 2]

## A combination matrix with 3 sets and 3 combinations.
## ranges of combination set size: c(1, 4).
## mode for the combination size: distinct.
## sets are on rows.
##
```

```
## Combination sets are:
##   a b c code size
##   x x    110    1
##   x   x   101    1
##   x   x   011    4
##
## Sets are:
##   set size
##   a     5
##   b     10
##   c    15
```

For the complement set, the name for this special combination set is only composed of zeros.

```
m2 = make_comb_mat(lt, universal_set = letters)
comb_name(m2) # see the first element

## [1] "111" "110" "101" "011" "100" "010" "001" "000"
comb_degree(m2)

## 111 110 101 011 100 010 001 000
## 3   2   2   2   1   1   1   0
```

If `universal_set` was set in `make_comb_mat()`, `extract_comb()` can be applied to the complement set.

```
m2 = make_comb_mat(lt, universal_set = letters)
extract_comb(m2, "000")

## [1] "a" "b" "f" "p" "u" "z"
m2 = make_comb_mat(lt, universal_set = letters[1:10])
extract_comb(m2, "000")

## [1] "a" "b" "f"
```

When `universal_set` was set, `extract_comb()` also works for genomic region sets.

In previous examples, we demonstrated using “one-dimensional index” such as:

```
m[comb_degree(m) == 2]
```

Since the combination matrix is naturally a matrix, the indices can also be applied to both dimensions. In the default settings, sets are on the rows and combination sets are on the columns, thus, indices on the first dimension of the matrix correspond to sets and indices on the second dimension correspond to combination sets:

```
# by set names
m[c("a", "b", "c"), ]
# by numeric indices
m[3:1, ]
```

New empty sets can be added to the combination matrix by:

```
# `d` is the new empty set
m[c("a", "b", "c", "d"), ]
```

Note when the indices specified do not cover all non-empty sets in the original combination matrix, the combination matrix will be re-calculated because it affects the values in the combination sets:

```
# if `c` is a non-empty set
m[c("a", "b"), ]
```

Similar for subsetting on the second dimension which correspond to the combination sets:

```
# reorder
m[, 5:1]
# take a subset
m[, 1:3]
# by character indices
m[, c("110", "101", "011")]
```

New empty combination sets can also be added by setting the character indices:

```
m[, c(comb_name(m), "100")]
```

Indices can be set on both dimension simultaneously only when the set indices covers all non-empty sets:

```
m[3:1, 5:1]
# this will throw an error because `c` is a non-empty set
m[c("a", "b"), 5:1]
```

If the combination matrix was transposed, the margin of the matrix where set indices and combination set indices should be switched.

```
tm = t(m)
tm[reverse(comb_name(tm)), reverse(set_name(tm))]
```

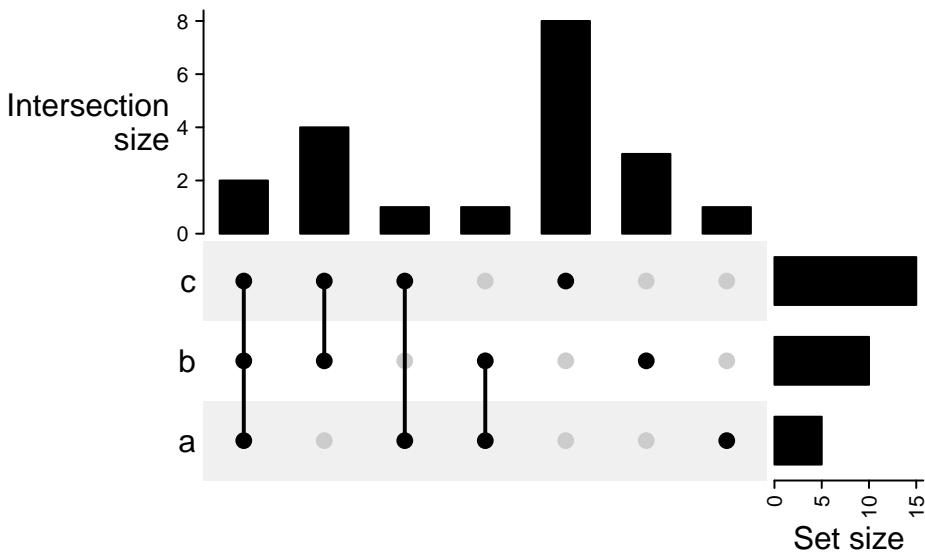
If only the indices for the combination sets are set as one-dimensional, it automatically works for both matrices that are transposed or not:

```
m[1:5]
tm[1:5]
```

8.5 Make the plot

Making the UpSet plot is very straightforward that users just send the combination matrix to `UpSet()` function:

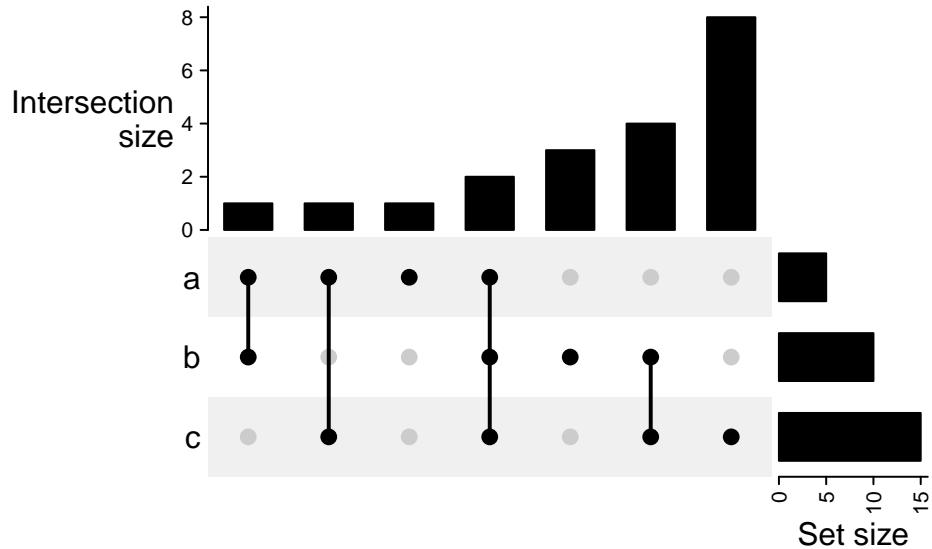
```
m = make_comb_mat(lt)
UpSet(m)
```



By default the sets are ordered by the size and the combination sets are ordered by the degree (number of sets that are selected).

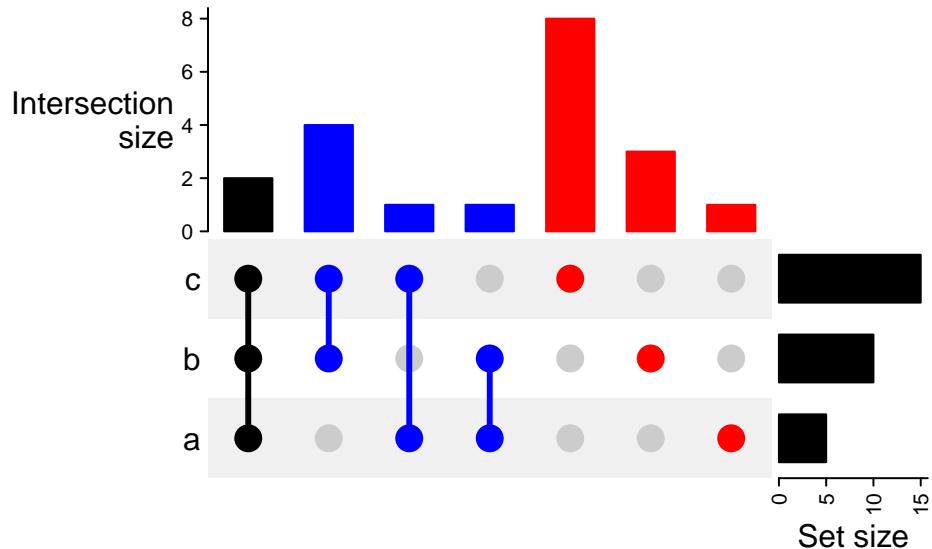
The order is controlled by `set_order` and `comb_order`:

```
UpSet(m, set_order = c("a", "b", "c"), comb_order = order(comb_size(m)))
```



Color of dots, size of dots and line width of the segments are controlled by `pt_size`, `comb_col` and `lwd`. `comb_col` should be a vector corresponding to the combination sets. In following code, since `comb_degree(m)` returns a vector of integers, we just use it as index for the color vector.

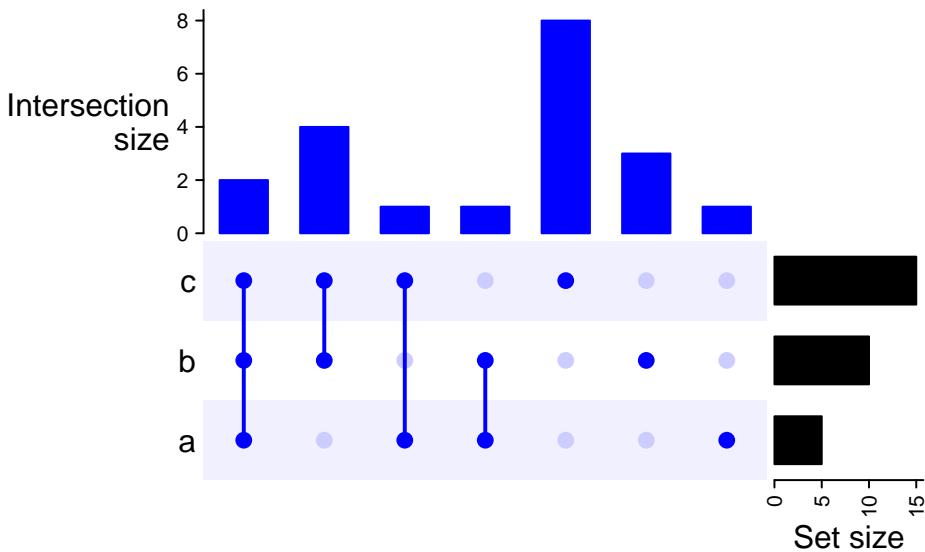
```
UpSet(m, pt_size = unit(5, "mm"), lwd = 3,
      comb_col = c("red", "blue", "black")[comb_degree(m)])
```



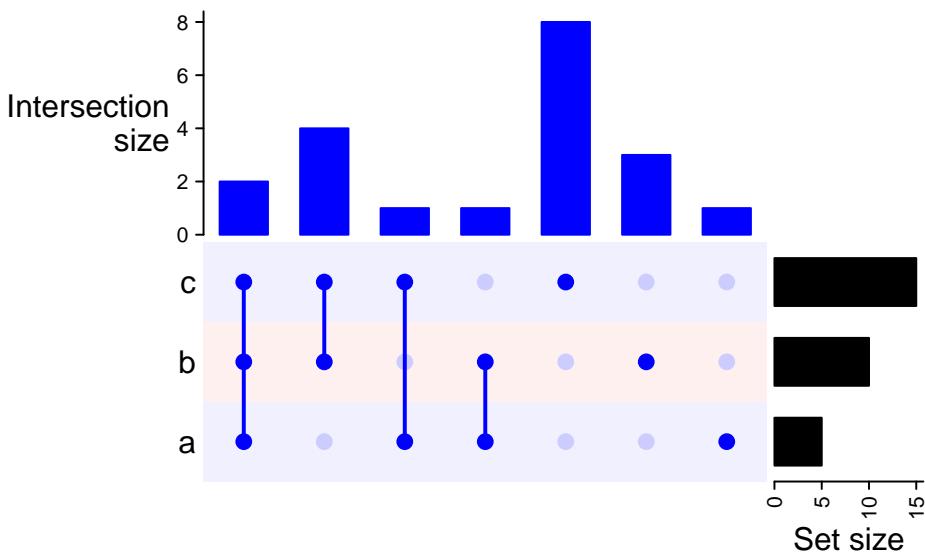
Colors for the background (the rectangles and the dots representing the set is

not selected) are controlled by `bg_col`, `bg_pt_col`. The length of `bg_col` can have length of one or two.

```
UpSet(m, comb_col = "#0000FF", bg_col = "#F0F0FF", bg_pt_col = "#CCCCFF")
```

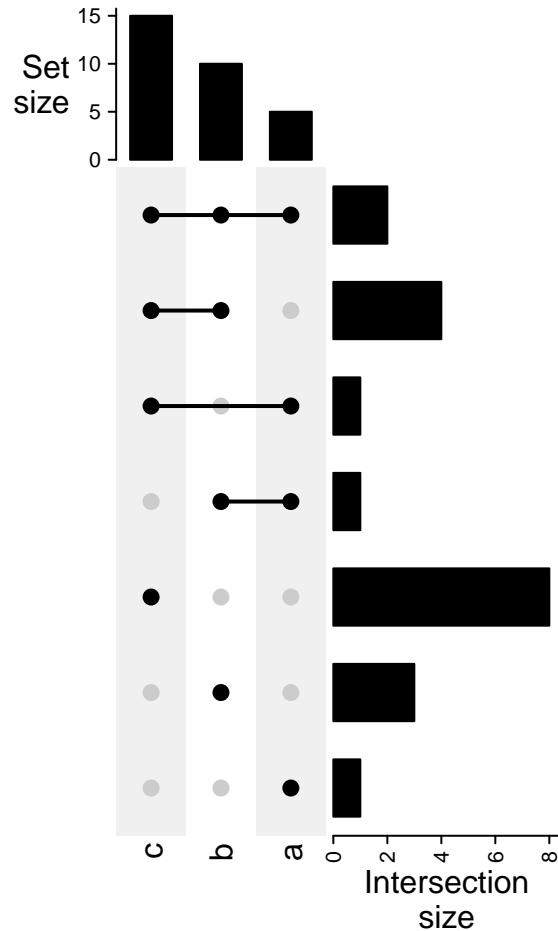


```
UpSet(m, comb_col = "#0000FF", bg_col = c("#F0F0FF", "#FFF0F0"), bg_pt_col = "#CCCCFF")
```



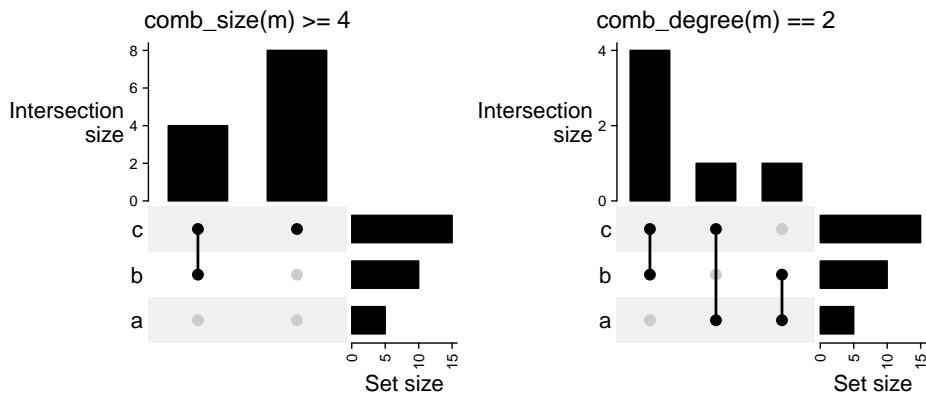
Transposing the combination matrix switches the sets to columns and combination sets to rows.

```
UpSet(t(m))
```



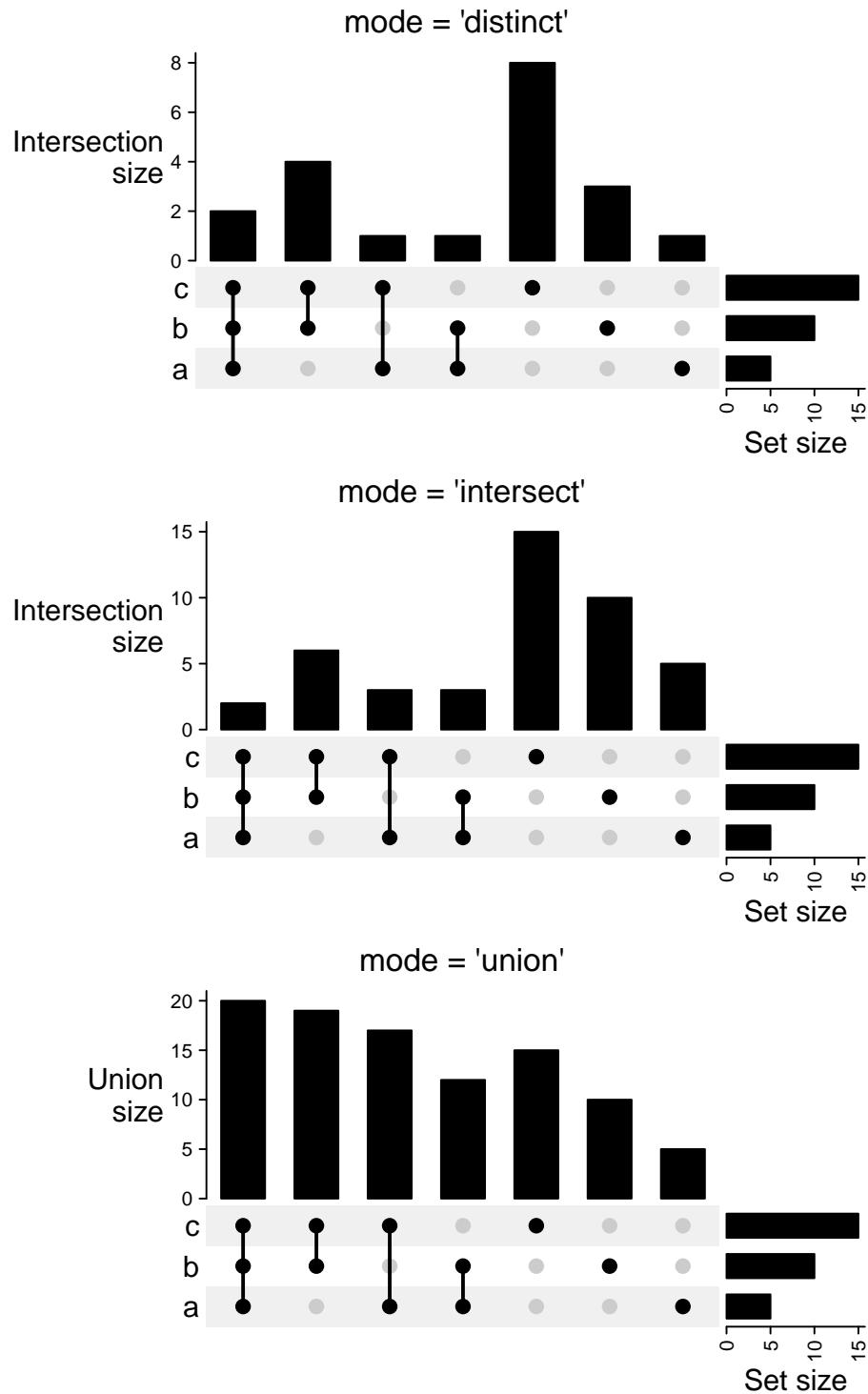
As we have introduced, if do subsetting on the combination sets, the subset of the matrix can be visualized as well:

```
UpSet(m[comb_size(m) >= 4])
UpSet(m[comb_degree(m) == 2])
```



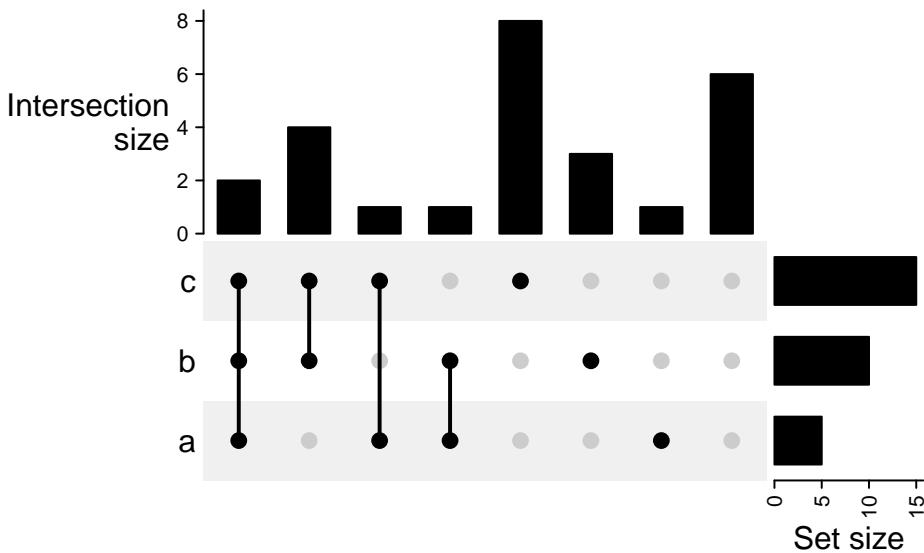
Following compares the different mode in `make_comb_mat()`:

```
m1 = make_comb_mat(lt) # the default mode is `distinct`
m2 = make_comb_mat(lt, mode = "intersect")
m3 = make_comb_mat(lt, mode = "union")
UpSet(m1)
UpSet(m2)
UpSet(m3)
```



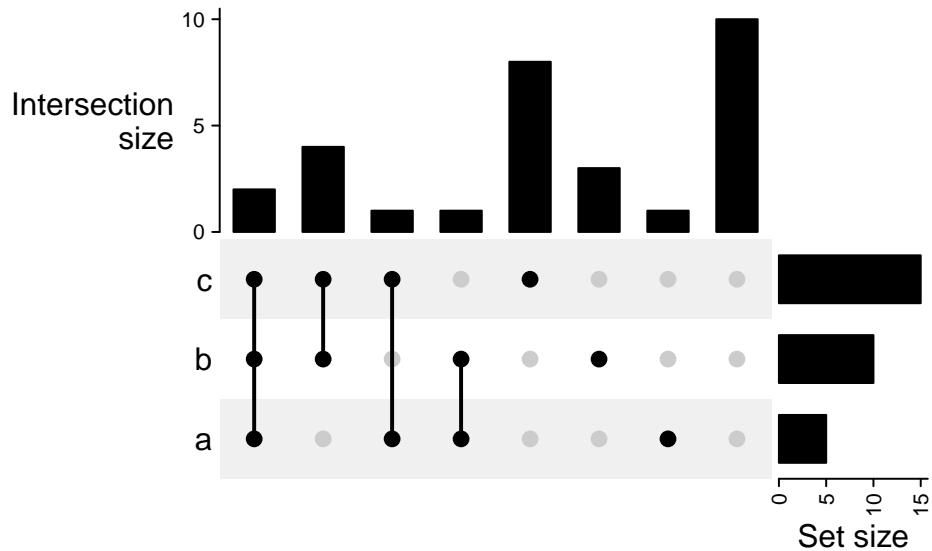
For the plot containing complement set, there is one additional column showing this complement set does not overlap to any of the sets (all dots are in grey).

```
m2 = make_comb_mat(lt, universal_set = letters)
UpSet(m2)
```



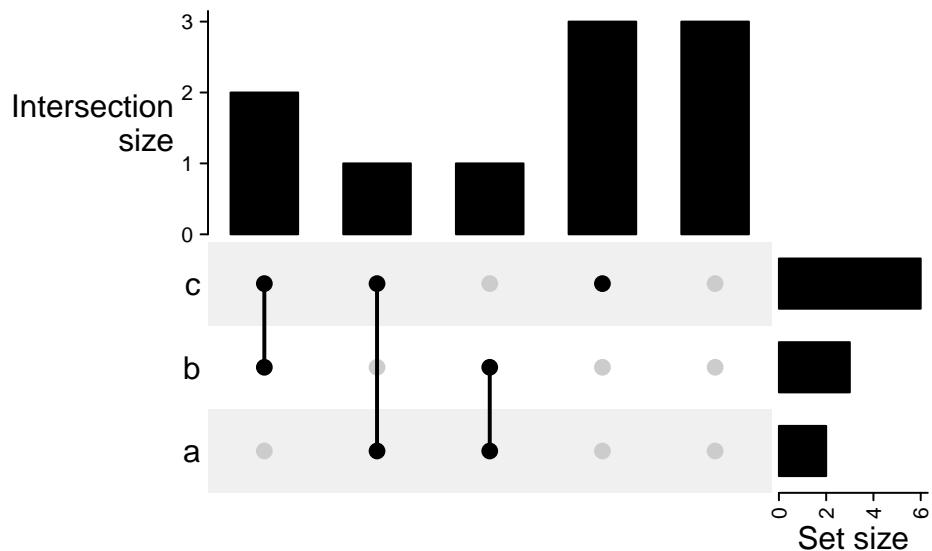
Remember if you already know the size for the complement set, you can directly assign it by `complement_size` argument in `make_comb_mat()`.

```
m2 = make_comb_mat(lt, complement_size = 10)
UpSet(m2)
```



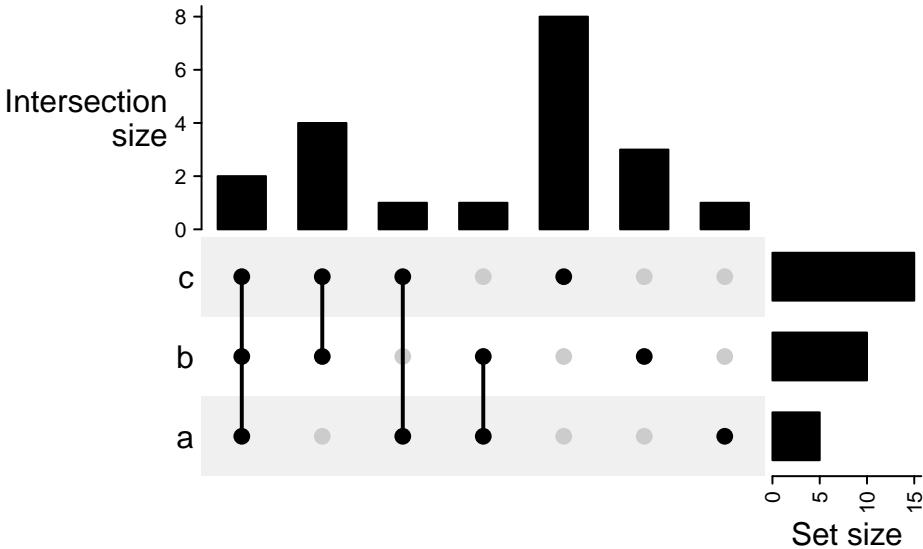
For the case where the universal set is smaller than the union of all sets:

```
m2 = make_comb_mat(lt, universal_set = letters[1:10])
UpSet(m2)
```



There are some cases that you may have complement set but you don't want to show it, especially when the input for `make_comb_mat()` is a matrix which already contains complement set, you can filter by the combination degrees.

```
x = list_to_matrix(lt, universal_set = letters)
m2 = make_comb_mat(x)
m2 = m2[comb_degree(m2) > 0]
UpSet(m2)
```



8.6 UpSet plots as heatmaps

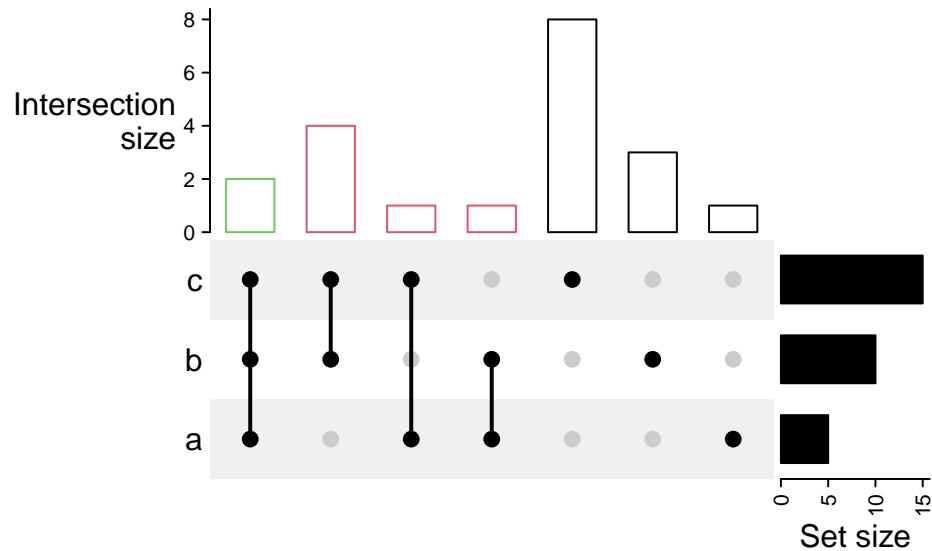
In the UpSet plot, the major component is the combination matrix, and on the two sides are the barplots representing the size of sets and the combination sets, thus, it is quite straightforward to implement it as a “heatmap” where the heatmap is self-defined with dots and segments, and the two barplots are two barplot annotations constructed by `anno_barplot()`.

The default top annotation is:

```
HeatmapAnnotation("Intersection\ncsize" = anno_barplot(comb_size(m),
    border = FALSE, gp = gpar(fill = "black"), height = unit(3, "cm")),
    annotation_name_side = "left", annotation_name_rot = 0)
```

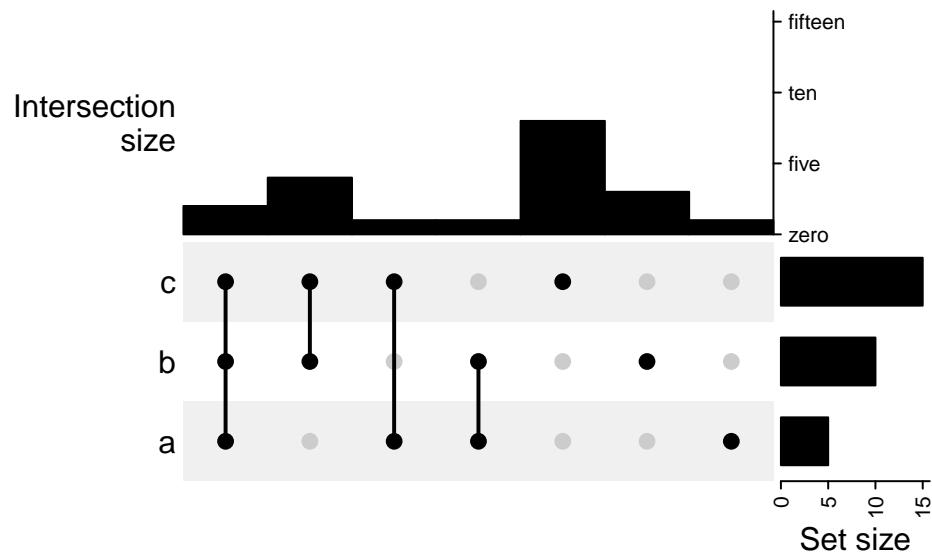
This top annotation is wrapped in `upset_top_annotation()` which only contains the upset top barplot annotation. Most of the arguments in `upset_top_annotation()` directly go to the `anno_barplot()`, e.g. to set the colors of bars:

```
UpSet(m, top_annotation = upset_top_annotation(m,
    gp = gpar(col = comb_degree(m))))
```



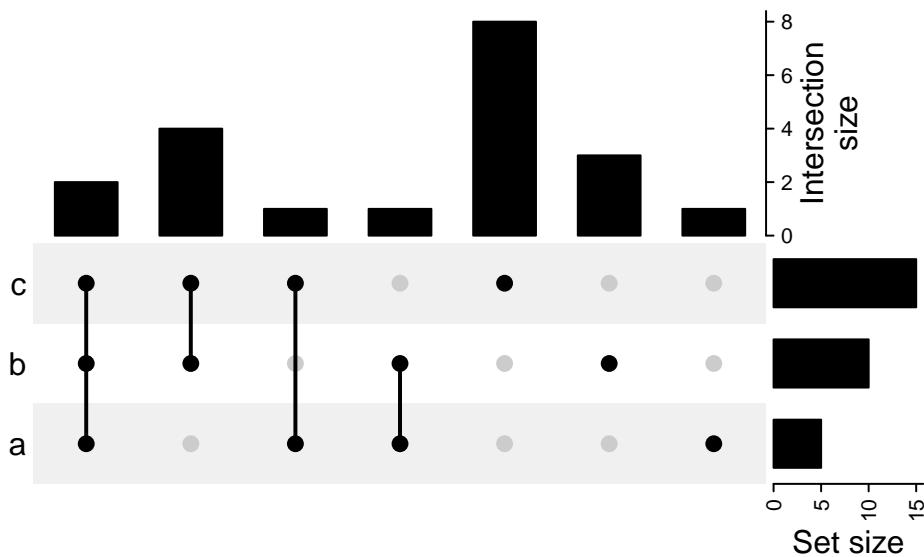
To control the data range and axis:

```
UpSet(m, top_annotation = upset_top_annotation(m,
    ylim = c(0, 15),
    bar_width = 1,
    axis_param = list(side = "right", at = c(0, 5, 10, 15),
        labels = c("zero", "five", "ten", "fifteen"))))
```



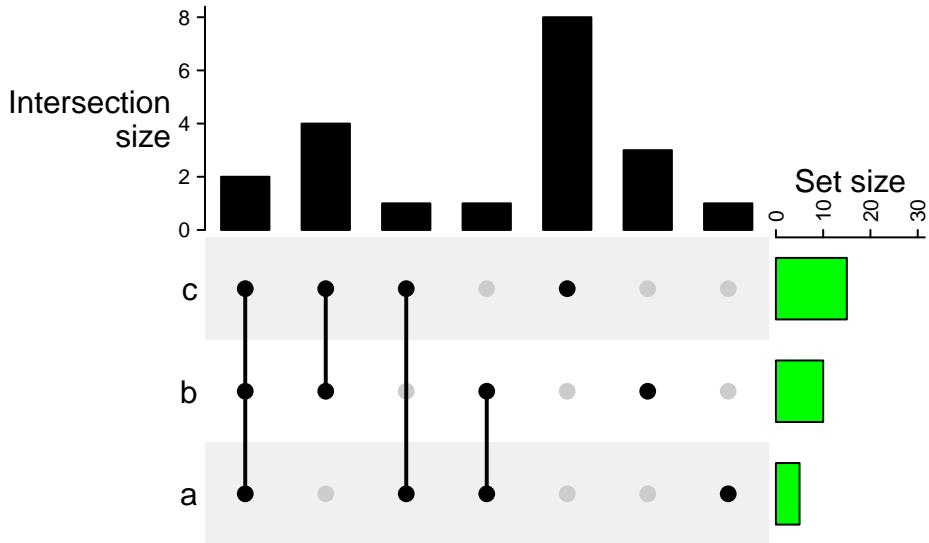
To control the annotation name:

```
UpSet(m, top_annotation = upset_top_annotation(m,
    annotation_name_rot = 90,
    annotation_name_side = "right",
    axis_param = list(side = "right")))
```



The settings are very similar for the right annotation:

```
UpSet(m, right_annotation = upset_right_annotation(m,
    ylim = c(0, 30),
    gp = gpar(fill = "green"),
    annotation_name_side = "top",
    axis_param = list(side = "top")))
```

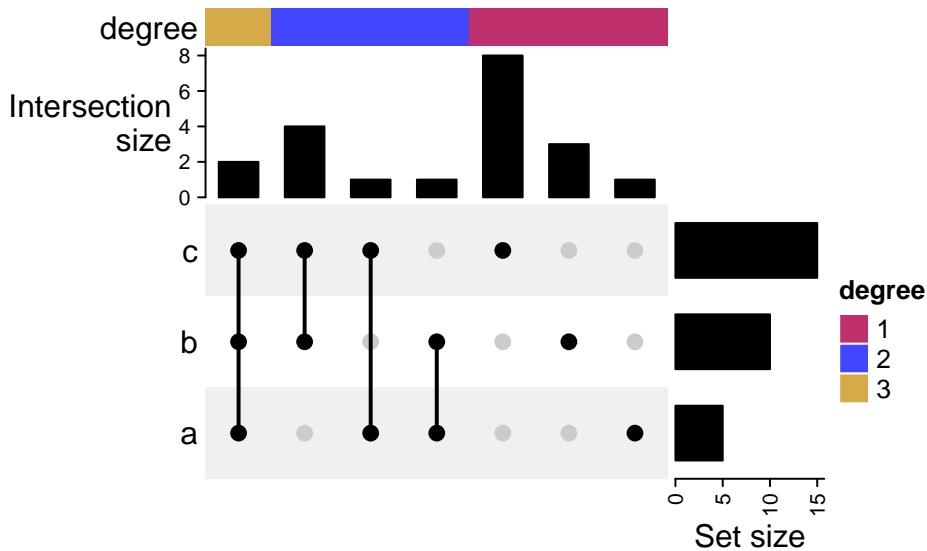


`upset_top_annotation()` and `upset_right_annotation()` can automatically recognize whether sets are on rows or columns.

`upset_top_annotation()` and `upset_right_annotation()` only contain one barplot annotation. If users want to add more annotations, they need to manually construct a `HeatmapAnnotation` object with multiple annotations.

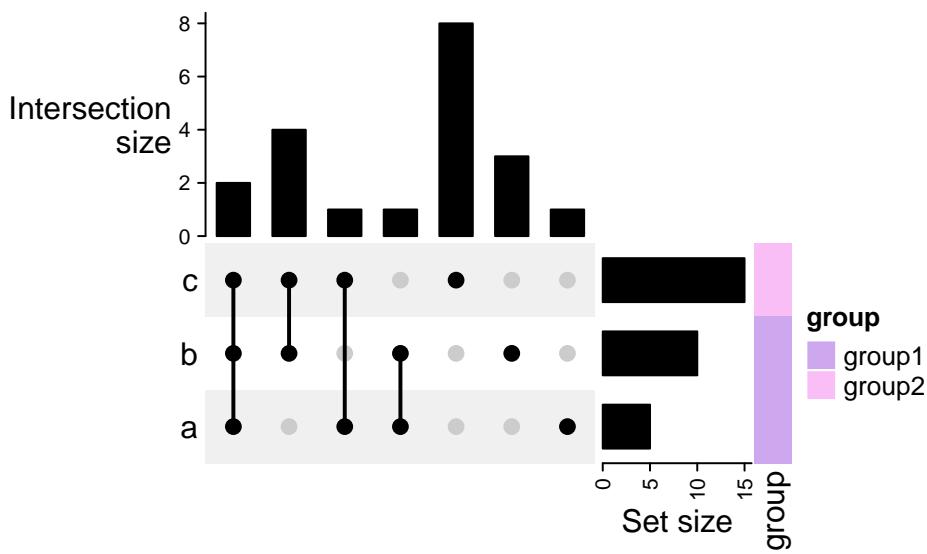
To add more annotations on top:

```
UpSet(m, top_annotation = HeatmapAnnotation(
  degree = as.character(comb_degree(m)),
  "Intersection\ncsize" = anno_barplot(comb_size(m),
    border = FALSE,
    gp = gpar(fill = "black"),
    height = unit(2, "cm")
  ),
  annotation_name_side = "left",
  annotation_name_rot = 0))
```



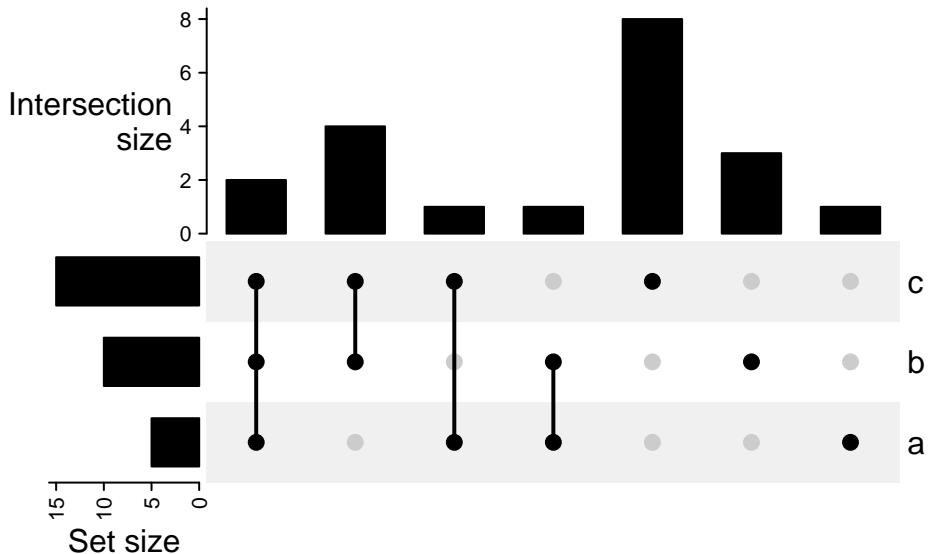
To add more annotation on the right:

```
UpSet(m, right_annotation = rowAnnotation(
  "Set size" = anno_barplot(set_size(m),
    border = FALSE,
    gp = gpar(fill = "black"),
    width = unit(2, "cm"))
),
group = c("group1", "group1", "group2"))
```



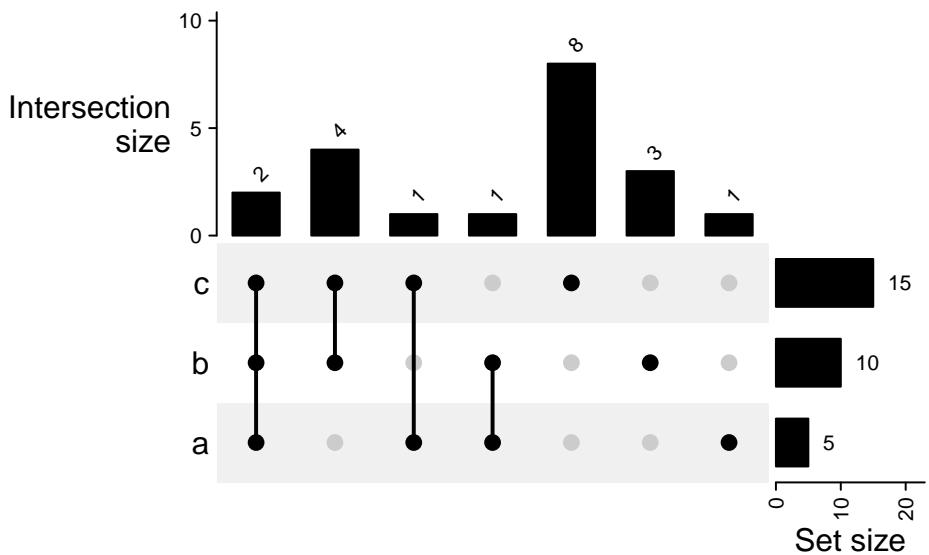
To move the right annotation to the left of the combination matrix, use `upset_left_annotation()`:

```
UpSet(m, left_annotation = upset_left_annotation(m))
```



To add numbers on top of the bars:

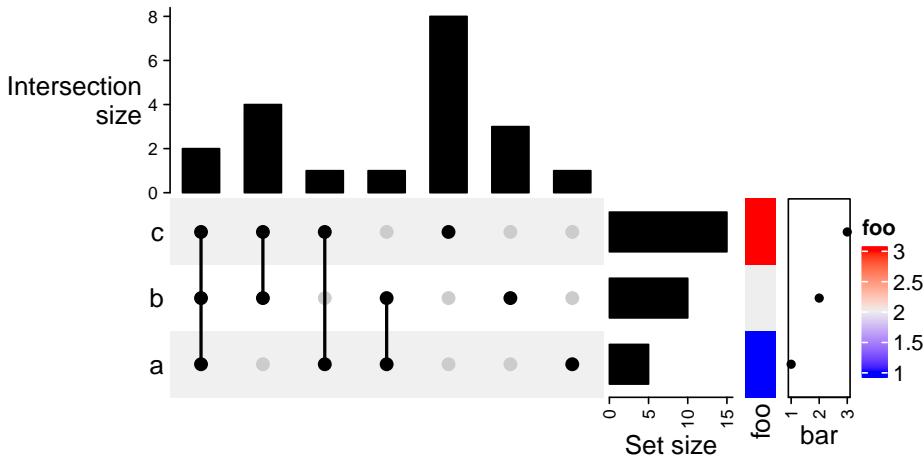
```
UpSet(m, top_annotation = upset_top_annotation(m, add_numbers = TRUE),
      right_annotation = upset_right_annotation(m, add_numbers = TRUE))
```



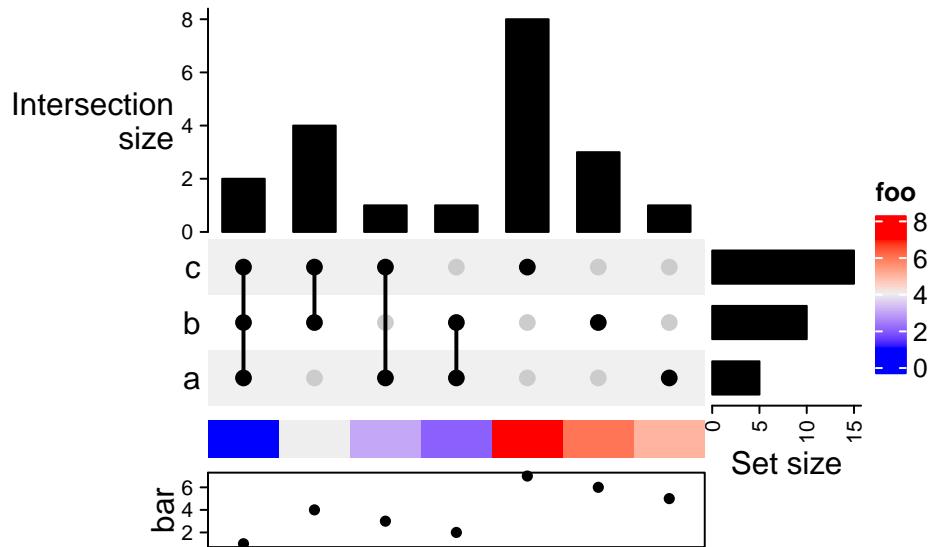
The object returned by `UpSet()` is actually a `Heatmap` class object, thus, you can add to other heatmaps and annotations by `+` or `%v%`.

```
ht = UpSet(m)
class(ht)
```

```
## [1] "Heatmap"
## attr(,"package")
## [1] "ComplexHeatmap"
ht + Heatmap(1:3, name = "foo", width = unit(5, "mm")) +
  rowAnnotation(bar = anno_points(1:3))
```

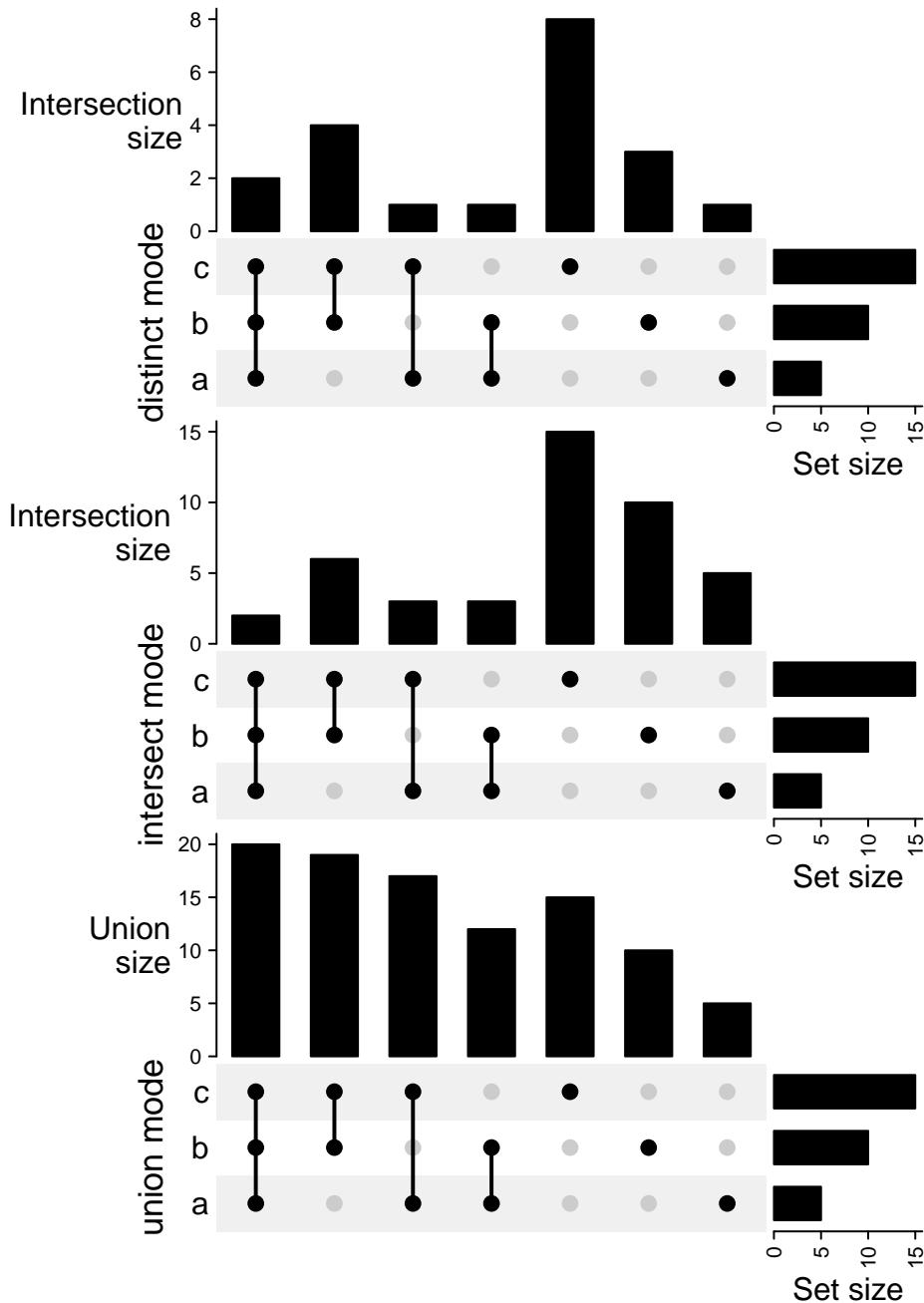


```
ht %v% Heatmap(rbind(1:7), name = "foo", row_names_side = "left",
  height = unit(5, "mm")) %v%
  HeatmapAnnotation(bar = anno_points(1:7),
    annotation_name_side = "left")
```



Add multiple UpSet plots:

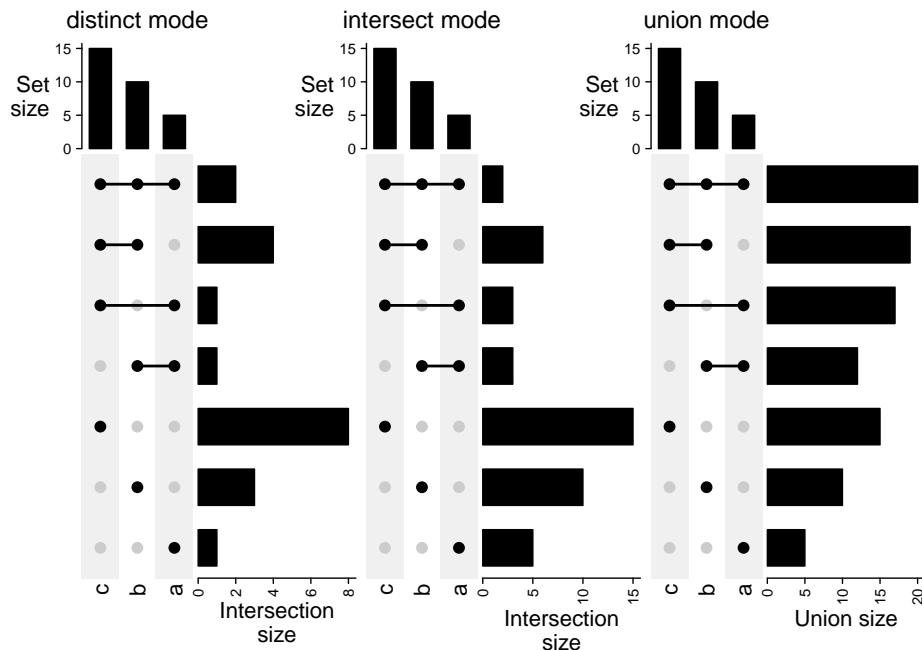
```
m1 = make_comb_mat(lt, mode = "distinct")
m2 = make_comb_mat(lt, mode = "intersect")
m3 = make_comb_mat(lt, mode = "union")
UpSet(m1, row_title = "distinct mode") %v%
    UpSet(m2, row_title = "intersect mode") %v%
    UpSet(m3, row_title = "union mode")
```



Or first transpose all the combination matrices and add them horizontally:

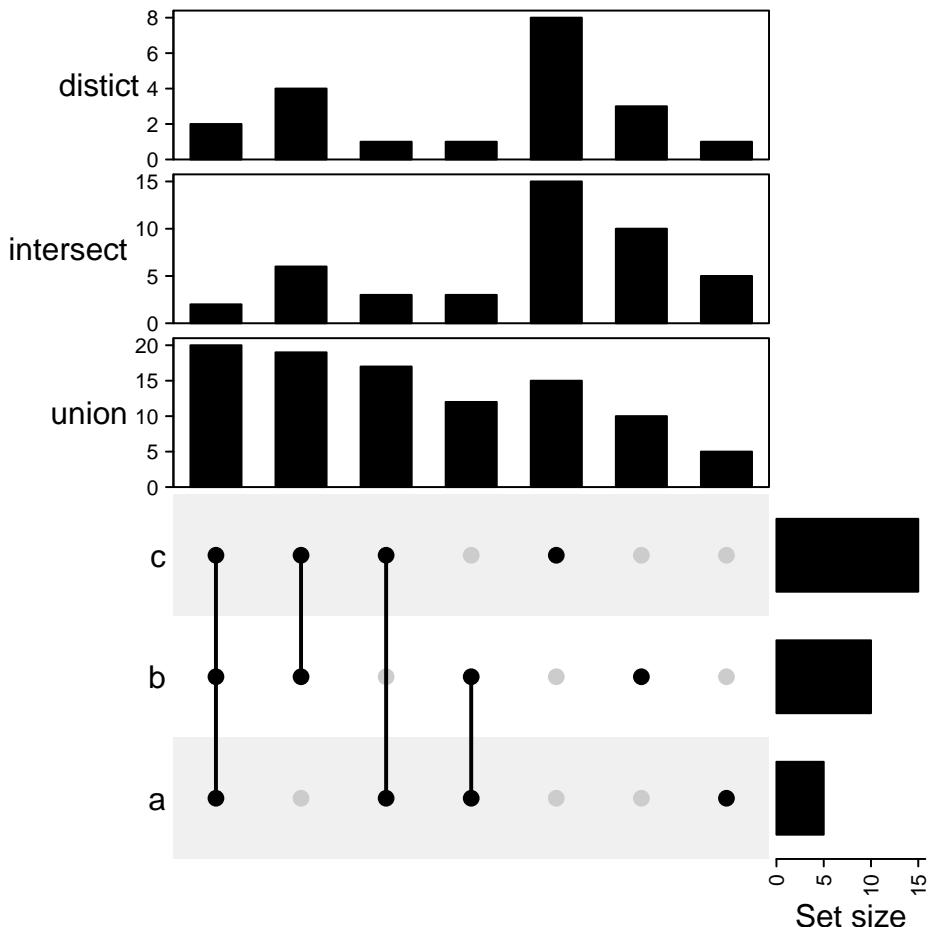
```
m1 = make_comb_mat(lt, mode = "distinct")
m2 = make_comb_mat(lt, mode = "intersect")
```

```
m3 = make_comb_mat(lt, mode = "union")
UpSet(t(m1), column_title = "distinct mode") +
  UpSet(t(m2), column_title = "intersect mode") +
  UpSet(t(m3), column_title = "union mode")
```



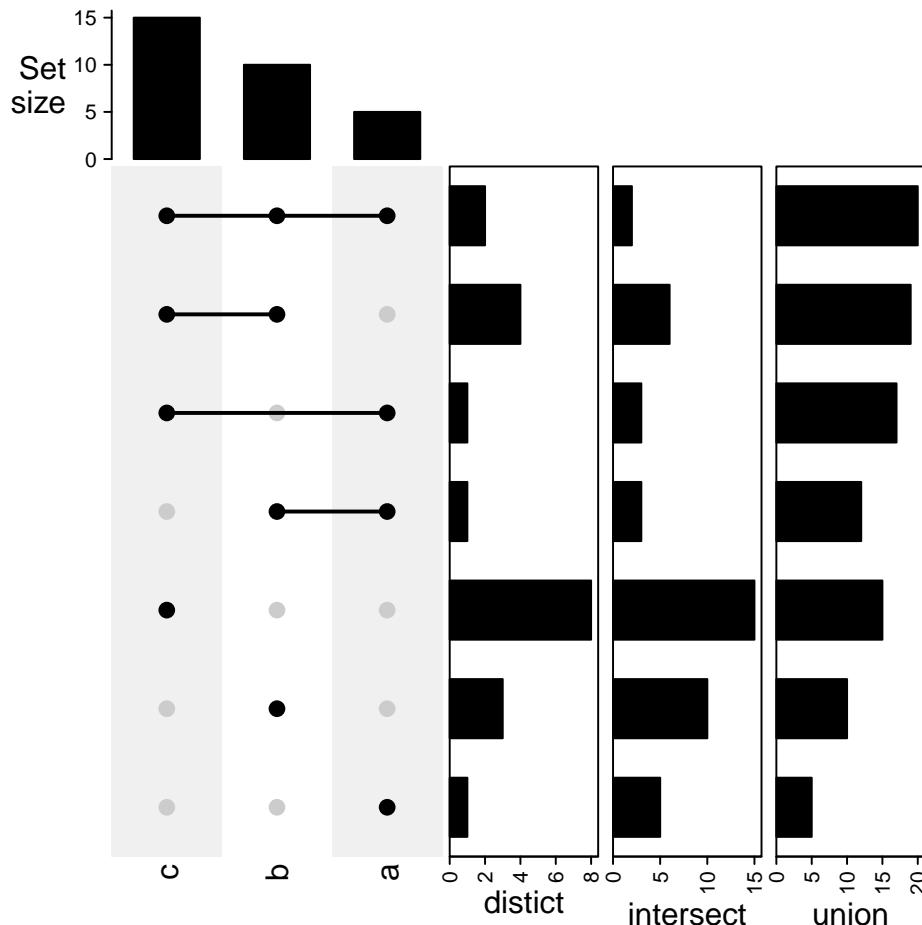
The three combination matrices are actually the same and plotting them three times is redundant. With the functionality in **ComplexHeatmap** package, we can directly add three barplot annotations.

```
top_ha = HeatmapAnnotation(
  "distinct" = anno_barplot(comb_size(m1),
    gp = gpar(fill = "black", height = unit(2, "cm"))),
  "intersect" = anno_barplot(comb_size(m2),
    gp = gpar(fill = "black", height = unit(2, "cm"))),
  "union" = anno_barplot(comb_size(m3),
    gp = gpar(fill = "black", height = unit(2, "cm"))),
  gap = unit(2, "mm"), annotation_name_side = "left",
  annotation_name_rot = 0)
# the same for using m2 or m3
UpSet(m1, top_annotation = top_ha)
```



Similar when the combination matrix is transposed:

```
right_ha = rowAnnotation(
  "distinct" = anno_barplot(comb_size(m1),
    gp = gpar(fill = "black"), width = unit(2, "cm")),
  "intersect" = anno_barplot(comb_size(m2),
    gp = gpar(fill = "black"), width = unit(2, "cm")),
  "union" = anno_barplot(comb_size(m3),
    gp = gpar(fill = "black"), width = unit(2, "cm")),
  gap = unit(2, "mm"), annotation_name_side = "bottom")
# the same for using m2 or m3
UpSet(t(m1), right_annotation = right_ha)
```



8.7 Example with the movies dataset

UpsetR package also provides a `movies` dataset, which contains 17 genres for 3883 movies. First load the dataset.

```
movies = read.csv(system.file("extdata", "movies.csv", package = "UpSetR"),
                  header = TRUE, sep = ";")
head(movies) # `make_comb_mat()` automatically ignores the first two columns
```

	Name	ReleaseDate	Action	Adventure	Children
## 1	Toy Story (1995)	1995	0	0	1
## 2	Jumanji (1995)	1995	0	1	1
## 3	Grumpier Old Men (1995)	1995	0	0	0
## 4	Waiting to Exhale (1995)	1995	0	0	0
## 5	Father of the Bride Part II (1995)	1995	0	0	0
## 6	Heat (1995)	1995	1	0	0

```

##   Comedy Crime Documentary Drama Fantasy Noir Horror Musical Mystery Romance
## 1     1     0         0     0     0     0     0     0     0     0     0
## 2     0     0         0     0     1     0     0     0     0     0     0
## 3     1     0         0     0     0     0     0     0     0     0     1
## 4     1     0         0     1     0     0     0     0     0     0     0
## 5     1     0         0     0     0     0     0     0     0     0     0
## 6     0     1         0     0     0     0     0     0     0     0     0

##   SciFi Thriller War Western AvgRating Watches
## 1     0     0     0     0     4.15    2077
## 2     0     0     0     0     3.20     701
## 3     0     0     0     0     3.02    478
## 4     0     0     0     0     2.73    170
## 5     0     0     0     0     3.01    296
## 6     0     1     0     0     3.88    940

```

To make a same UpSet plot as in this vignette:

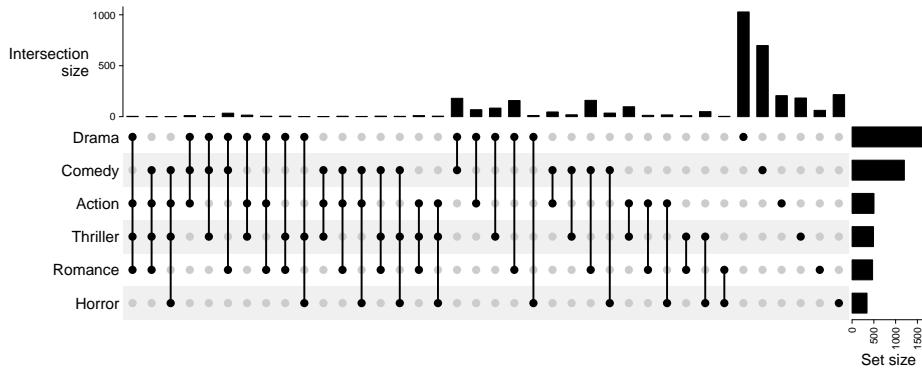
```

m = make_comb_mat(movies, top_n_sets = 6)
m

## A combination matrix with 6 sets and 39 combinations.
## ranges of combination set size: c(1, 1028).
## mode for the combination size: distinct.
## sets are on rows.
##
## Top 8 combination sets are:
##   Action Comedy Drama Horror Romance Thriller   code size
##           x          001000 1028
##           x          010000  698
##           x          000100  216
##           x          100000  206
##           x 000001 183
##           x   x 011000 180
##           x           010010 160
##           x           001010 158
##
## Sets are:
##   set size
##   Action 503
##   Comedy 1200
##   Drama 1603
##   Horror 343
##   Romance 471
##   Thriller 492
##   complement 2

```

```
m = m[comb_degree(m) > 0]
UpSet(m)
```



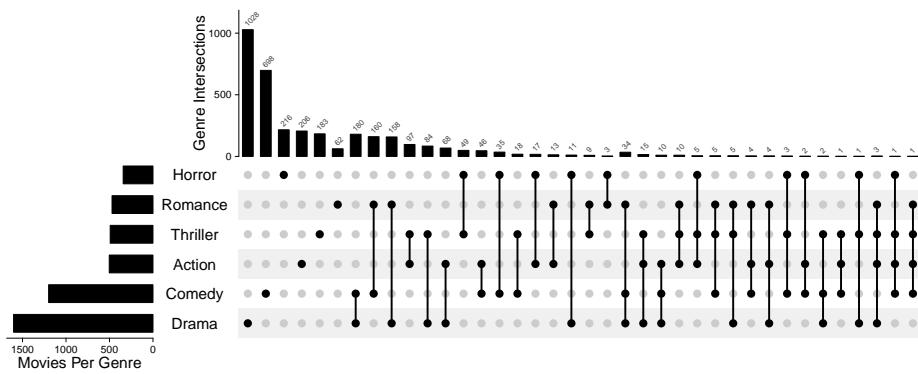
Following code makes it look more similar as the original plot. The code is a little bit long, but most of the code mainly customize the annotations and row/column orders.

```
ss = set_size(m)
cs = comb_size(m)
ht = UpSet(m,
            set_order = order(ss),
            comb_order = order(comb_degree(m), -cs),
            top_annotation = HeatmapAnnotation(
                "Genre Intersections" = anno_barplot(cs,
                                                       ylim = c(0, max(cs)*1.1),
                                                       border = FALSE,
                                                       gp = gpar(fill = "black"),
                                                       height = unit(4, "cm"))
            ),
            annotation_name_side = "left",
            annotation_name_rot = 90),
            left_annotation = rowAnnotation(
                "Movies Per Genre" = anno_barplot(-ss,
                                                    baseline = 0,
                                                    axis_param = list(
                                                        at = c(0, -500, -1000, -1500),
                                                        labels = c(0, 500, 1000, 1500),
                                                        labels_rot = 0),
                                                    border = FALSE,
                                                    gp = gpar(fill = "black"),
                                                    width = unit(4, "cm"))
            ),
            set_name = anno_text(set_name(m),
                                 location = 0.5,
```

```

        just = "center",
        width = max_text_width(set_name(m)) + unit(4, "mm"))
),
right_annotation = NULL,
show_row_names = FALSE)
ht = draw(ht)
od = column_order(ht)
decorate_annotation("Genre Intersections", {
  grid.text(cs[od], x = seq_along(cs), y = unit(cs[od], "native") + unit(2, "pt"),
            default.units = "native", just = c("left", "bottom"),
            gp = gpar(fontsize = 6, col = "#404040"), rot = 45)
})

```



In movies dataset, there is also one column `AvgRating` which gives the rating of each movie, we next split all the movies into five groups based on the ratings.

```

genre = c("Action", "Romance", "Horror", "Children", "SciFi", "Documentary")
rating = cut(movies$AvgRating, c(0, 1, 2, 3, 4, 5))
m_list = tapply(seq_len(nrow(movies)), rating, function(ind) {
  m = make_comb_mat(movies[ind, genre, drop = FALSE])
  m[comb_degree(m) > 0]
})

```

The combination matrices in `m_list` might have different combination sets:

```

sapply(m_list, comb_size)

## $`(0,1]`  

## 010000 001000 000100 000001  

##      1      2      1      1  

##  

## $`(1,2]`  

## 101010 100110 110000 101000 100100 100010 001010 100000 010000 001000 000100  

##      1      1      1      4      5      5      8     14      7     38     14  

## 000010 000001

```

```

##      3      2
##
## $`(2,3]` 
## 101010 110000 101000 100100 100010 010100 010010 001010 000110 100000 010000
##      4      8      2      6     35      3      1     27      7     126      99
## 001000 000100 000010 000001
## 142     77     27      9
##
## $`(3,4]` 
## 110010 101010 100110 110000 101000 100010 011000 010100 010010 001100 001010
##      1      6      1     20      6     45      3      4      4      1      11
## 000110 100000 010000 001000 000100 000010 000001
##      5     176     276     82    122      66     87
##
## $`(4,5]` 
## 110010 101010 110000 101000 100010 100000 010000 001000 000100 000010 000001
##      1      1      4      1      6     23     38      4      4     10     28

```

To compare between multiple groups with UpSet plots, we need to normalize all the matrices to make them have same sets and same combination sets. `normalize_comb_mat()` basically adds zero to the new combination sets which were not there before.

```

m_list = normalize_comb_mat(m_list)
sapply(m_list, comb_size)

```

```

##      (0,1] (1,2] (2,3] (3,4] (4,5]
## 110001    0    1    0    1    0
## 100101    0    1    4    6    1
## 100011    0    0    0    1    1
## 110000    0    5    6    0    0
## 100100    0    4    2    6    1
## 100010    0    1    8   20    4
## 100001    0    5   35   45    6
## 010100    0    0    0    1    0
## 010010    0    0    3    4    0
## 010001    0    0    7    5    0
## 000110    0    0    0    3    0
## 000101    0    8   27   11    0
## 000011    0    0    1    4    0
## 100000    0   14  126  176   23
## 010000    1   14   77  122    4
## 001000    1    2    9   87   28
## 000100    2   38  142   82    4
## 000010    1    7   99  276   38
## 000001    0    3   27   66   10

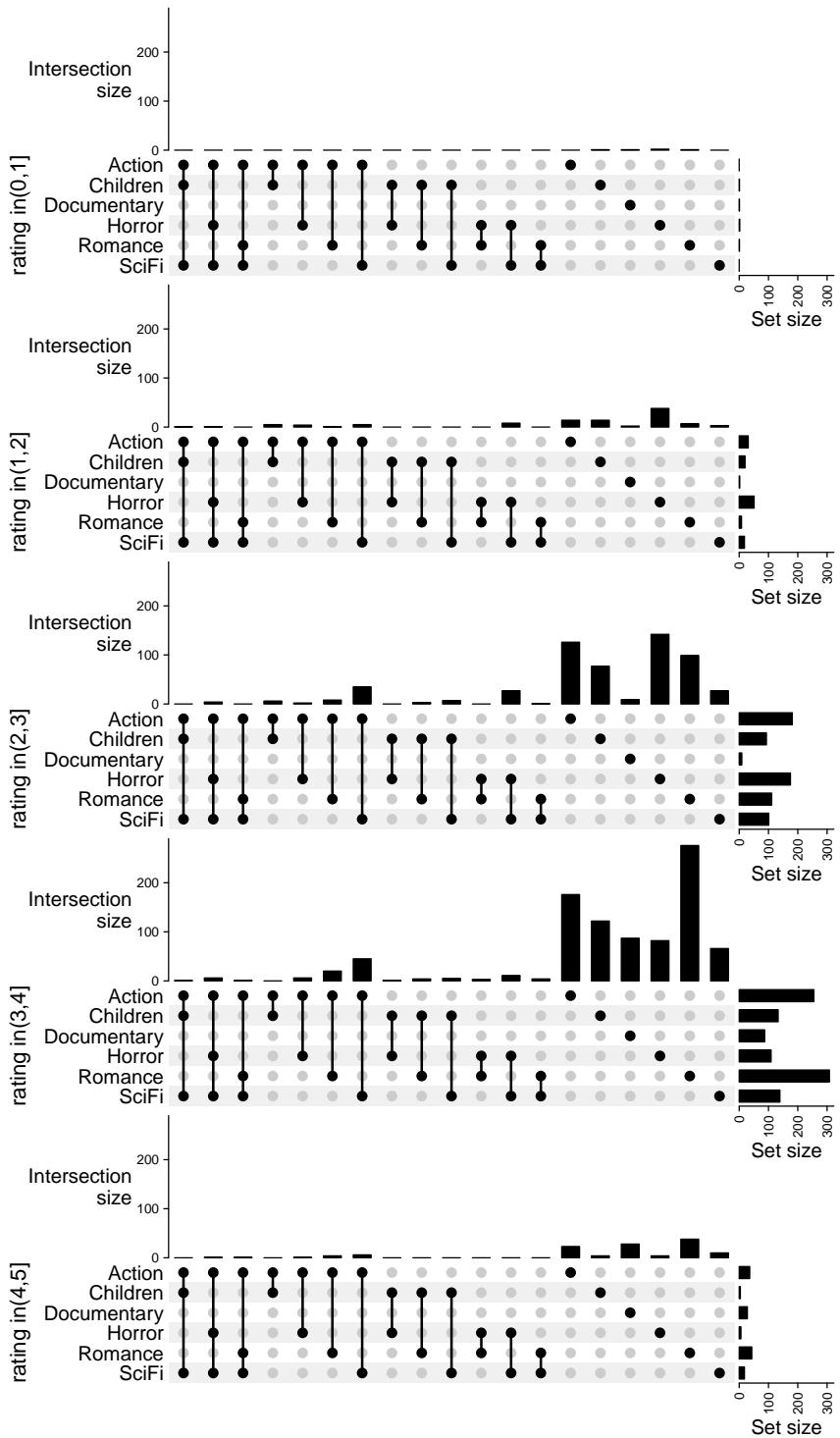
```

We calculate the range for the two barplots:

```
max_set_size = max(sapply(m_list, set_size))
max_comb_size = max(sapply(m_list, comb_size))
```

And finally we add the five UpSet plots vertically:

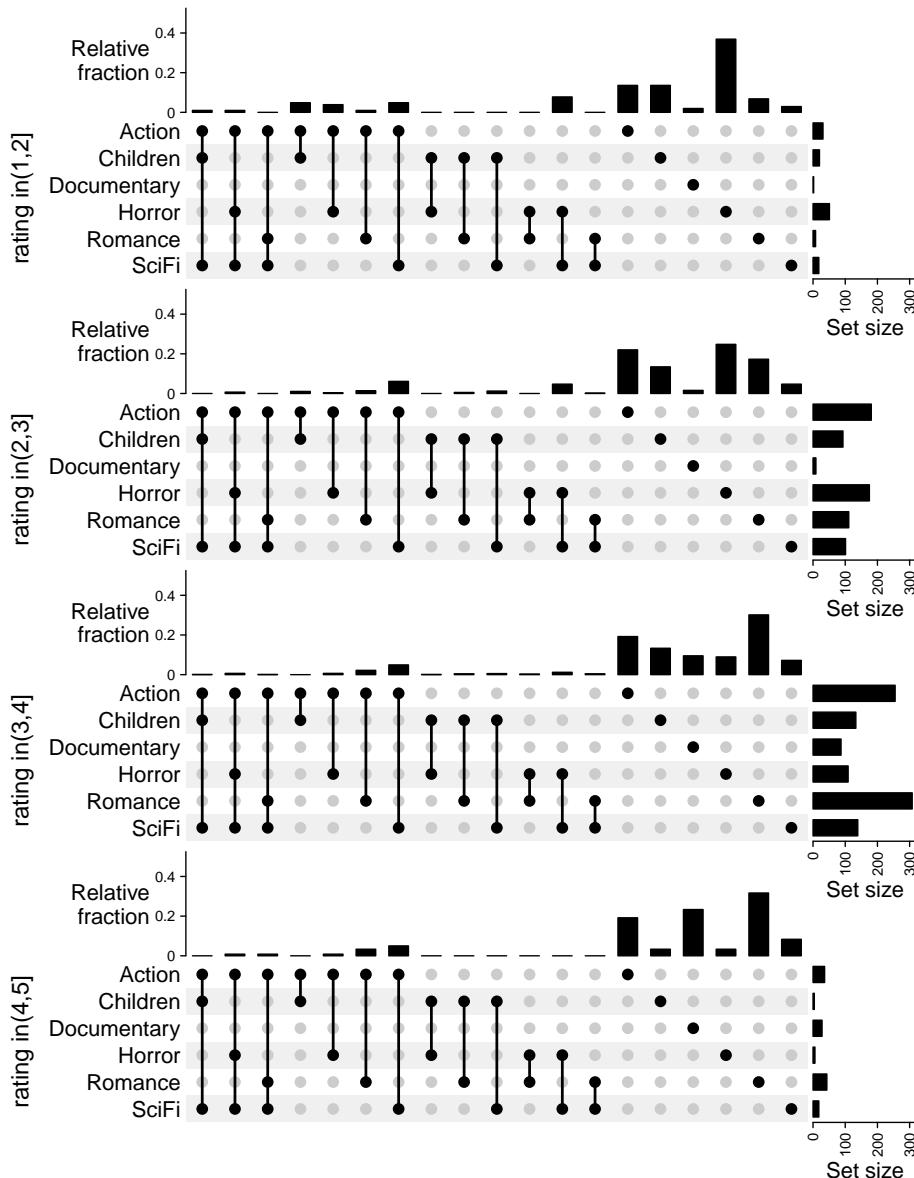
```
ht_list = NULL
for(i in seq_along(m_list)) {
  ht_list = ht_list %v%
    UpSet(m_list[[i]], row_title = paste0("rating in", names(m_list)[i]),
          set_order = NULL, comb_order = NULL,
          top_annotation = upset_top_annotation(m_list[[i]], ylim = c(0, max_comb_size)),
          right_annotation = upset_right_annotation(m_list[[i]], ylim = c(0, max_set_size)))
}
ht_list
```



After comparing the five UpSet plots, we can see most of the movies are rated between 2 and 4. Horror movies tend to have lower ratings and romance movies tend to have higher ratings.

Instead of directly comparing the size of the combination sets, we can also compare the relative fraction to the full sets. In following code, we remove the group of `c(0, 1)` because the number of movies are too few there.

```
m_list = m_list[-1]
max_set_size = max(sapply(m_list, set_size))
rel_comb_size = sapply(m_list, function(m) {
  s = comb_size(m)
  # because the combination matrix is generated under "distinct" mode
  # the sum of `s` is the size of the full set
  s/sum(s)
})
ht_list = NULL
for(i in seq_along(m_list)) {
  ht_list = ht_list %v%
    UpSet(m_list[[i]], row_title = paste0("rating in", names(m_list)[i]),
          set_order = NULL, comb_order = NULL,
          top_annotation = HeatmapAnnotation(
            "Relative\\nfraction" = anno_barplot(
              rel_comb_size[, i],
              ylim = c(0, 0.5),
              gp = gpar(fill = "black"),
              border = FALSE,
              height = unit(2, "cm"),
            ),
            annotation_name_side = "left",
            annotation_name_rot = 0),
          right_annotation = upset_right_annotation(m_list[[i]],
            ylim = c(0, max_set_size)))
  )
}
ht_list
```



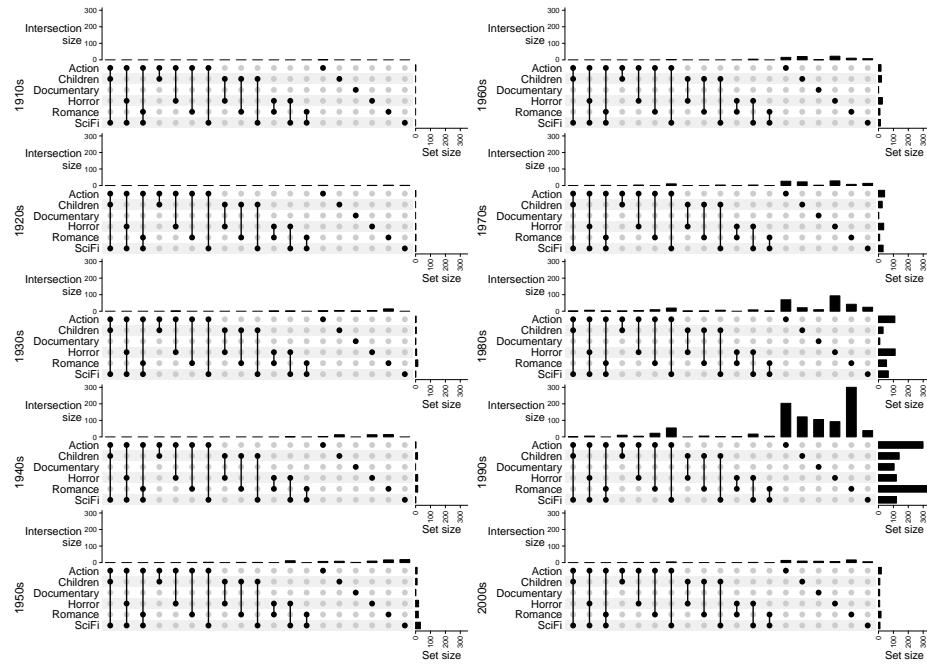
Now the trend is more clear that horror movies are rated low and documentaries are rated high.

Next we split the movies by years:

```
year = floor(movies$ReleaseDate/10)*10
m_list = tapply(seq_len(nrow(movies)), year, function(ind) {
  m = make_comb_mat(movies[ind, genre, drop = FALSE])
  m[comb_degree(m) > 0]
```

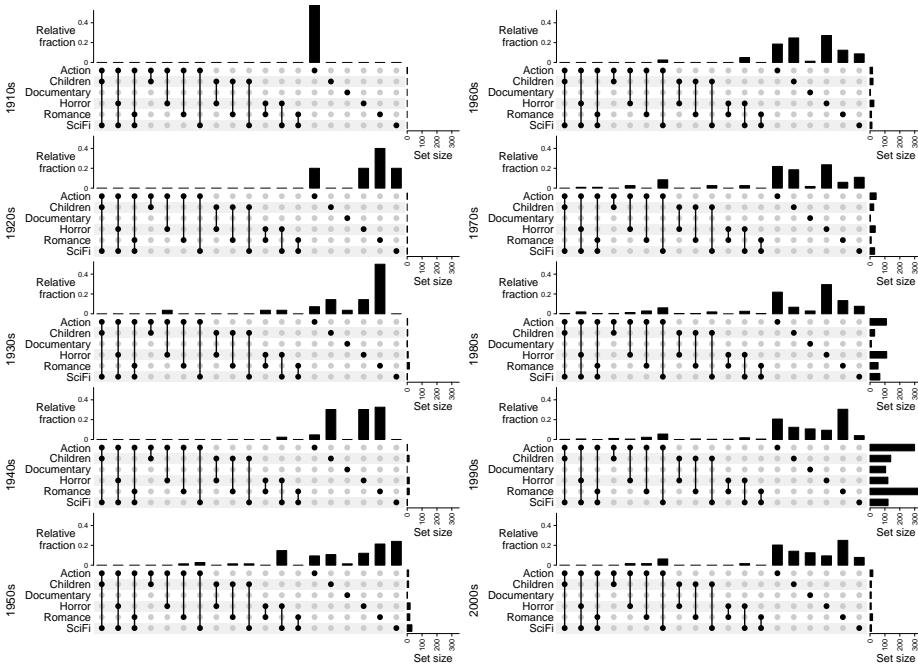
```
})
m_list = normalize_comb_mat(m_list)
max_set_size = max(sapply(m_list, set_size))
max_comb_size = max(sapply(m_list, comb_size))
ht_list1 = NULL
for(i in 1:5) {
  ht_list1 = ht_list1 %v%
    UpSet(m_list[[i]], row_title = paste0(names(m_list)[i], "s"),
      set_order = NULL, comb_order = NULL,
      top_annotation = upset_top_annotation(m_list[[i]], ylim = c(0, max_comb_size),
        height = unit(2, "cm")),
      right_annotation = upset_right_annotation(m_list[[i]], ylim = c(0, max_set_size)))
}

ht_list2 = NULL
for(i in 6:10) {
  ht_list2 = ht_list2 %v%
    UpSet(m_list[[i]], row_title = paste0(names(m_list)[i], "s"),
      set_order = NULL, comb_order = NULL,
      top_annotation = upset_top_annotation(m_list[[i]], ylim = c(0, max_comb_size),
        height = unit(2, "cm")),
      right_annotation = upset_right_annotation(m_list[[i]], ylim = c(0, max_set_size)))
}
grid.newpage()
pushViewport(viewport(x = 0, width = 0.5, just = "left"))
draw(ht_list1, newpage = FALSE)
popViewport()
pushViewport(viewport(x = 0.5, width = 0.5, just = "left"))
draw(ht_list2, newpage = FALSE)
popViewport()
```



Now we can see most of the movies were produced in 1990s and the two major genres are actions and romance.

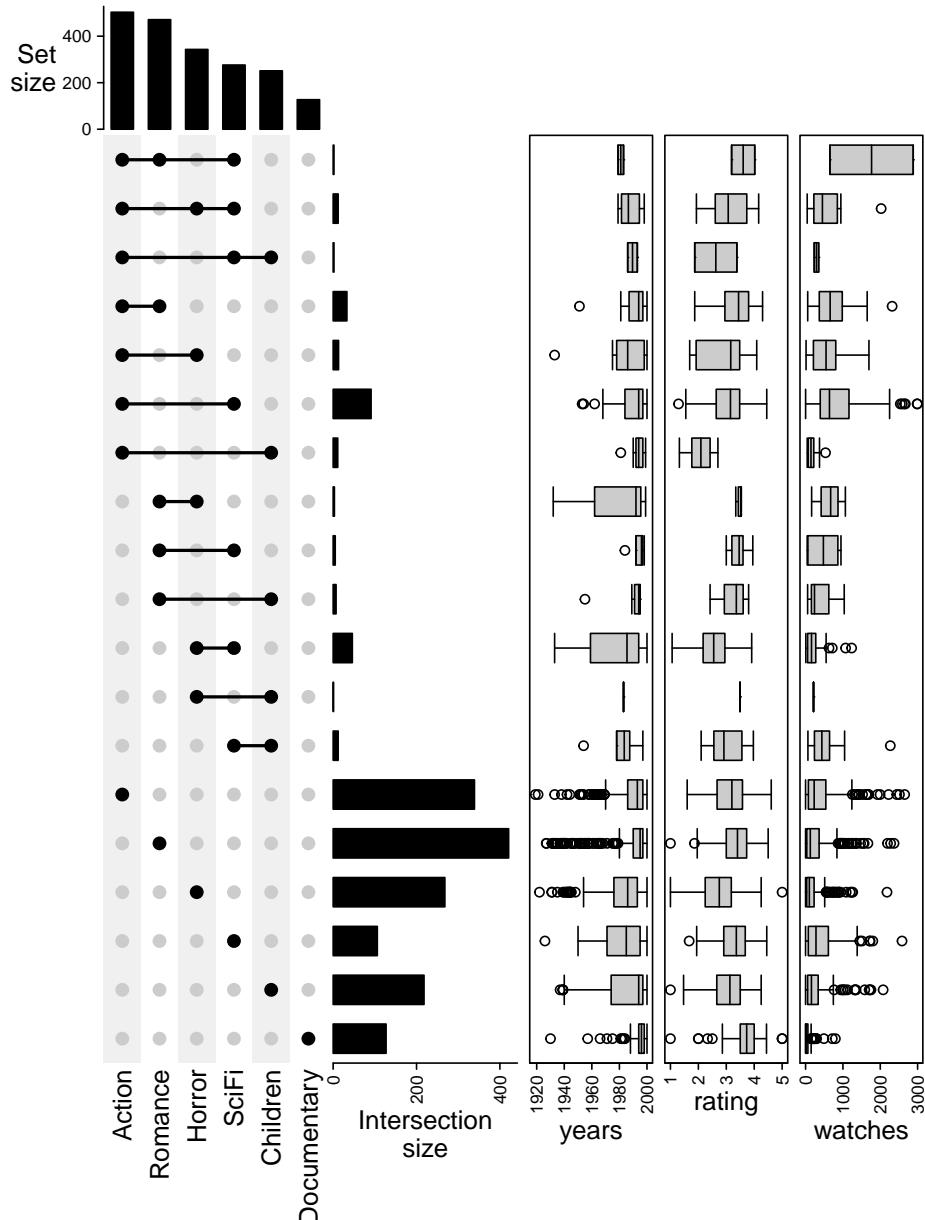
Similarly, if we change the top annotation to the relative fraction to the full sets (code not shown):



Finally we can add the statistics of years, ratings and number of watches for each combination set as boxplot annotations to the right of the UpSet plot.

```
m = make_comb_mat(movies[, genre])
m = m[comb_degree(m) > 0]
comb_elements = lapply(comb_name(m), function(nm) extract_comb(m, nm))
years = lapply(comb_elements, function(ind) movies$ReleaseDate[ind])
rating = lapply(comb_elements, function(ind) movies$AvgRating[ind])
watches = lapply(comb_elements, function(ind) movies$Watches[ind])

UpSet(t(m)) + rowAnnotation(years = anno_boxplot(years),
  rating = anno_boxplot(rating),
  watches = anno_boxplot(watches),
  gap = unit(2, "mm"))
```



We can see the movies with genre “Scifi + Children” were produced quite old but the ratings are not bad. The movies with genre “Action + Children” have the lowest ratings.

8.8 Example with the genomic regions

The H3K4me3 ChIP-seq peaks from six Roadmap samples are visualized by UpSet plot. The six samples are:

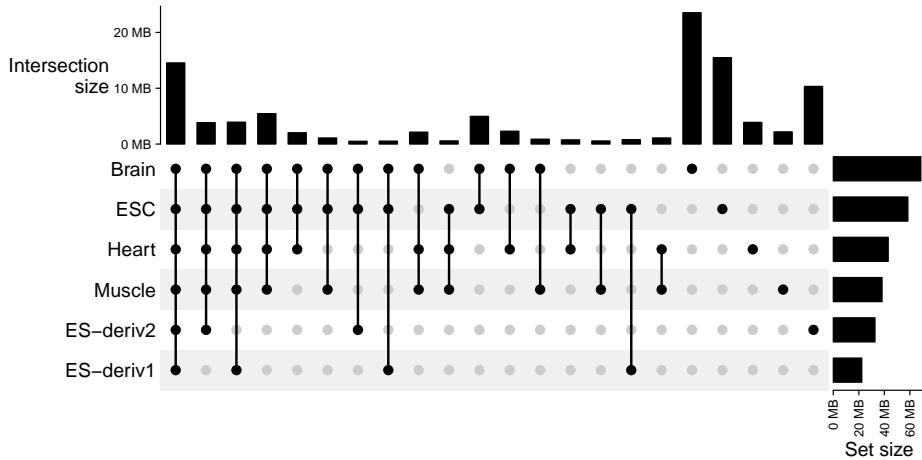
- ESC, E016
- ES-derived, E004
- ES-derived, E006
- Brain, E071
- Muscle, E100
- Heart, E104

First read the files and convert to `GRanges` objects.

```
file_list = c(
  "ESC" = "data/E016-H3K4me3.narrowPeak.gz",
  "ES-deriv1" = "data/E004-H3K4me3.narrowPeak.gz",
  "ES-deriv2" = "data/E006-H3K4me3.narrowPeak.gz",
  "Brain" = "data/E071-H3K4me3.narrowPeak.gz",
  "Muscle" = "data/E100-H3K4me3.narrowPeak.gz",
  "Heart" = "data/E104-H3K4me3.narrowPeak.gz"
)
library(GenomicRanges)
peak_list = lapply(file_list, function(f) {
  df = read.table(f)
  GRanges(seqnames = df[, 1], ranges = IRanges(df[, 2], df [, 3]))
})
```

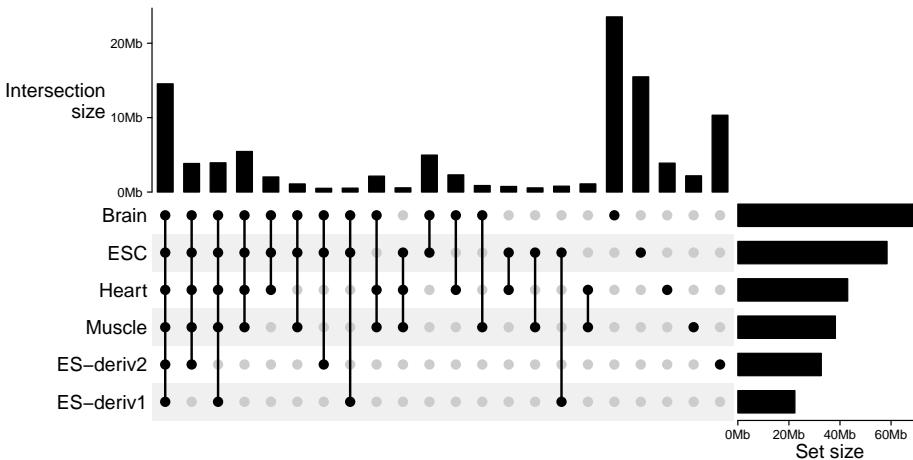
Make the combination matrix. Note now the size of the sets and the combination sets are **total base pairs or the sum of width of the regions**. We only keep the combination sets with more than 500kb.

```
m = make_comb_mat(peak_list)
m = m[comb_size(m) > 500000]
UpSet(m)
```



We can nicely format the axis labels by setting `axis_param`:

```
UpSet(m,
  top_annotation = upset_top_annotation(
    m,
    axis_param = list(at = c(0, 1e7, 2e7),
                      labels = c("0Mb", "10Mb", "20Mb")),
    height = unit(4, "cm")
  ),
  right_annotation = upset_right_annotation(
    m,
    axis_param = list(at = c(0, 2e7, 4e7, 6e7),
                      labels = c("0Mb", "20Mb", "40Mb", "60Mb"),
                      labels_rot = 0),
    width = unit(4, "cm")
  )
)
```



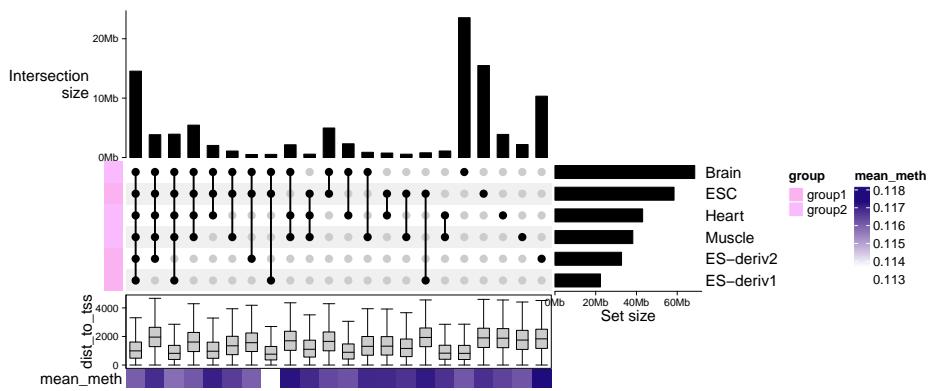
With each set of genomic regions, we can associate more information to it, such as the mean methylation or the distance to nearest TSS.

```
subgroup = c("ESC" = "group1",
           "ES-deriv1" = "group1",
           "ES-deriv2" = "group1",
           "Brain" = "group2",
           "Muscle" = "group2",
           "Heart" = "group2"
)
comb_sets = lapply(comb_name(m), function(nm) extract_comb(m, nm))
comb_sets = lapply(comb_sets, function(gr) {
  # we just randomly generate dist_to_tss and mean_meth
  gr$dist_to_tss = abs(rnorm(length(gr), mean = runif(1, min = 500, max = 2000), sd = 1000))
  gr$mean_meth = abs(rnorm(length(gr), mean = 0.1, sd = 0.1))
  gr
})
UpSet(m,
      top_annotation = upset_top_annotation(
        m,
        axis_param = list(at = c(0, 1e7, 2e7),
                          labels = c("0Mb", "10Mb", "20Mb")),
        height = unit(4, "cm")
      ),
      right_annotation = upset_right_annotation(
        m,
        axis_param = list(at = c(0, 2e7, 4e7, 6e7),
                          labels = c("0Mb", "20Mb", "40Mb", "60Mb"),
                          labels_rot = 0),
        width = unit(4, "cm")
      ),
      
```

```

left_annotation = rowAnnotation(group = subgroup[set_name(m)], show_annotation_name = TRUE)
bottom_annotation = HeatmapAnnotation(
  dist_to_tss = anno_boxplot(lapply(comb_sets, function(gr) gr$dist_to_tss), out = TRUE),
  mean_meth = sapply(comb_sets, function(gr) mean(gr$mean_meth)),
  annotation_name_side = "left"
)
)
)

```



Chapter 9

Interactive ComplexHeatmap

Please check the **InteractiveComplexHeatmap** package.

Chapter 10

Integrate with other packages

10.1 pheatmap

pheatmap is a great R package for making heatmaps, inspiring a lot of other heatmap packages such as **ComplexHeatmap**. From version 2.5.2 of **ComplexHeatmap**, I implemented a new `ComplexHeatmap::pheatmap()` function which actually maps all the parameters in `pheatmap::pheatmap()` to proper parameters in `ComplexHeatmap::Heatmap()`, which means, it converts a pheatmap to a complex heatmap. By doing this, the most significant improvement is now you can add multiple pheatmaps and annotations (defined by `ComplexHeatmap::rowAnnotation()`).

`ComplexHeatmap::pheatmap()` includes all arguments in `pheatmap::pheatmap()`, which means, you don't need to do any adaptation on your pheatmap code, you just rerun your pheatmap code and it will automatically and nicely convert to the complex heatmap.

Some arguments in `pheatmap::pheatmap()` are disabled and ignored in this translation, listed as follows:

- `kmeans_k`
- `filename`
- `width`
- `height`
- `silent`

The usage of remaining arguments is exactly the same as in `pheatmap::pheatmap()`.

In `pheatmap::pheatmap()`, the `color` argument is specified with a long color vector, e.g. :

```
pheatmap::pheatmap(mat,
  color = colorRampPalette(rev(brewer.pal(n = 7, name = "RdYlBu")))(100)
)
```

You can use the same setting of `color` in `ComplexHeatmap::pheatmap()`, but you can also simplify it as:

```
ComplexHeatmap::pheatmap(mat,
  color = rev(brewer.pal(n = 7, name = "RdYlBu"))
)
```

The colors for individual values are automatically interpolated.

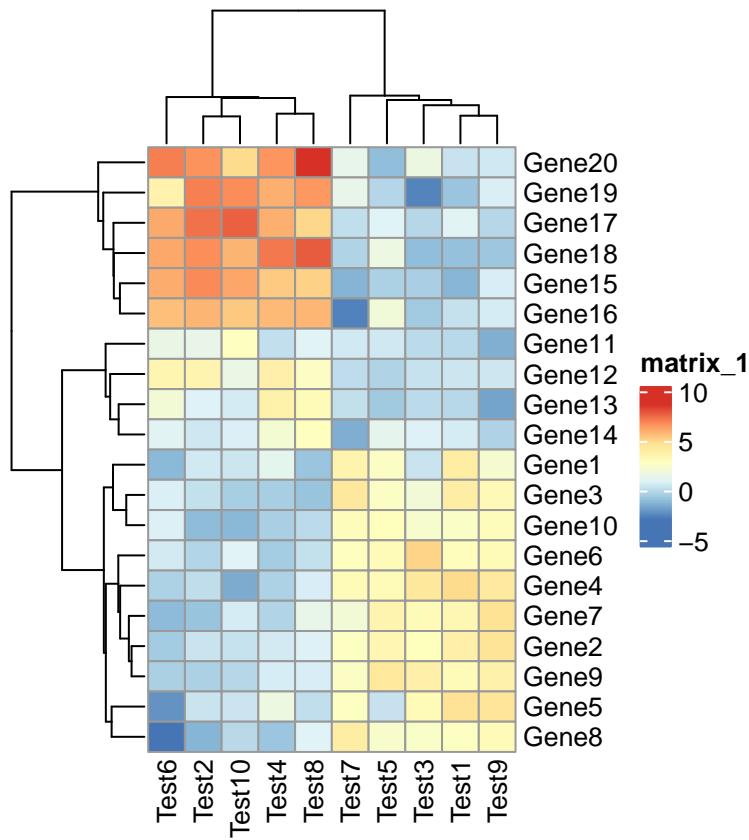
10.1.1 Examples

First we load an example dataset which is from the “Examples” section of the documentation of `pheatmap::pheatmap()` function .

```
library(ComplexHeatmap)
test = matrix(rnorm(200), 20, 10)
test[1:10, seq(1, 10, 2)] = test[1:10, seq(1, 10, 2)] + 3
test[11:20, seq(2, 10, 2)] = test[11:20, seq(2, 10, 2)] + 2
test[15:20, seq(2, 10, 2)] = test[15:20, seq(2, 10, 2)] + 4
colnames(test) = paste("Test", 1:10, sep = "")
rownames(test) = paste("Gene", 1:20, sep = "")
```

Calling `pheatmap()` (which is now actually `ComplexHeatmap::pheatmap()`) generates a similar heatmap as by `pheatmap::pheatmap()`.

```
pheatmap(test) # this is ComplexHeatmap::pheatmap
```



Everything looks the same except the style of the heatmap legend. There are also some other visual difference which you can find in the “Comparisons” section in this post.

The next one is an example for setting annotations (you should be familiar with how to set these data frames and color list if you are a pheatmap user).

```
annotation_col = data.frame(
  CellType = factor(rep(c("CT1", "CT2"), 5)),
  Time = 1:5
)
rownames(annotation_col) = paste("Test", 1:10, sep = "")

annotation_row = data.frame(
  GeneClass = factor(rep(c("Path1", "Path2", "Path3"), c(10, 4, 6)))
)
rownames(annotation_row) = paste("Gene", 1:20, sep = "")

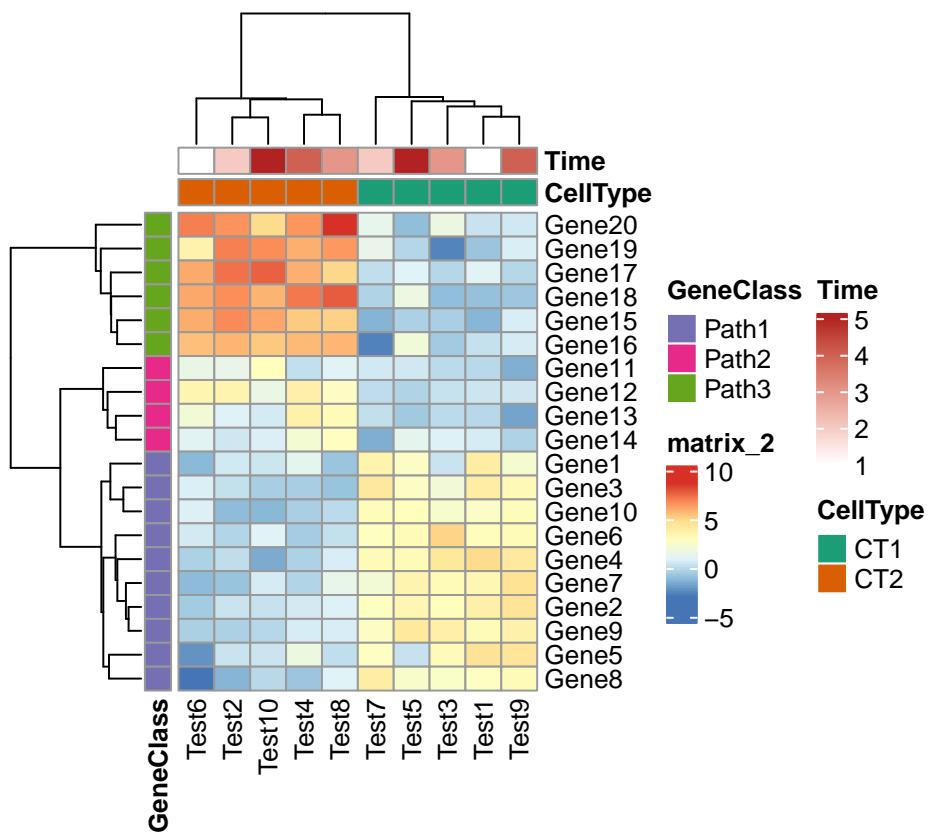
ann_colors = list(
  Time = c("white", "firebrick"),
  GeneClass = c("white", "black"),
  CellType = c("white", "black")
)
```

```

CellType = c(CT1 = "#1B9E77", CT2 = "#D95F02"),
GeneClass = c(Path1 = "#7570B3", Path2 = "#E7298A", Path3 = "#66A61E")
)

pheatmap(test, annotation_col = annotation_col, annotation_row = annotation_row,
annotation_colors = ann_colors)

```

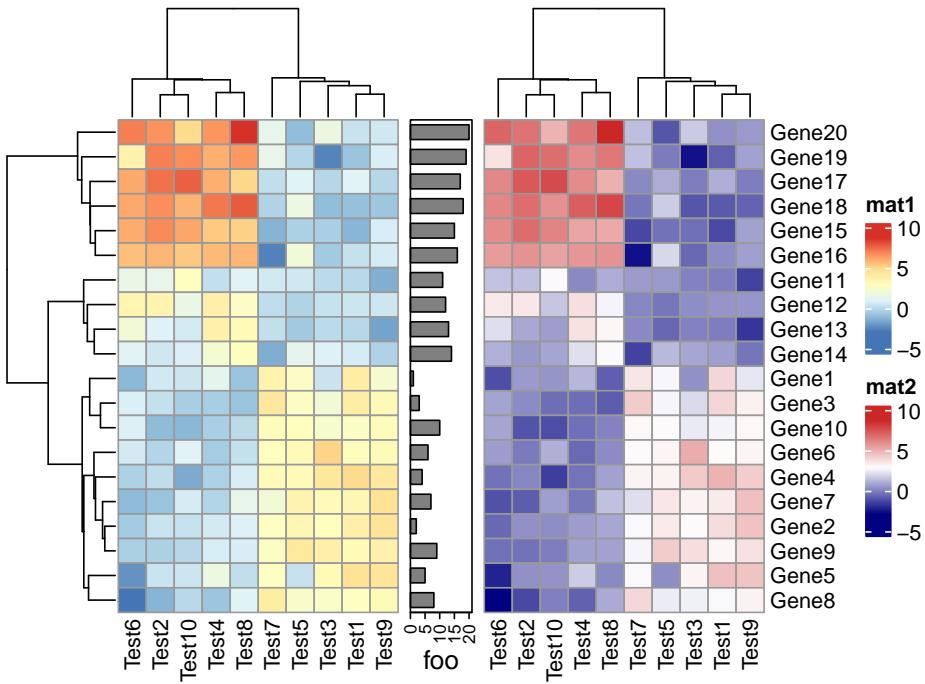


`ComplexHeatmap::pheatmap()` returns a `Heatmap` object, so it can be added with other heatmaps and annotations. Or in other words, you can add multiple `pheatmaps` and annotations. Cool!

```

p1 = pheatmap(test, name = "mat1")
p2 = rowAnnotation(foo = anno_barplot(1:nrow(test)))
p3 = pheatmap(test, name = "mat2",
  col = colorRampPalette(c("navy", "white", "firebrick3"))(50))
# or you can simply specify as
# p3 = pheatmap(test, name = "mat2", col = c("navy", "white", "firebrick3"))
p1 + p2 + p3

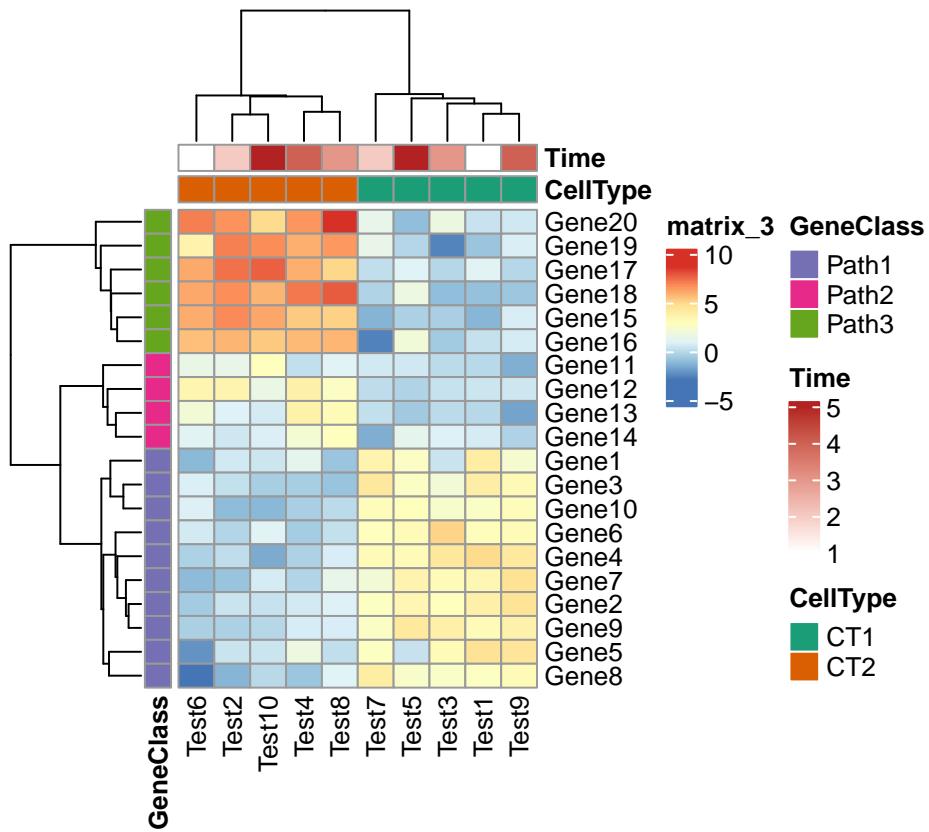
```



Nevertheless, if you really want to add multiple pheatmaps, I still suggest you to directly use the `Heatmap()` function. You can find how to migrate from `pheatmap::pheatmap()` to `ComplexHeatmap::Heatmap()` in the next section.

In previous examples, the legend for row annotation is grouped with heatmap legend. This can be modified by setting `legend_grouping` argument in `draw()` function:

```
p = pheatmap(test, annotation_col = annotation_col, annotation_row = annotation_row,
             annotation_colors = ann_colors)
draw(p, legend_grouping = "original")
```



One last thing is since `ComplexHeatmap::pheatmap()` returns a `Heatmap` object, if `pheatmap()` is not called in an interactive environment, e.g. in an R script, inside a function or in a `for` loop, you need to explicitly use `draw()` function:

```
for(...) {
  p = pheatmap(...)
  draw(p)
}
```

10.1.2 Translation

Following table lists how to map parameters in `pheatmap::pheatmap()` to `ComplexHeatmap::Heatmap()`.

Arguments in <code>pheatmap::pheatmap()</code>	Identical settings/arguments in <code>ComplexHeatmap::Heatmap()</code>
<code>mat</code>	<code>matrix</code>

Arguments in <code>pheatmap::pheatmap()</code>	Identical settings/arguments in <code>ComplexHeatmap::Heatmap()</code>
<code>color</code>	Users can specify a color mapping function by <code>circlize::colorRamp2()</code> , or provide a vector of colors on which colors for individual values are linearly interpolated.
<code>kmeans_k</code>	No corresponding parameter because it changes the matrix for heatmap.
<code>breaks</code>	It should be specified in the color mapping function.
<code>border_color</code>	<code>rect_gp = gpar(col = border_color).</code> In the annotations, it is <code>HeatmapAnnotation(..., gp = gpar(col = border_color)).</code>
<code>cellwidth</code>	<code>width = ncol(mat)*unit(cellwidth, "pt")</code>
<code>cellheight</code>	<code>height = nrow(mat)*unit(cellheight, "pt")</code>
<code>scale</code>	Users should simply apply <code>scale()</code> on the matrix before sending to <code>Heatmap()</code> .
<code>cluster_rows</code>	<code>cluster_rows</code>
<code>cluster_cols</code>	<code>cluster_columns</code>
<code>clustering_distance_rows</code>	<code>clustering_distance_rows</code> . The value <code>correlation</code> should be changed to <code>pearson</code> .
<code>clustering_distance_cols</code>	<code>clustering_distance_columns</code> , The value <code>correlation</code> should be changed to <code>pearson</code> .
<code>clustering_method</code>	<code>clustering_method_rows/clustering_method_columns</code>
<code>clustering_callback</code>	The processing on the dendrogram should be applied before sending to <code>Heatmap()</code> .
<code>cutree_rows</code>	<code>row_split</code> and row clustering should be applied.
<code>cutree_cols</code>	<code>column_split</code> and column clustering should be applied.
<code>treeheight_row</code>	<code>row_dend_width = unit(treeheight_row, "pt")</code>
<code>treeheight_col</code>	<code>column_dend_height = unit(treeheight_col, "pt")</code>
<code>legend</code>	<code>show_heatmap_legend</code>
<code>legend_breaks</code>	<code>heatmap_legend_param = list(at = legend_breaks)</code>
<code>legend_labels</code>	<code>heatmap_legend_param = list(labels = legend_labels)</code>
<code>annotation_row</code>	<code>left_annotation = rowAnnotation(df = annotation_row)</code>

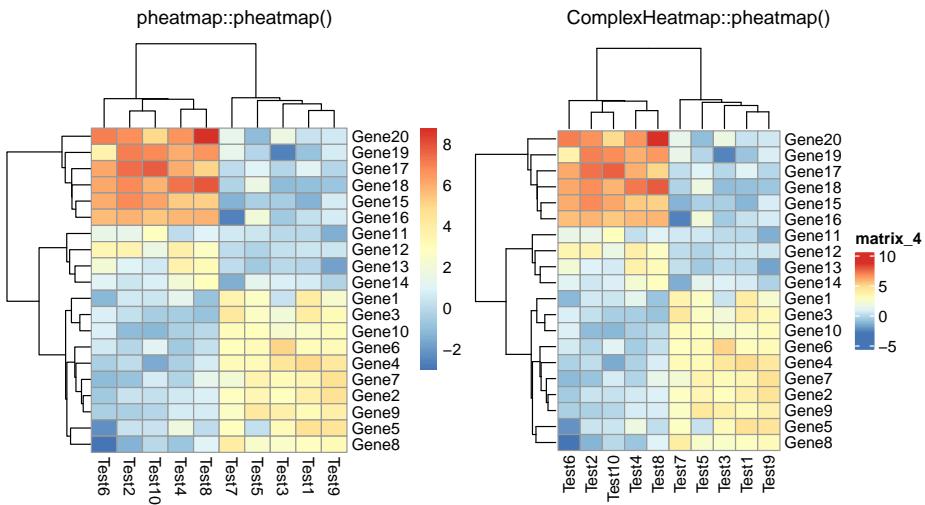
Arguments in <code>pheatmap::pheatmap()</code>	Identical settings/arguments in <code>ComplexHeatmap::Heatmap()</code>
<code>annotation_col</code>	<code>top_annotation = HeatmapAnnotation(df = annotation_col)</code>
<code>annotation</code>	Not supported.
<code>annotation_colors</code>	<code>col</code> argument in <code>HeatmapAnnotation()/rowAnnotation()</code> .
<code>annotation_legend</code>	<code>show_legend</code> argument in <code>HeatmapAnnotation()/rowAnnotation()</code> .
<code>annotation_names_row</code>	<code>show_annotation_name</code> in <code>rowAnnotation()</code> .
<code>annotation_names_col</code>	<code>show_annotation_name</code> in <code>HeatmaoAnnotation()</code> .
<code>drop_levels</code>	Unused levels are all dropped.
<code>show_rownames</code>	<code>show_row_names</code>
<code>show_colnames</code>	<code>show_column_names</code>
<code>main</code>	<code>column_title</code>
<code>fontsize</code>	<code>gpar(fontsize = fontsize)</code> in corresponding heatmap components.
<code>fontsize_row</code>	<code>row_names_gp = gpar(fontsize = fontsize_row)</code>
<code>fontsize_col</code>	<code>column_names_gp = gpar(fontsize = fontsize_col)</code>
<code>angle_col</code>	<code>column_names_rot</code> . The rotation of row annotation names are not supported.
<code>display_numbers</code>	Users should set a proper <code>cell_fun</code> or <code>layer_fun</code> (vectorized and faster version of <code>cell_fun</code>). E.g. if <code>display_numbers</code> is <code>TRUE</code> , <code>layer_fun</code> can be set as <code>function(j, i, x, y, w, h, fill) { grid.text(sprintf(number_format, pindex(mat, i, j)), x = x, y = y, gp = gpar(col = number_color, fontsize = fontsize_number)) }</code> . If <code>display_numbers</code> is a matrix, replace <code>mat</code> to <code>display_numbers</code> in the <code>layer_fun</code> . See above.
<code>number_format</code>	See above.
<code>number_color</code>	See above.
<code>fontsize_number</code>	See above.
<code>gaps_row</code>	Users should construct a “splitting variable” and send to <code>row_split</code> . E.g. <code>slices = diff(c(0, gaps_row, nrow(mat))); rep(seq_along(slices), times = slices)</code> .

Arguments in <code>pheatmap::pheatmap()</code>	Identical settings/arguments in <code>ComplexHeatmap::Heatmap()</code>
<code>gaps_col</code>	Users should construct a “splitting variable” and send to <code>column_split</code> .
<code>labels_row</code>	<code>row_labels</code>
<code>labels_col</code>	<code>column_labels</code>
<code>filename</code>	No corresponding setting in <code>Heatmap()</code> . Users need to explicitly use e.g. <code>pdf()</code> .
<code>width</code>	No corresponding setting in <code>Heatmap()</code> .
<code>height</code>	No corresponding setting in <code>Heatmap()</code> .
<code>silent</code>	No corresponding setting in <code>Heatmap()</code> .
<code>na_col</code>	<code>na_col</code>

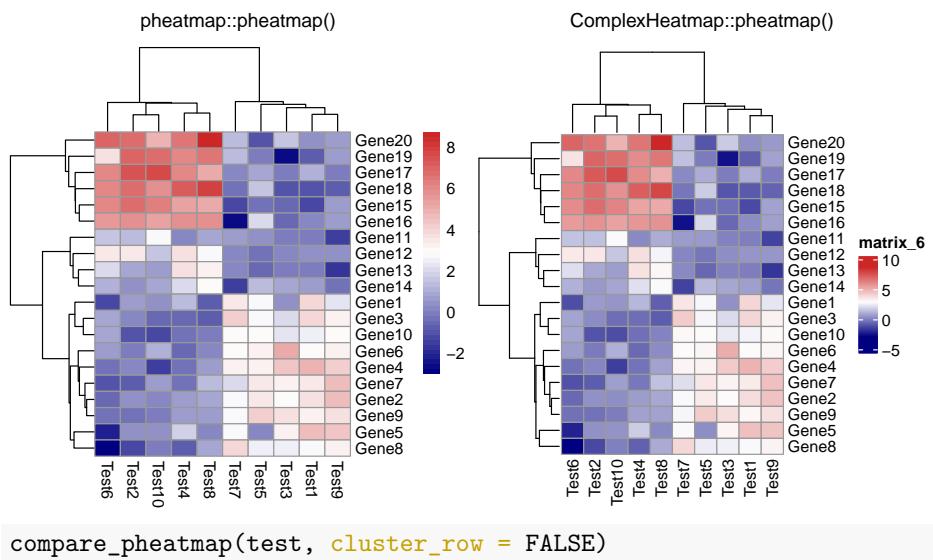
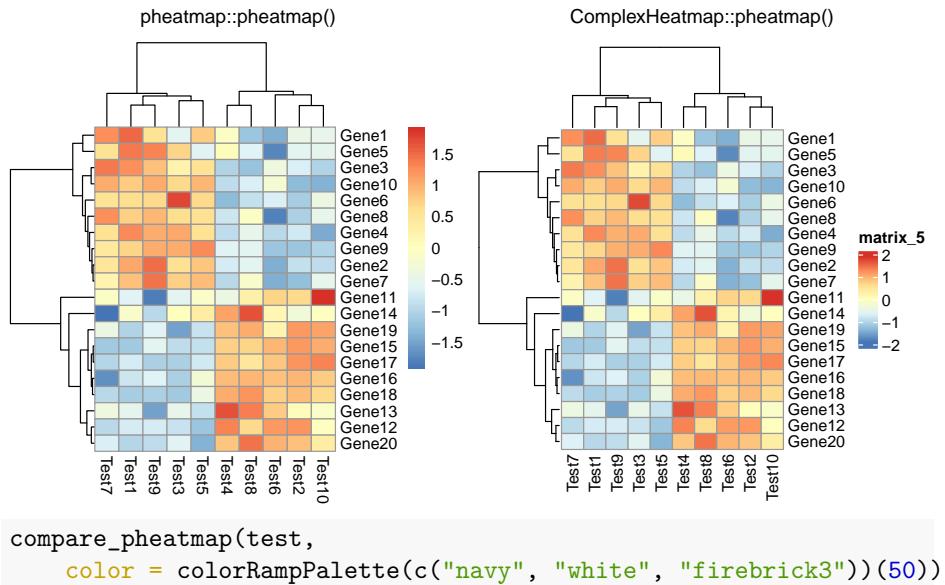
10.1.3 Comparisons

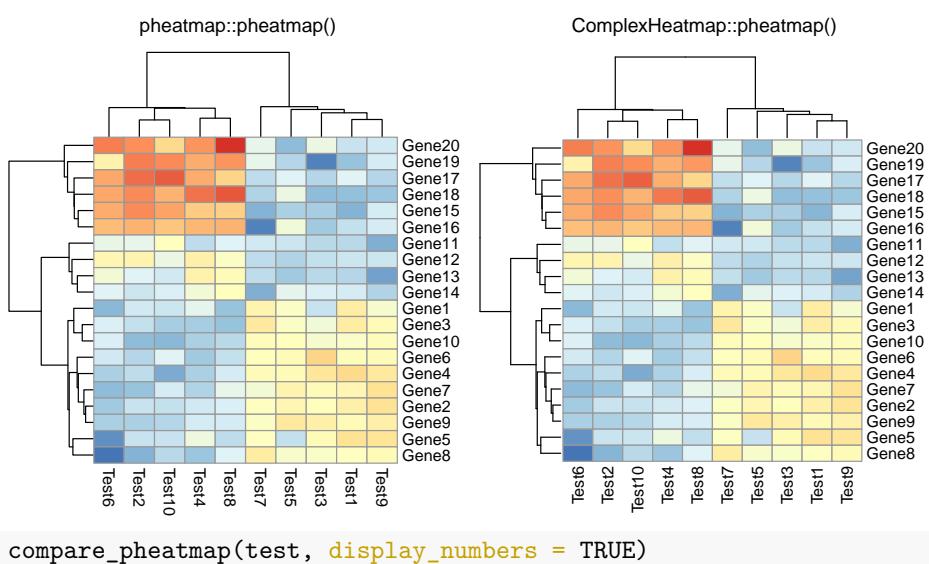
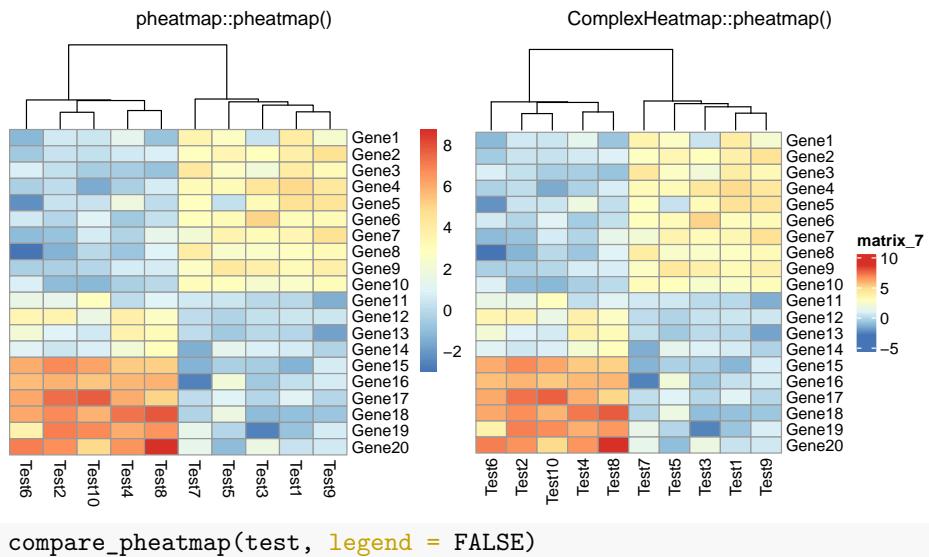
I ran all the example code in the “Examples” section of the documentation of `pheatmap::pheatmap()` function . I also implemented a wrapper function `ComplexHeatmap::compare_pheatmap()` which basically uses the same set of arguments for `pheatmap::pheatmap()` and `ComplexHeatmap::pheatmap()` and draws two heatmaps, so that you can directly see the similarity and difference of the two heatmap implementations.

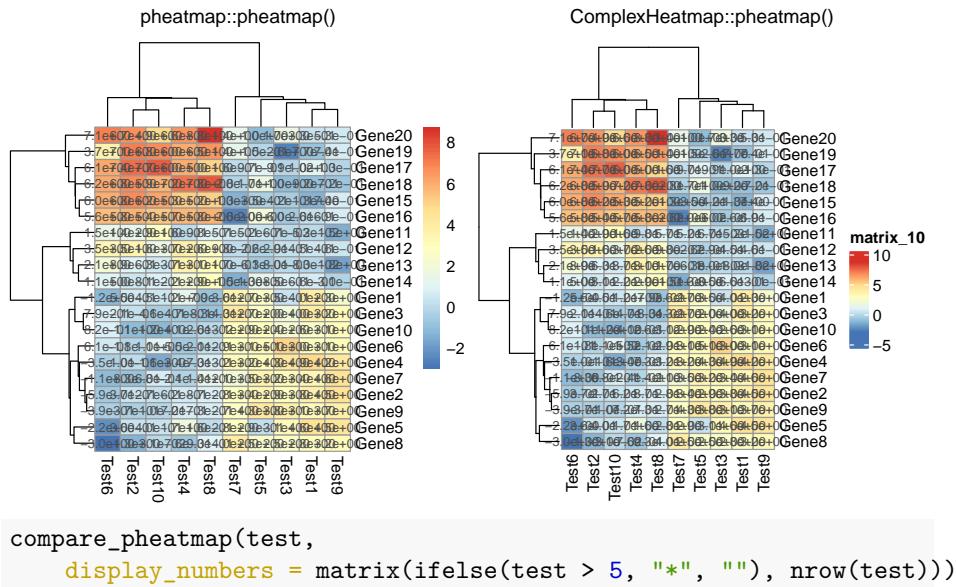
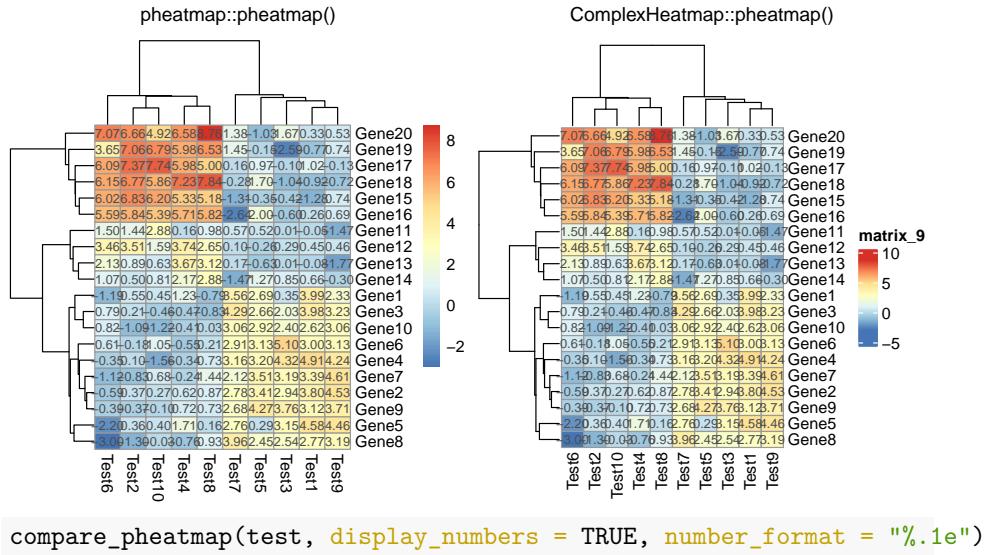
```
compare_pheatmap(test)
```

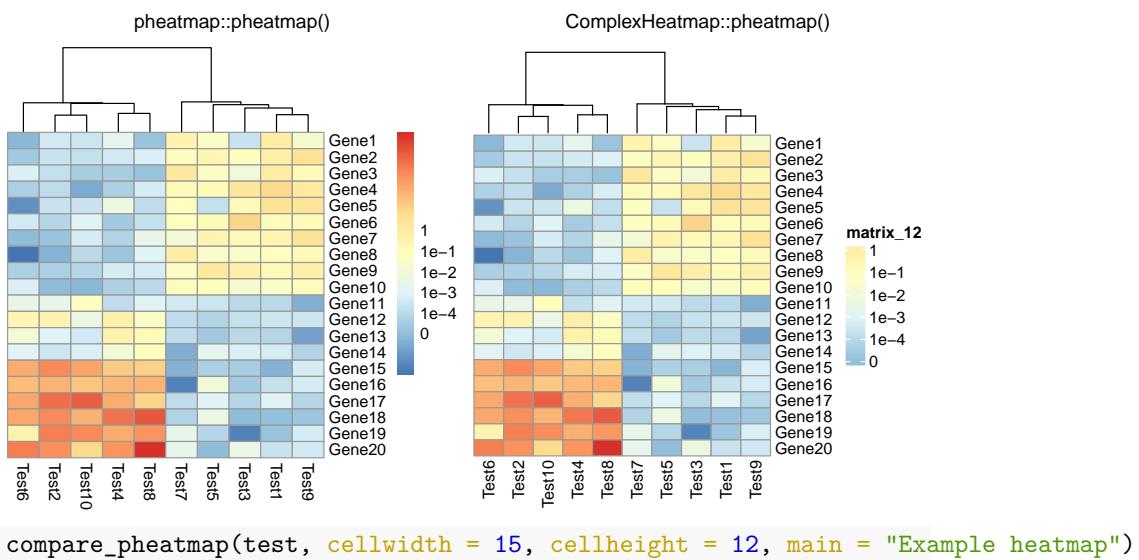
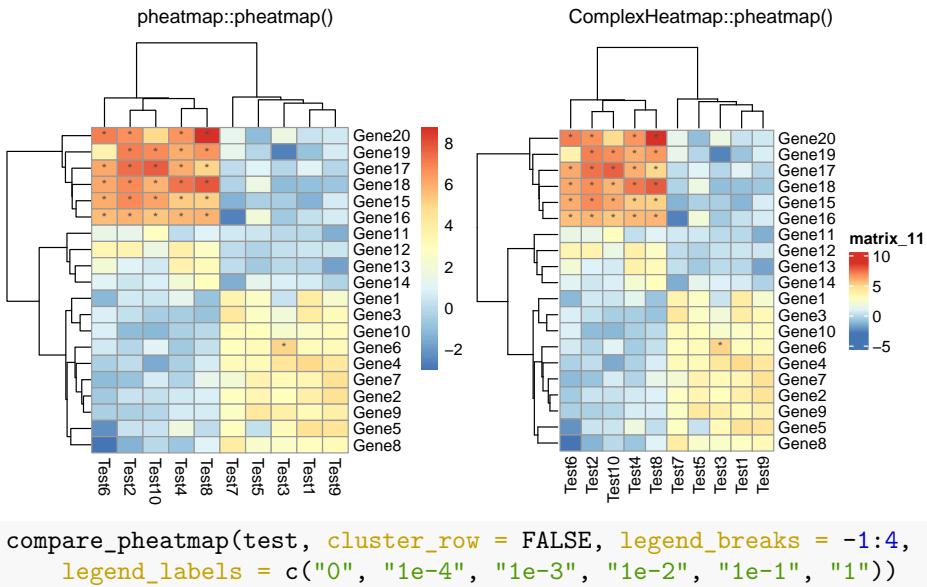


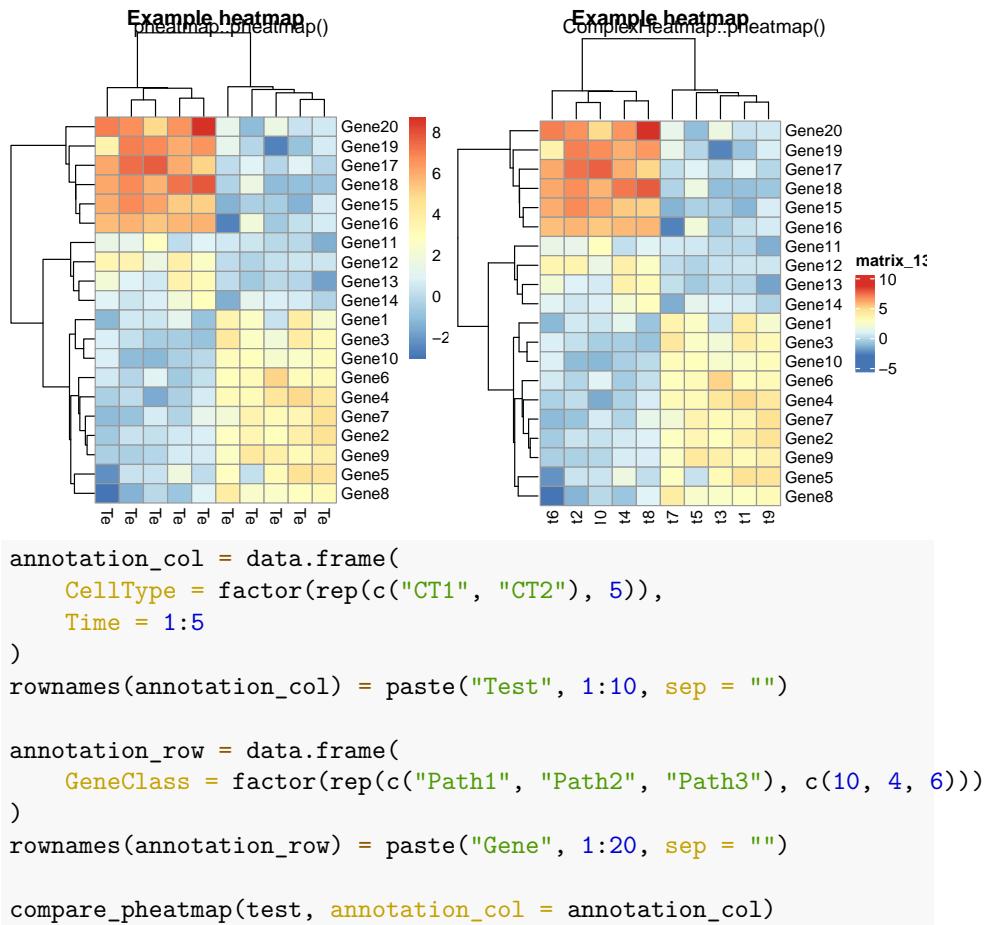
```
compare_pheatmap(test, scale = "row", clustering_distance_rows = "correlation")
```

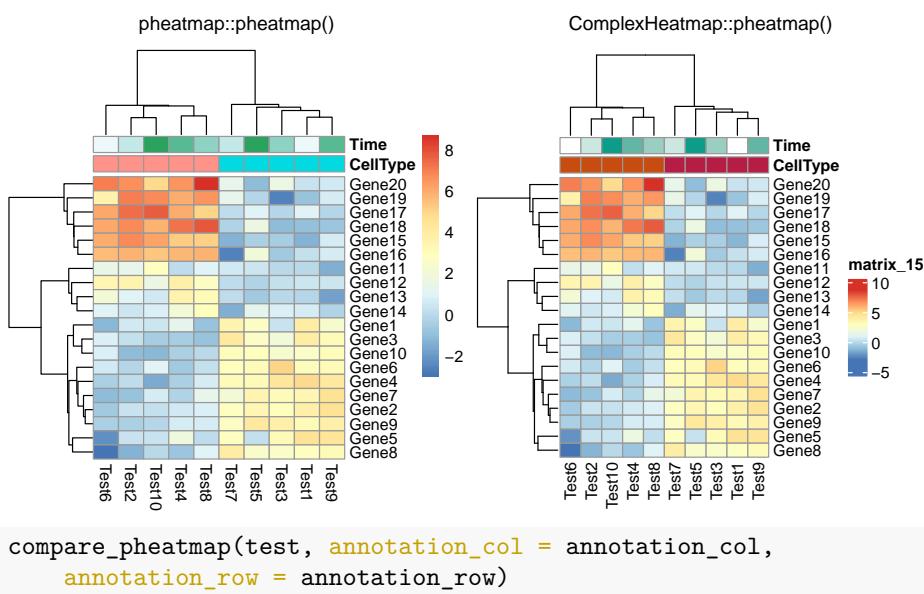
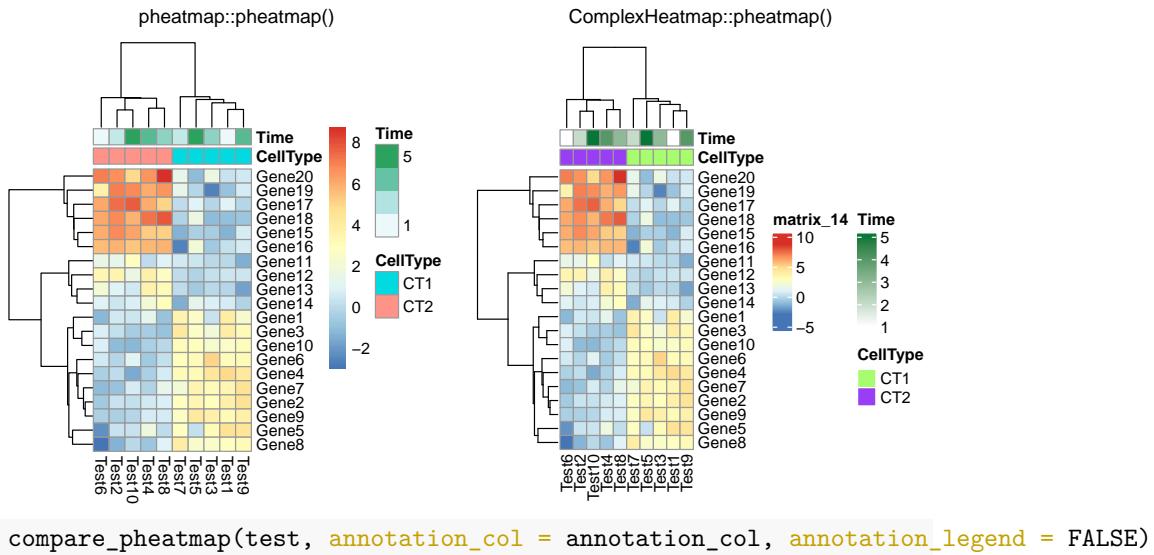


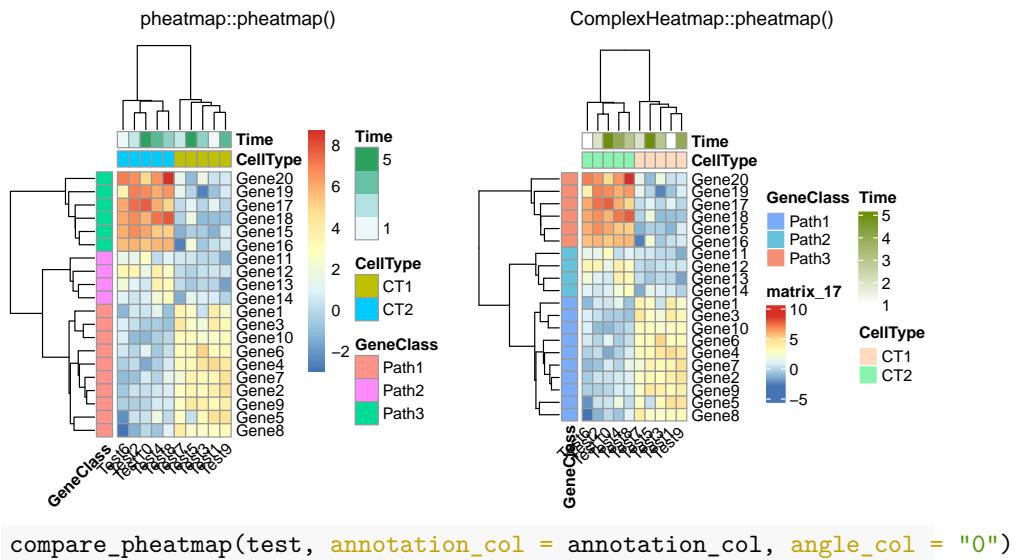
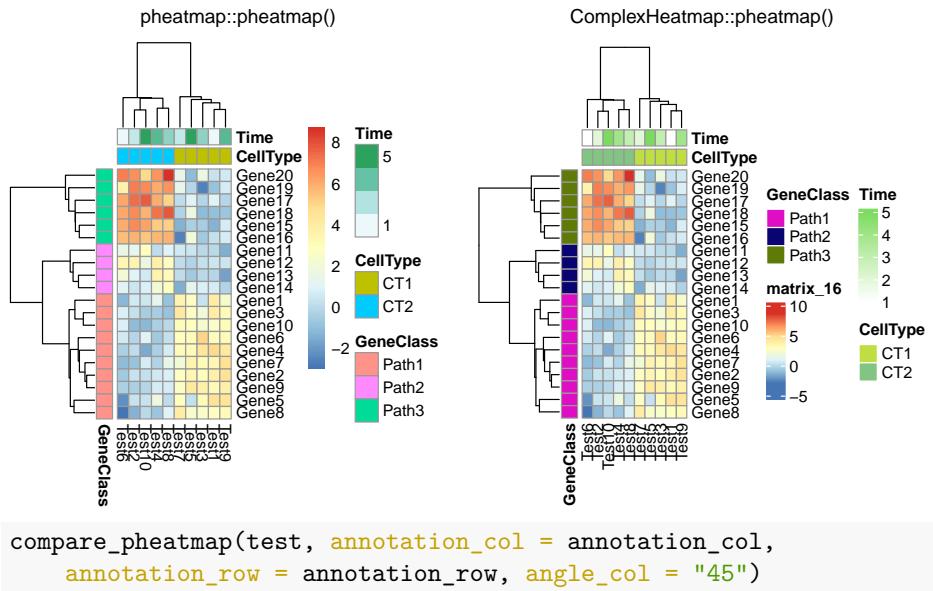


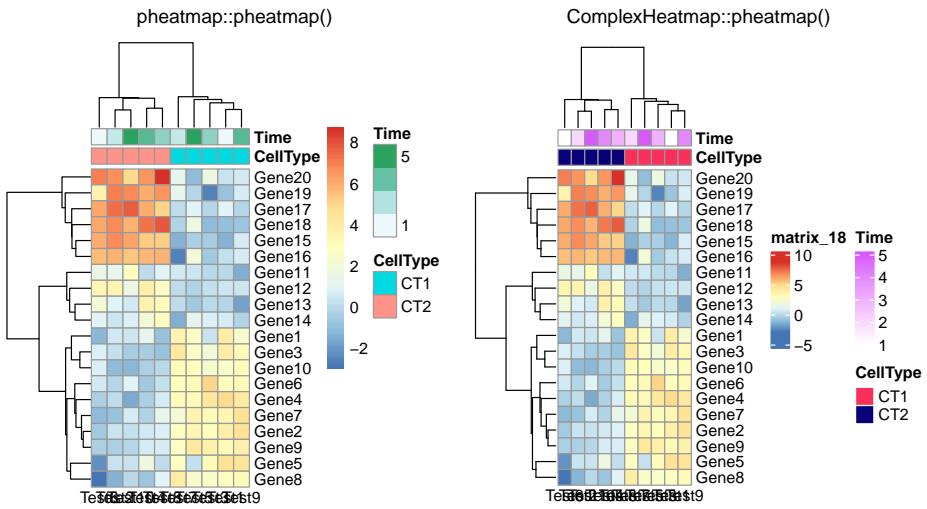










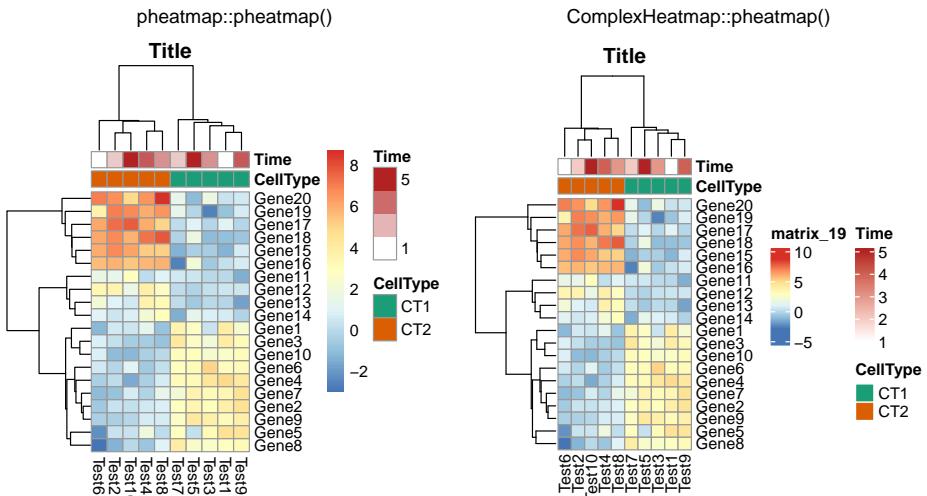


```

ann_colors = list(
  Time = c("white", "firebrick"),
  CellType = c(CT1 = "#1B9E77", CT2 = "#D95F02"),
  GeneClass = c(Path1 = "#7570B3", Path2 = "#E7298A", Path3 = "#66A61E")
)

compare_pheatmap(test, annotation_col = annotation_col,
  annotation_colors = ann_colors, main = "Title")

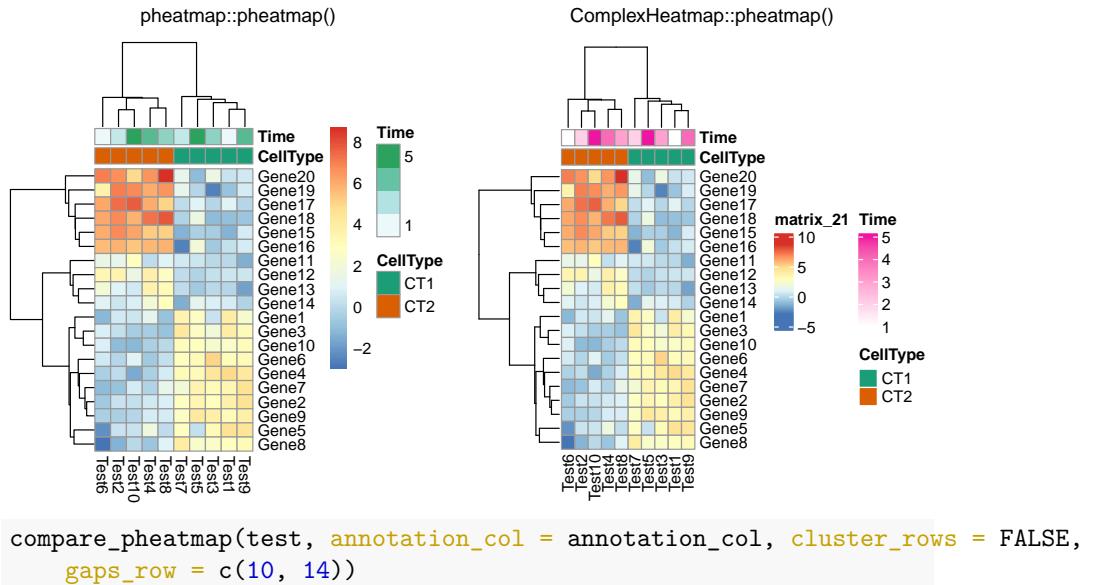
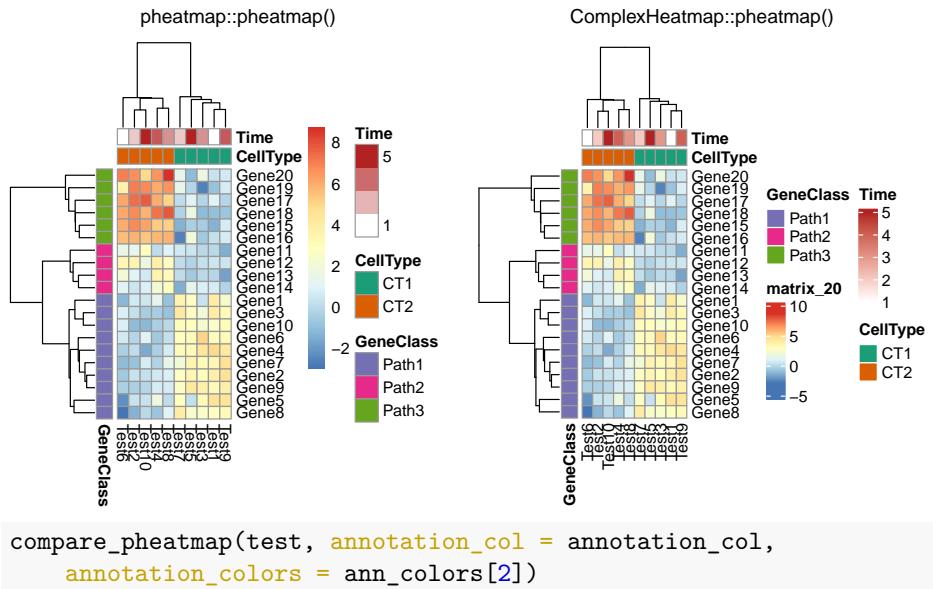
```

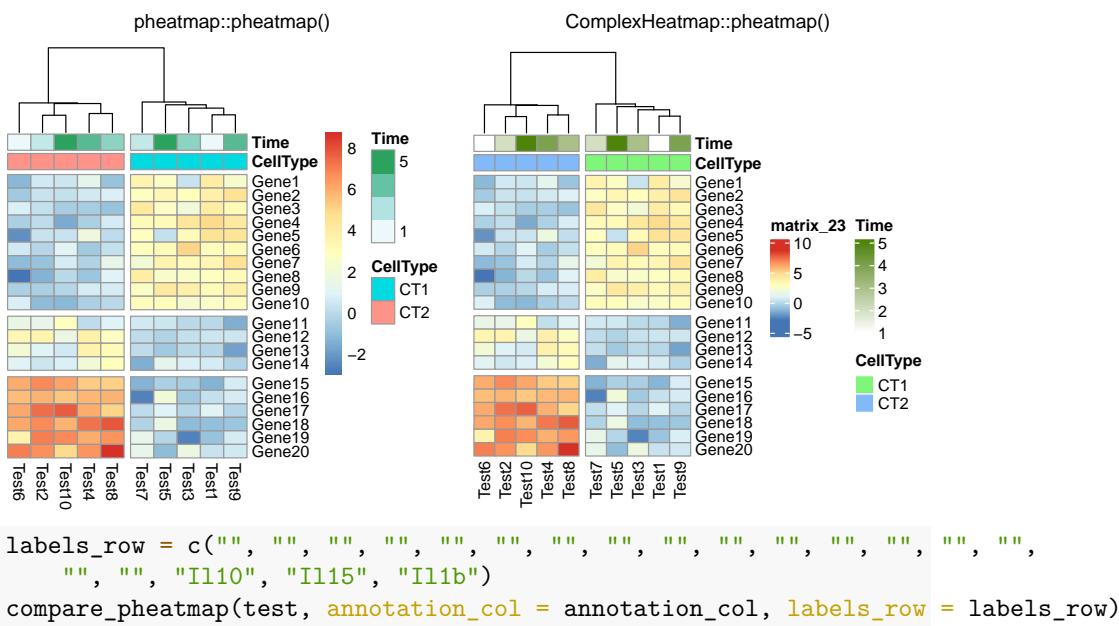
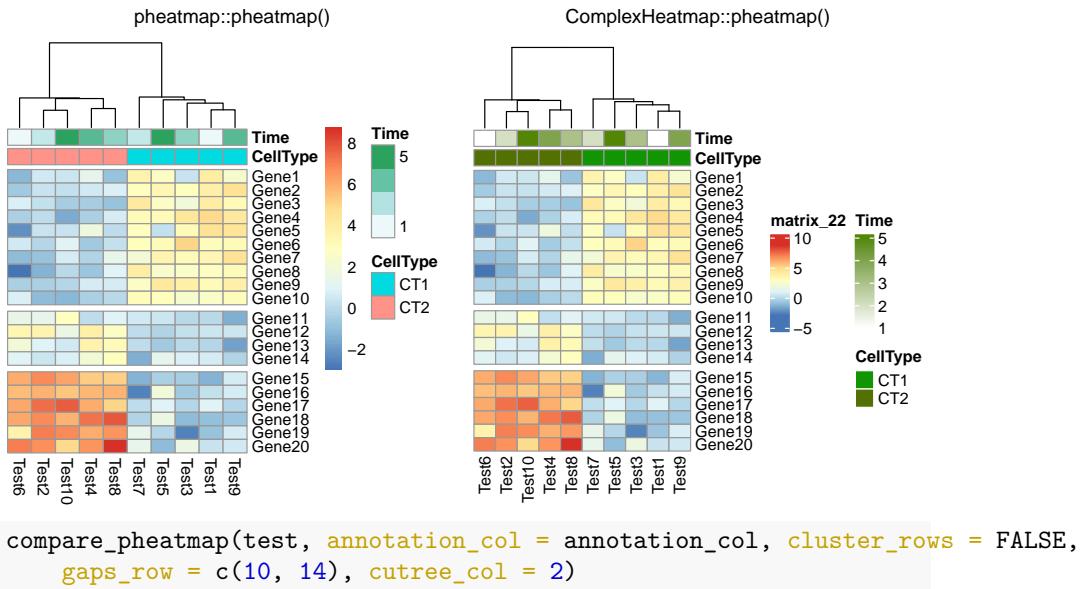


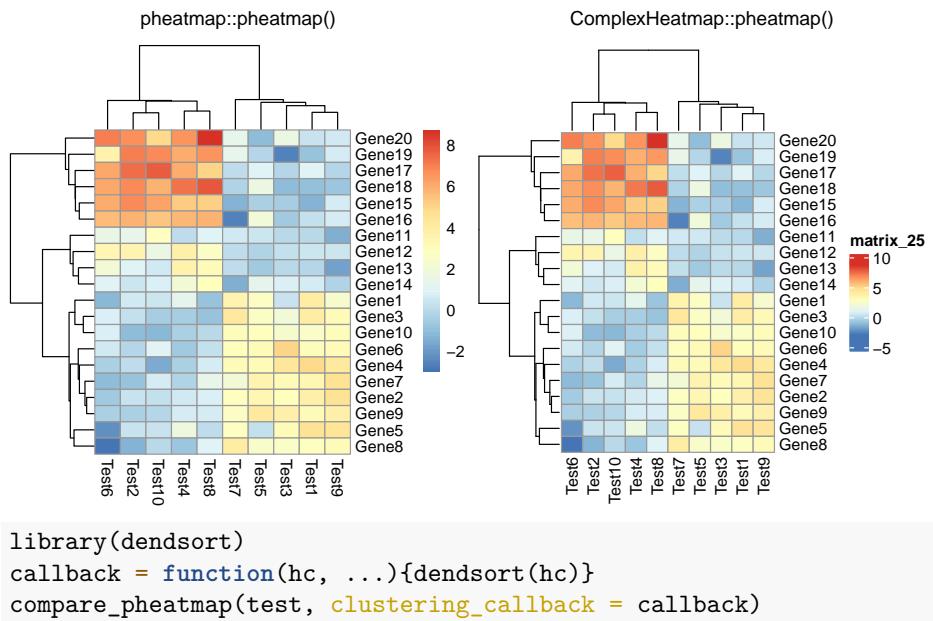
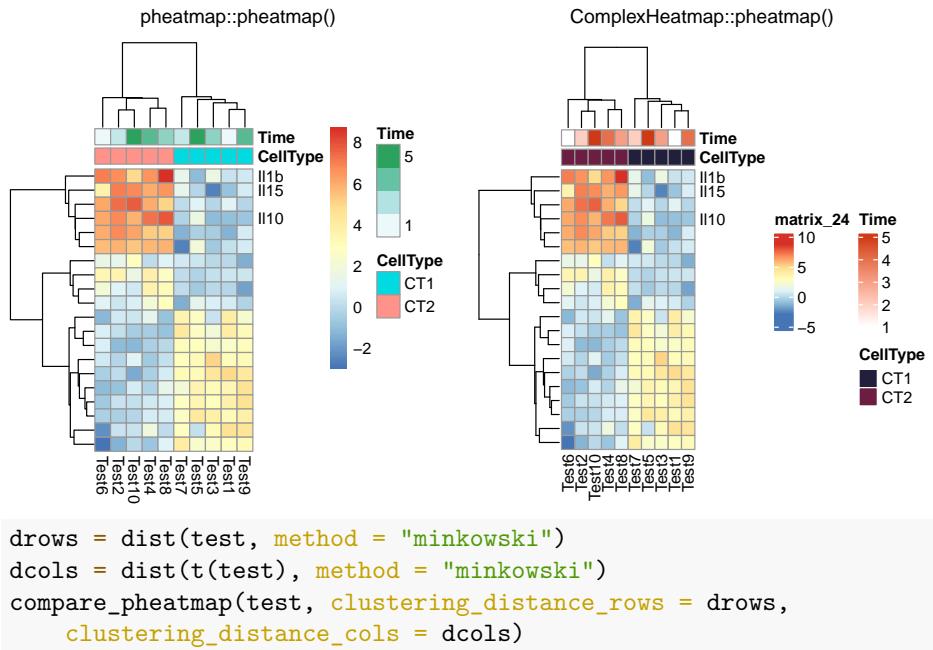
```

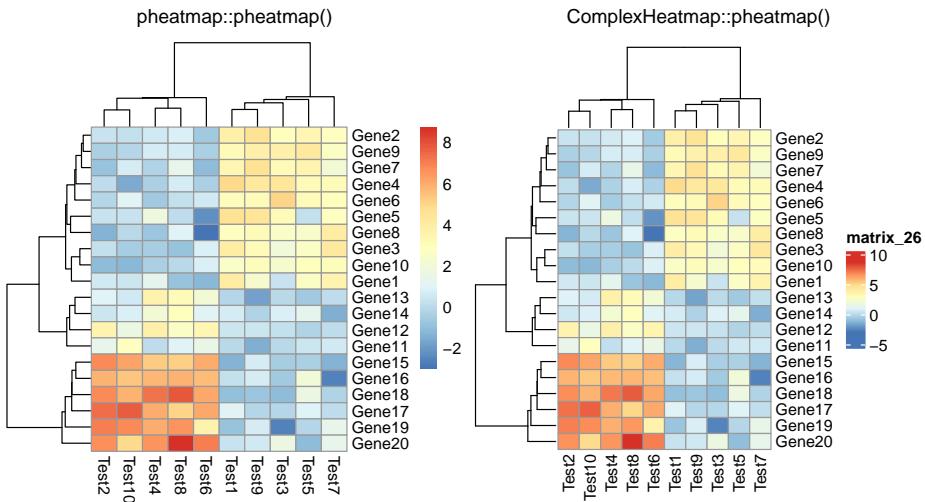
compare_pheatmap(test, annotation_col = annotation_col,
  annotation_row = annotation_row, annotation_colors = ann_colors)

```









10.2 cowplot

The **cowplot** package is used to combine multiple plots into a single figure. In most cases, **ComplexHeatmap** works perfectly with **cowplot**, but there are some cases that need special attention.

Also there are some other packages that combine multiple plots, such as **multi-panelfigure**, but I think the mechanism behind is the same.

Following functionalities in **ComplexHeatmap** cause problems with using **cowplot**.

1. `anno_zoom()`/`anno_link()`: The adjusted positions by these two functions rely on the size of the graphics device.
2. `anno_mark()`: The same reason as `anno_zoom()`. The adjusted positions also rely on the device size.
3. When there are too many legends, the legends will be wrapped into multiple columns. The calculation of the legend positions rely on the device size.

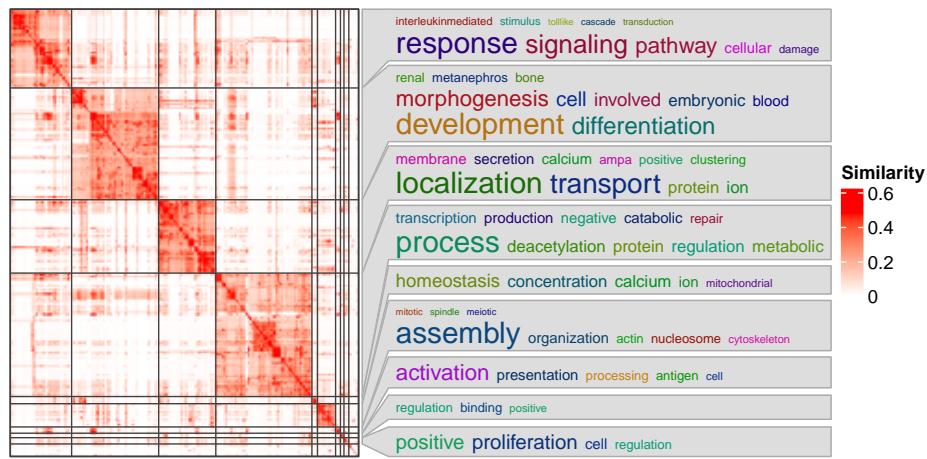
In following I demonstrate a case with using the `anno_zoom()`. Here the example is from the **simplifyEnrichment** package and the plot shows a GO similarity heatmap with word cloud annotation showing the major biological functions in each group.

You don't need to really understand the following code. The `ht_clusters()` function basically draws a heatmap with `Heatmap()` and add the word cloud annotation by `anno_zoom()`.

```
library(simplifyEnrichment)
set.seed(1234)
go_id = random_GO(500)
```

```
mat = GO_similarity(go_id)
cl = binary_cut(mat)
ht_clusters(mat, cl)
```

```
## Perform keywords enrichment for 9 GO lists...
```



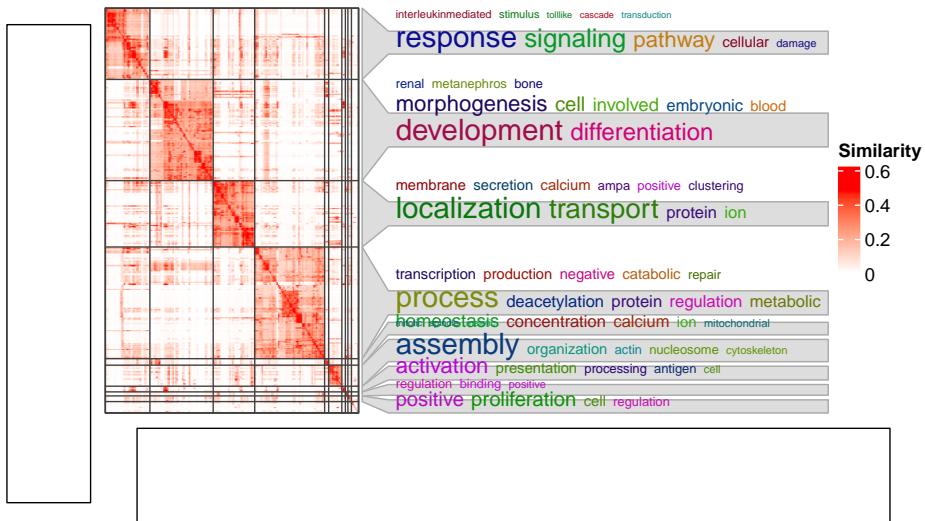
Next we put this heatmap as a sub-figure with **cowplot**. To integrate with **cowplot**, the heatmap should be captured by `grid::grid.grabExpr()` as a complex `grob` object. Note here you need to use `draw()` function to draw the heatmap explicitly.

```
library(cowplot)
library(grid)
p1 = rectGrob(width = 0.9, height = 0.9)
p2 = grid.grabExpr(ht_clusters(mat, cl))
```

```
## Perform keywords enrichment for 9 GO lists...
```

```
p3 = rectGrob(width = 0.9, height = 0.9)

plot_grid(p1,
  plot_grid(p2, p3, nrow = 2, rel_heights = c(4, 1)),
  nrow = 1, rel_widths = c(1, 9)
)
```



Woooo! The word cloud annotation is badly aligned.

There are some details that should be noted for `grid.grabExpr()` function. It actually opens an invisible graphics device (by `pdf(NULL)`) with a default size 7x7 inches. Thus, for this line:

```
p2 = grid.grabExpr(ht_clusters(mat, cl))
```

The word cloud annotation in `p2` is actually calculated in a region of 7x7 inches, and when it is written back to the figure by `plot_grid()`, the space for `p2` changes, that is why the word cloud annotation is wrongly aligned.

On the other hand, if “a simple heatmap” is captured by `grid.grabExpr()`, e.g.:

```
p2 = grid.grabExpr(draw(Heatmap(mat)))
```

when `p2` is put back, everything will work fine because now all the heatmap elements are not dependent on the device size and the positions will be automatically adjusted to the new space.

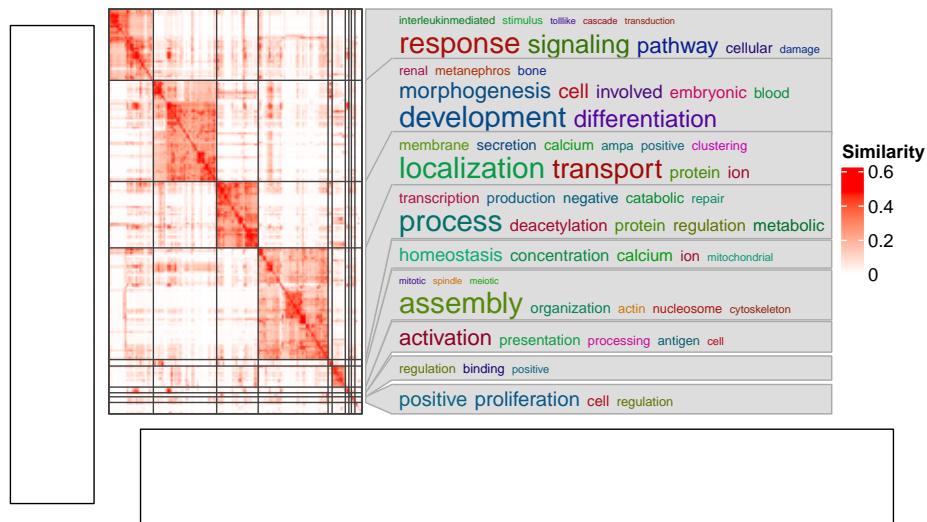
This effect can also be observed by plotting the heatmap in the interactive graphics device and resizing the window by dragging it.

The solution is rather simple. Since the reason for this inconsistency is the different space between where it is captured and where it is drawn, we only need to capture the heatmap under the device with the same size as where it is going to be put.

As in the layout which we set in the `plot_grid()` function, the heatmap occupies 9/10 width and 4/5 height of the figure. So, the width and height of the space for the heatmap is calculated as follows and assigned to the `width` and `height` arguments in `grid.grabExpr()`.

```
w = convertWidth(unit(1, "npc")*(9/10), "inch", valueOnly = TRUE)
h = convertHeight(unit(1, "npc")*(4/5), "inch", valueOnly = TRUE)
p2 = grid.grabExpr(ht_clusters(mat, cl), width = w, height = h)

## Perform keywords enrichment for 9 GO lists...
plot_grid(p1,
  plot_grid(p2, p3, nrow = 2, rel_heights = c(4, 1)),
  nrow = 1, rel_widths = c(1, 9)
)
```



Now everything is back to normal!

10.3 gridtext

The **gridtext** package provides a nice and easy way for rendering text under the **grid** system. From version 2.3.3 of **ComplexHeatmap**, text-related elements can be rendered by **gridtext**.

For all text-related elements, the text needs to be wrapped by **gt_render()** function, which marks the text and adds related parameters that are going to be processed by **gridtext**.

Currently **ComplexHeatmap** supports **gridtext::richtext_grob()**, so some of the parameters for **richtext_grob()** can be passed via **gt_render()**.

```
gt_render("foo", r = unit(2, "pt"), padding = unit(c(2, 2, 2, 2), "pt"))

## [1] "foo"
```

```
## attr(,"class")
## [1] "gridtext"
## attr(,"param")
## attr(,"param")$r
## [1] 2points
##
## attr(,"param")$padding
## [1] 2points 2points 2points 2points
```

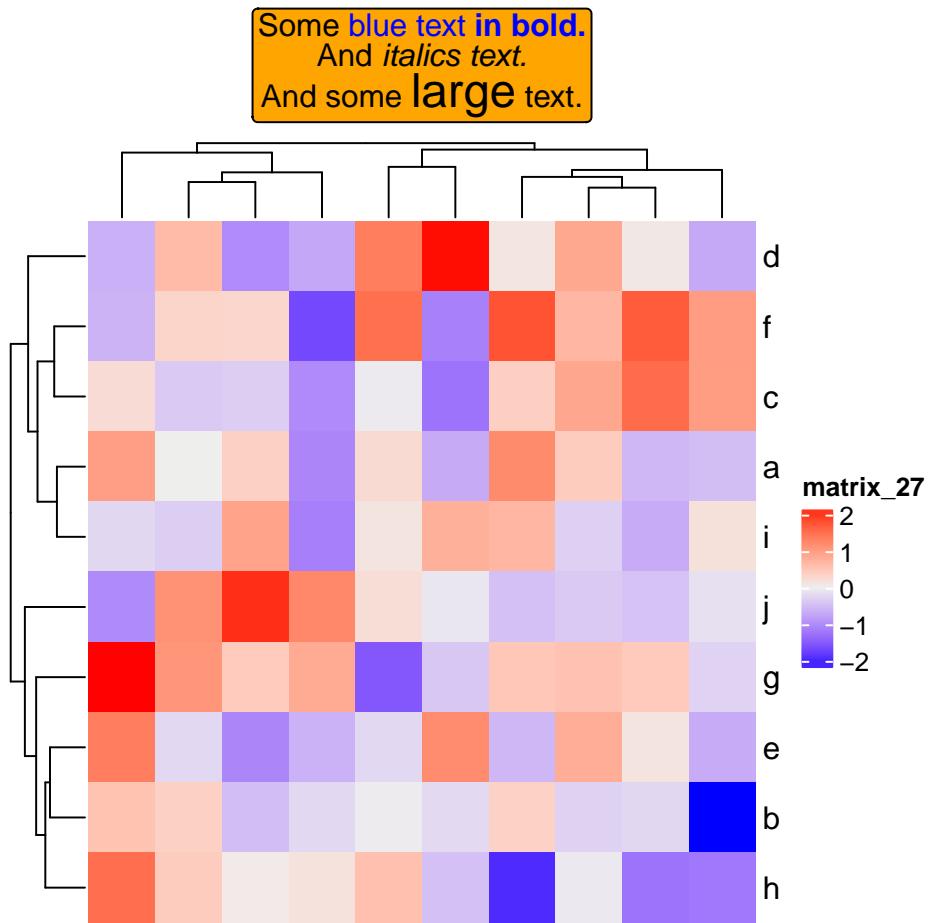
For each heatmap element, e.g. column title, graphic parameters can be set by the companion argument, e.g. `column_title_gp`. To make it simpler, all graphic parameters set by `box_gp` are merged with `*_gp` by adding `box_` prefix, e.g.:

```
..., column_title = gt_render("foo"), column_title_gp = gpar(col = "red", box_fill = "blue"), ...
```

Graphic parameters can also be specified inside `gt_render()`. Following is the same as the one above:

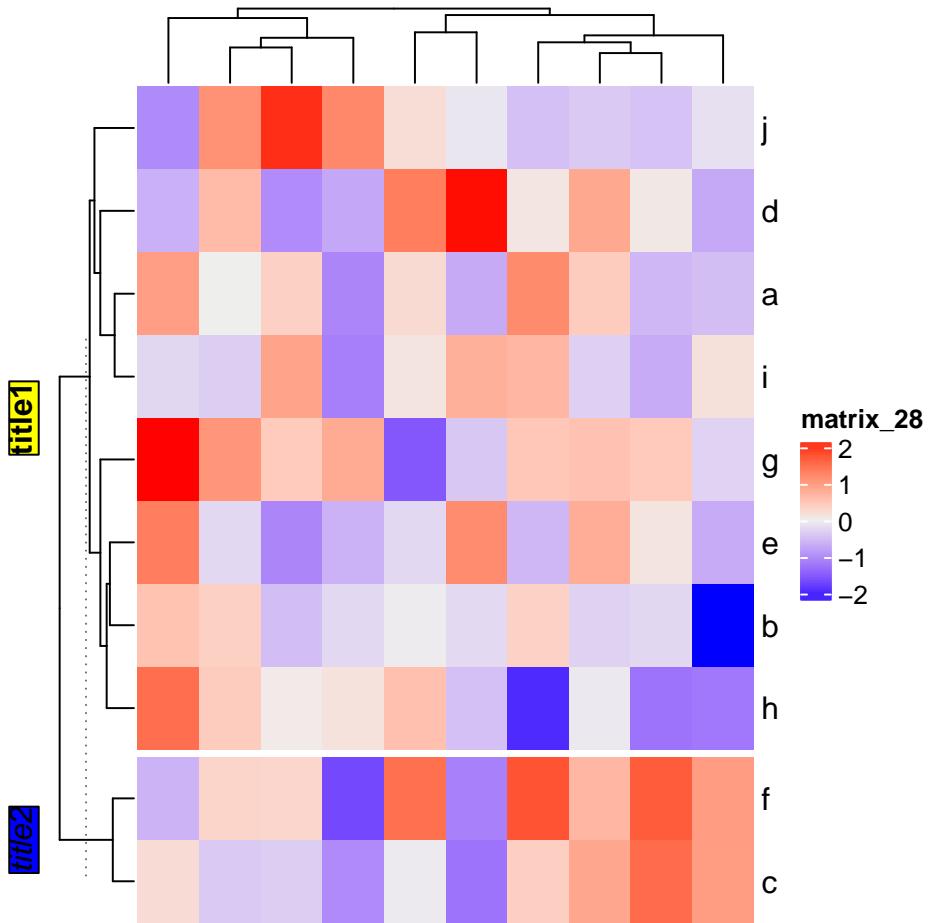
```
..., column_title = gt_render("foo", gp = gpar(col = "red", box_fill = "blue")), ...
```

```
set.seed(123)
mat = matrix(rnorm(100), 10)
rownames(mat) = letters[1:10]
Heatmap(mat,
        column_title = gt_render("Some <span style='color:blue'>blue text **in bold.**</span><br>And
                                  r = unit(2, "pt"),
                                  padding = unit(c(2, 2, 2, 2), "pt")),
        column_title_gp = gpar(box_fill = "orange"))
```



If heatmap is split:

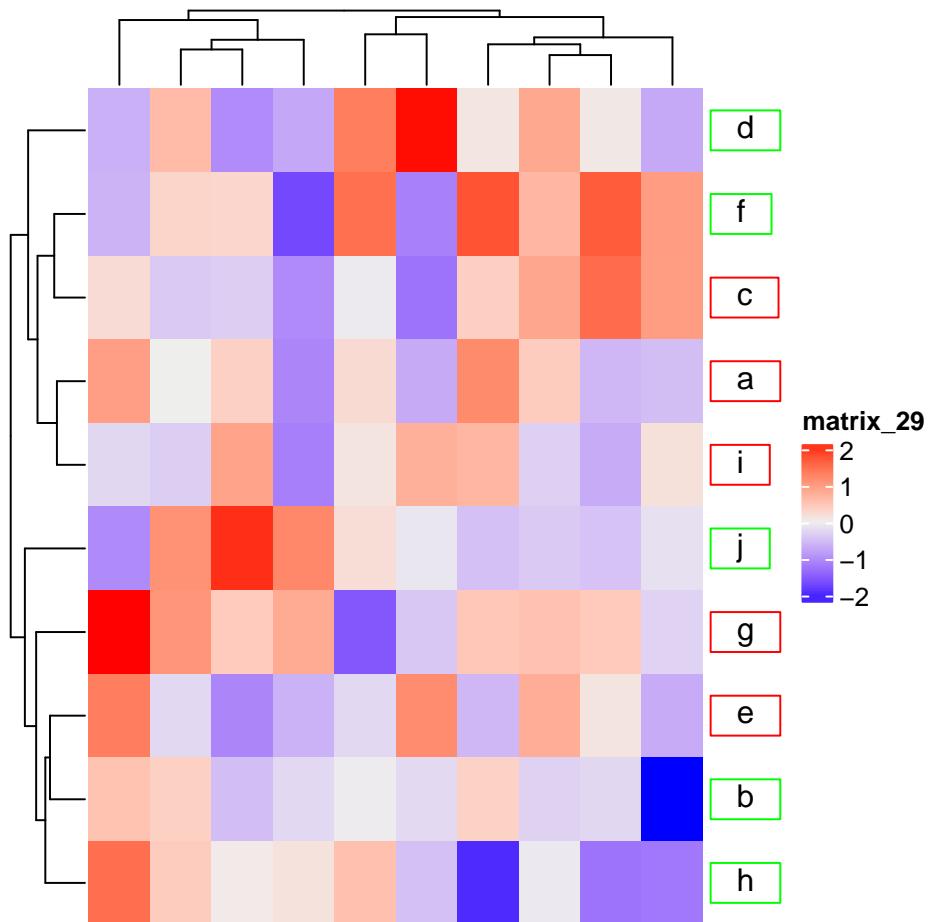
```
Heatmap(mat,
  row_km = 2,
  row_title = gt_render(c("**title1**", "_title2_")),
  row_title_gp = gpar(box_fill = c("yellow", "blue")))
```



10.3.2 Row/column names

Rendered row/column names should be explicitly specified by `row_labels/column_labels`

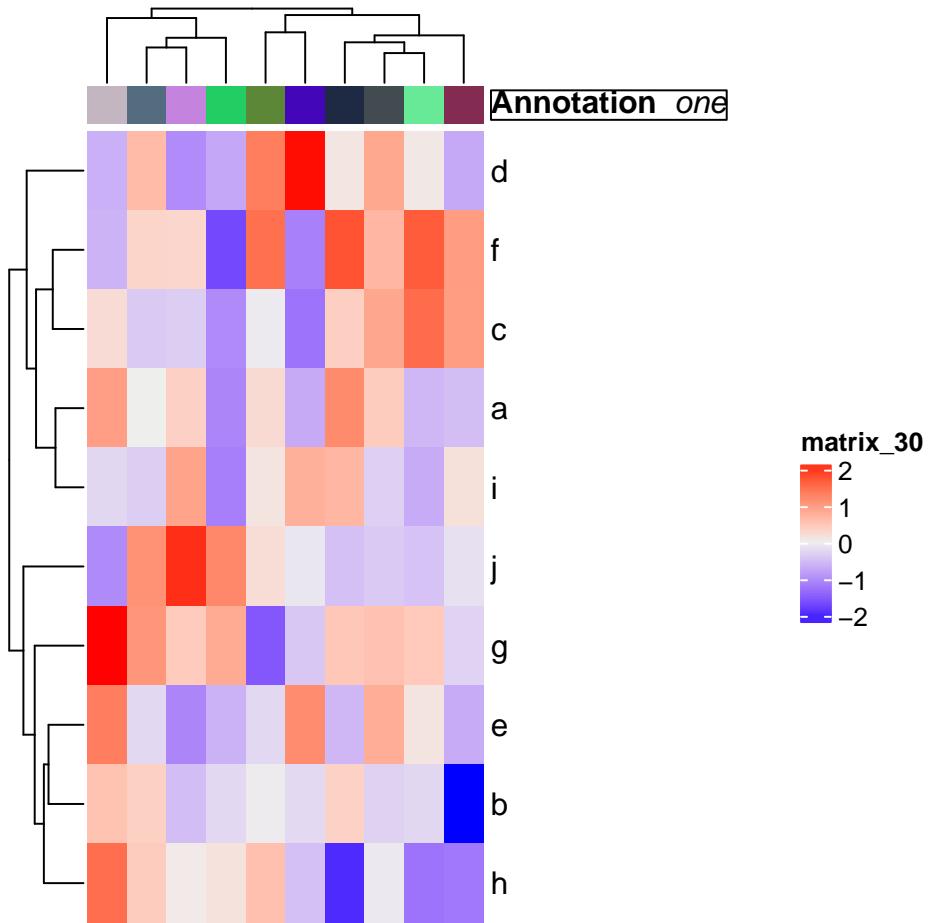
```
Heatmap(mat,
  row_labels = gt_render(letters[1:10], padding = unit(c(2, 10, 2, 10), "pt")),
  row_names_gp = gpar(box_col = rep(c("red", "green"), times = 5)))
```



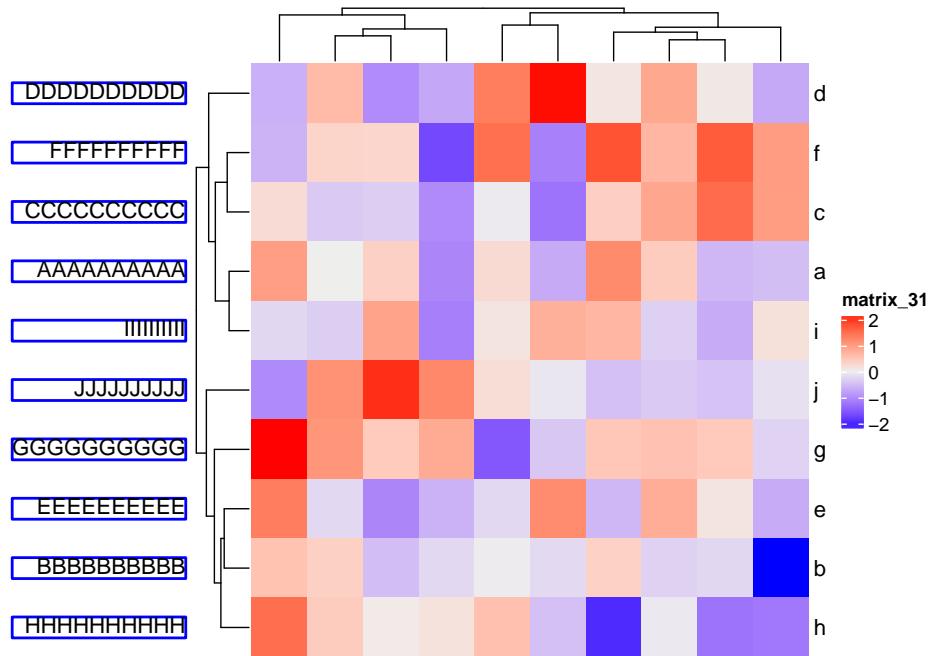
10.3.3 Annotation labels

`annotation_label` argument should be as rendered text.

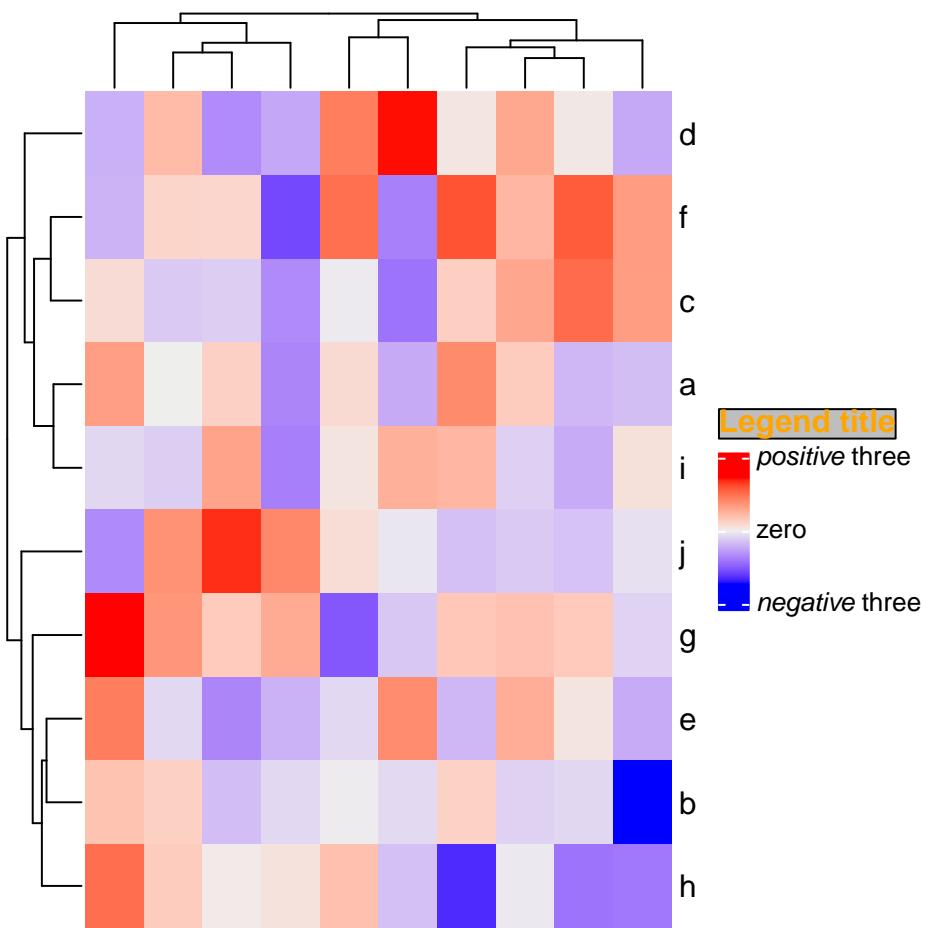
```
ha = HeatmapAnnotation(foo = letters[1:10],
  annotation_label = gt_render("**Annotation** _one_",
    gp = gpar(box_col = "black")),
  show_legend = FALSE)
Heatmap(mat, top_annotation = ha)
```



```
rowAnnotation(
  foo = anno_text(gt_render(sapply(LETTERS[1:10], strrep, 10), align_widths = TRUE),
                  gp = gpar(box_col = "blue", box_lwd = 2),
                  just = "right",
                  location = unit(1, "npc"))
)) + Heatmap(mat)
```



```
Heatmap(mat,
  heatmap_legend_param = list(
    title = gt_render("<span style='color:orange'>**Legend title**</span>"),
    title_gp = gpar(box_fill = "grey"),
    at = c(-3, 0, 3),
    labels = gt_render(c("*negative* three", "zero", "*positive* three")))
))
```



Chapter 11

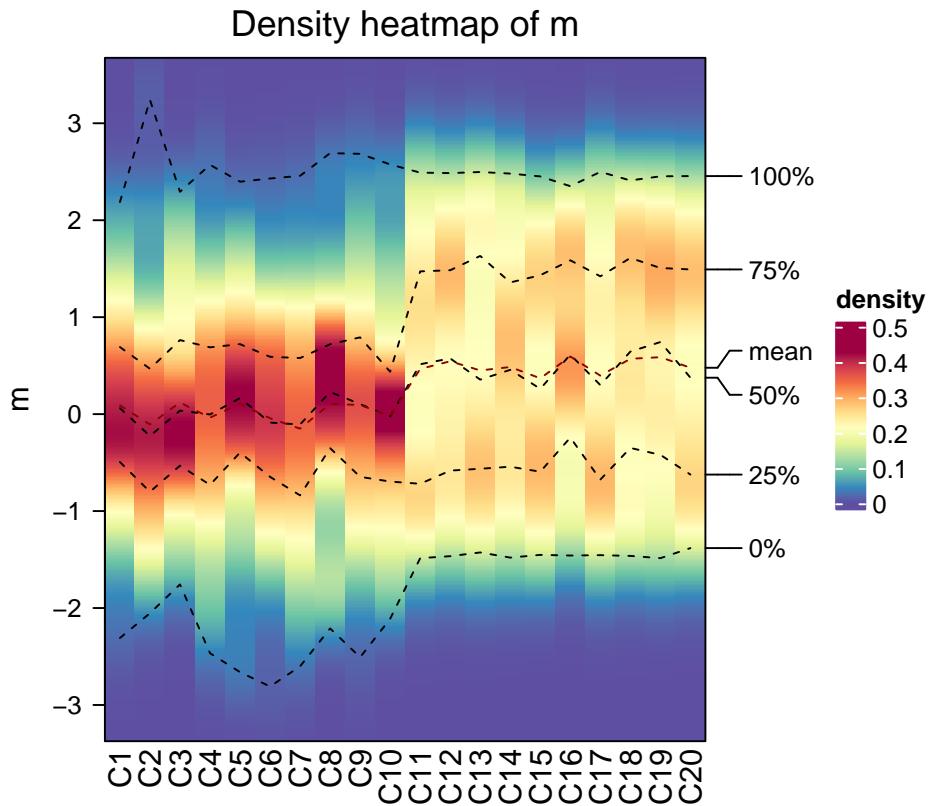
Other High-level Plots

11.1 Density heatmap

To visualize data distribution in a matrix or in a list, we normally use boxplot or violin plot. We can also use colors to map the density values and visualize distribution through a heatmap. It is useful if you have huge number of columns in data to visualize.

In following examples, we use matrix as input data where the density is calculated by columns. The input data can also be a list.

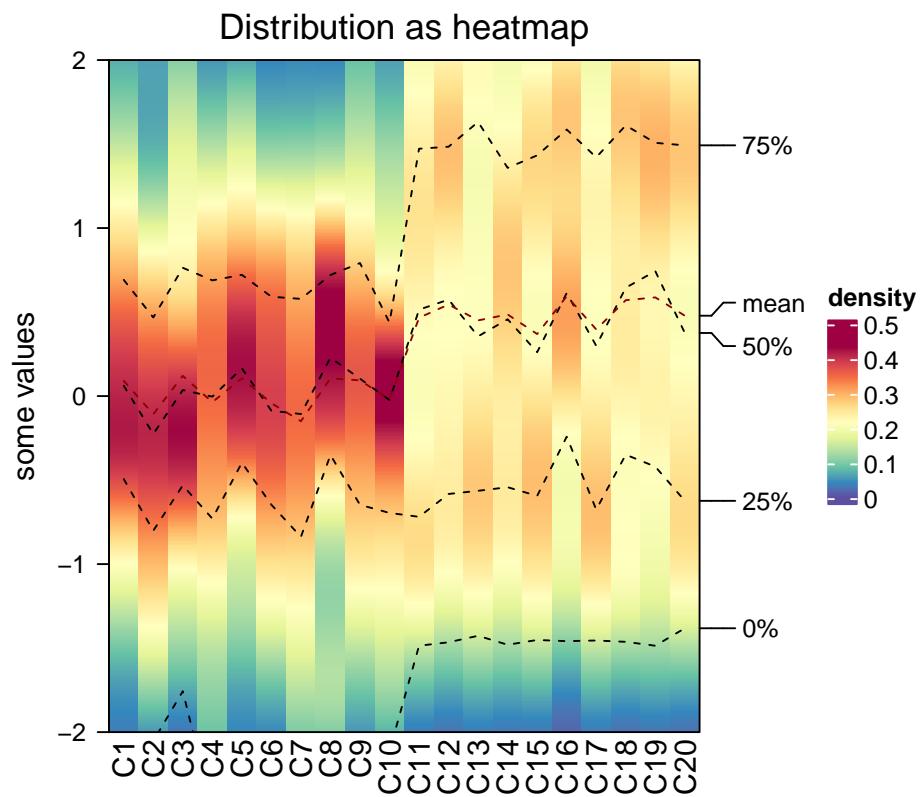
```
set.seed(123)
m = cbind(matrix(rnorm(10*100), ncol = 10),
          matrix(runif(10*100, min = -2, max = 2) + 0.5, ncol = 10))
colnames(m) = paste0("C", 1:ncol(m))
densityHeatmap(m)
```



On the heatmap, there are also lines representing five quantiles and mean values.

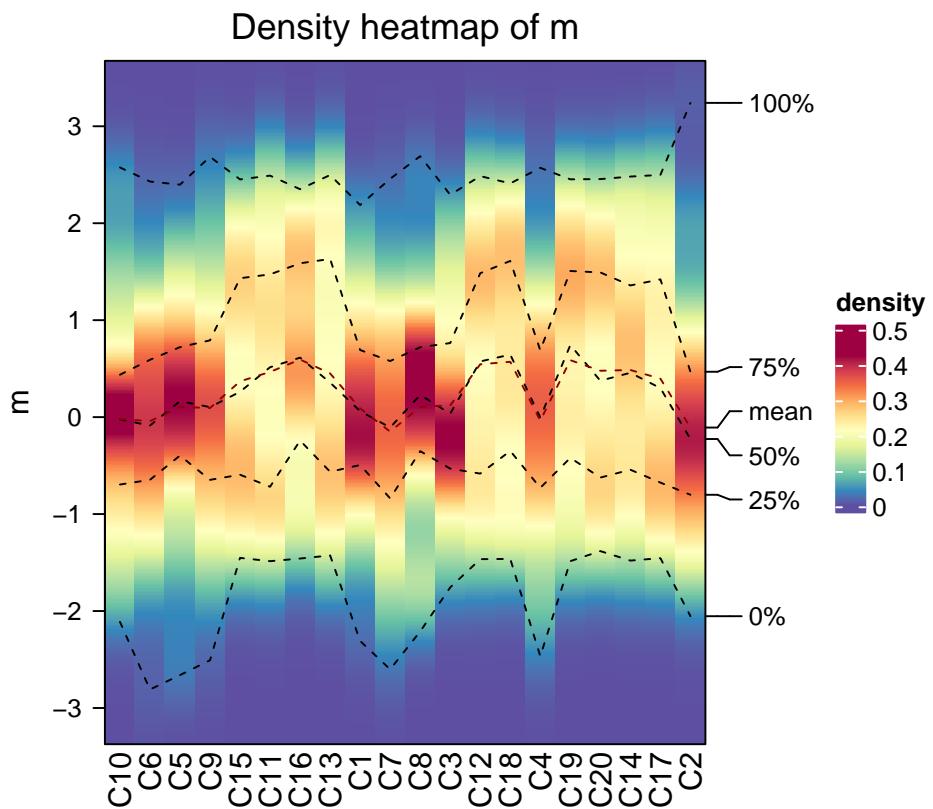
Data range is controlled by `ylim`. Title is controlled by `title` or `column_title`.
Title on y-axis is controlled by `ylab`.

```
densityHeatmap(m, ylim = c(-2, 2), title = "Distribution as heatmap", ylab = "some val")
```



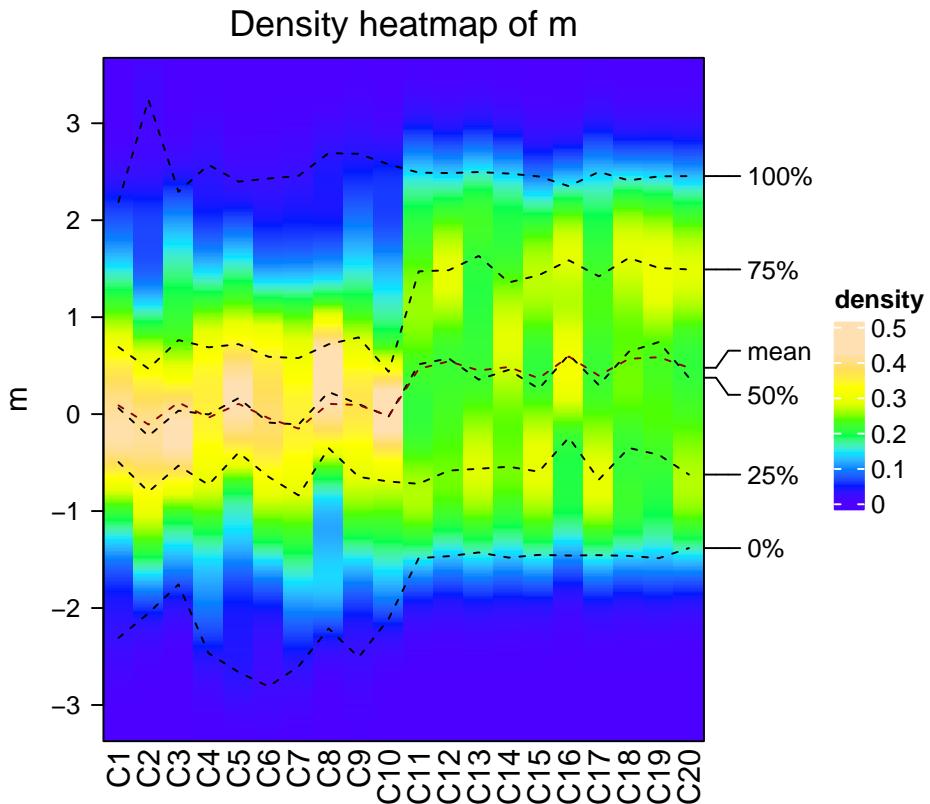
Column order is controlled by `column_order`.

```
densityHeatmap(m, column_order = sample(20, 20))
```



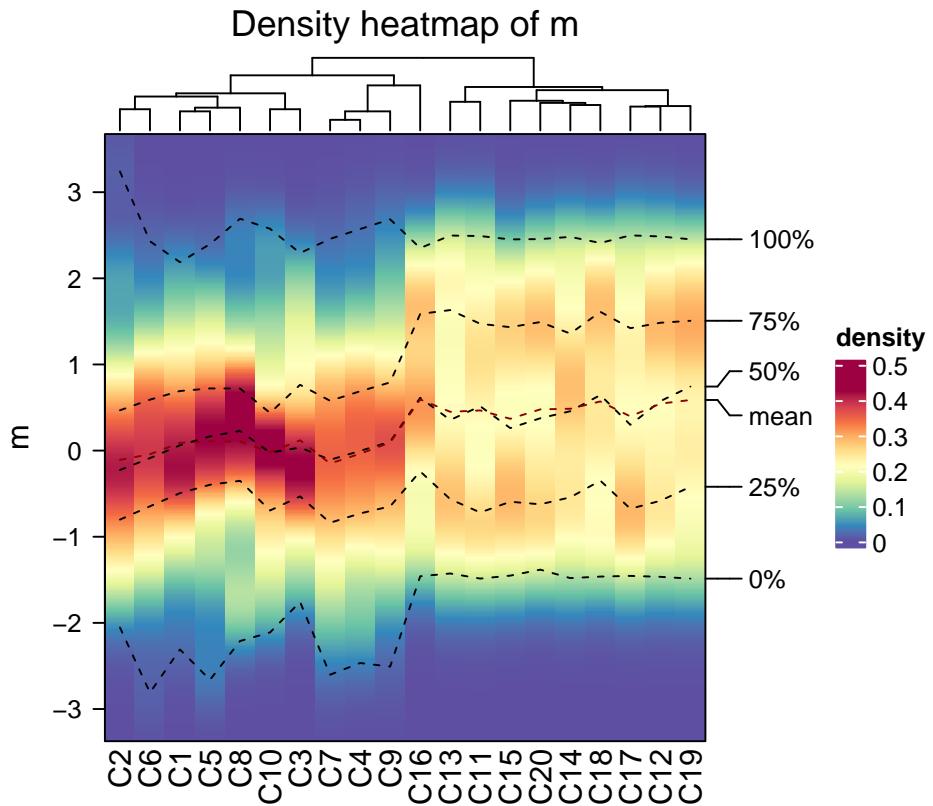
The color for the density values is controlled by `col` which is a vector of colors.

```
densityHeatmap(m, col = topo.colors(10))
```



Internally, the density for all columns are stored as a matrix where rows correspond to the same bins. Since it is a matrix, clustering can be applied on it. There is a special distance method `ks` for measuring similarity between distributions which is the Kolmogorov-Smirnov statistic between two distributions (`ks` distance is the default if `cluster_column = TRUE`).

```
densityHeatmap(m, cluster_columns = TRUE, clustering_distance_columns = "ks")
```

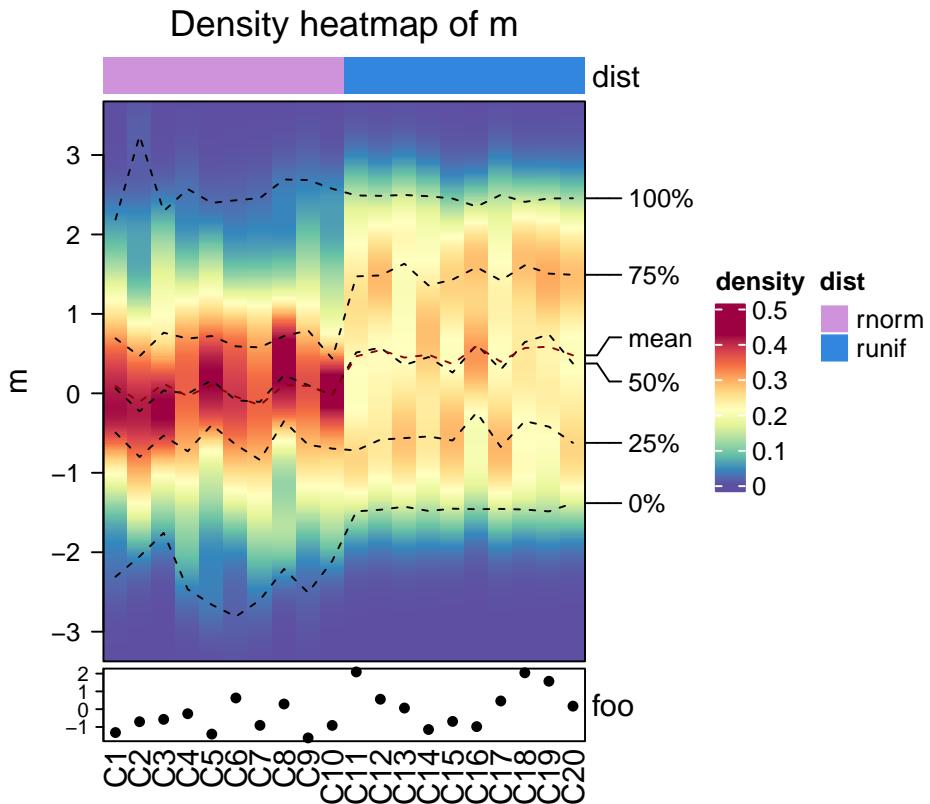


When there are many distributions to calculate the pairwise Kolmogorov-Smirnov distance, `mc.cores` argument can be set to make it running parallelly.

```
densityHeatmap(m, cluster_columns = TRUE, mc.cores = ...)
```

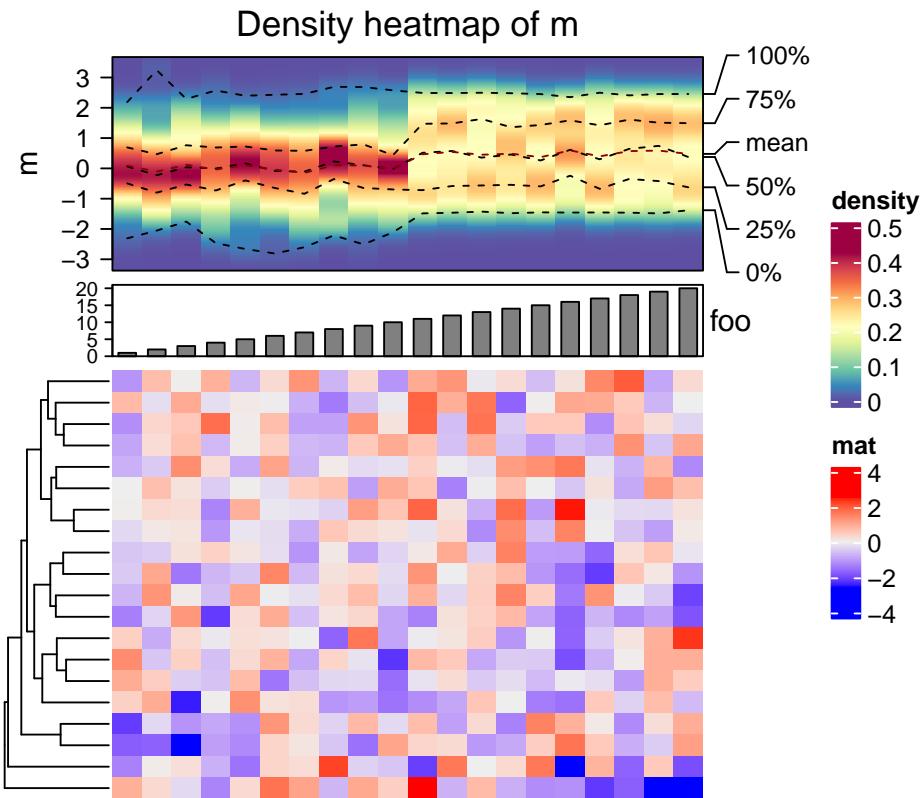
Column annotations can be added as top annotation or bottom annotation.

```
ha1 = HeatmapAnnotation(dist = c(rep("rnorm", 10), rep("runif", 10)))
ha2 = HeatmapAnnotation(foo = anno_points(rnorm(20)))
densityHeatmap(m, top_annotation = ha1, bottom_annotation = ha2)
```



Heatmaps and column annotations can only be concatenated to the density heatmap vertically.

```
densityHeatmap(m) %v%
HeatmapAnnotation(foo = anno_barplot(1:20)) %v%
Heatmap(matrix(rnorm(20*20), ncol = 20), name = "mat", height = unit(6, "cm"))
```



There is also a function `frequencyHeatmap()` which is like a histogram-version of density heatmap. The usage is similar as `densityHeatmap()`:

```
frequencyHeatmap(m)
```

11.2 Stacked summary plot

Multiple annotations and heatmaps can be used to visualize multiple summary statistics for a same set of features. In following example, there are multiple statistics for differential methylated regions (DMRs) from four different subgroups.

```
lt = readRDS(system.file("extdata", package = "ComplexHeatmap", "dmr_summary.rds"))
names(lt)

## [1] "label"          "mean_meth"       "n_gr"           "n_corr"
## [5] "dist_tss"        "gene_anno"       "cgi_anno"       "mat_enrich_gf"
## [9] "mat_pct_st"     "mat_enrich_st"
```

These statistics for DMRs are:

- `label`: The labels for the sets of DMRs. There are DMRs for four subgroups

and for each subgroup, DMRs are separated into hyper-methylated DMRs and hypo-methylated DMRs.

- **mean_meth**: The mean methylation in DMRs in tumor samples and in normal samples.
 - **n_gr**: Number of DMRs in each set.
 - **n_corr**: Percent of DMRs that show significant correlation to nearby genes. The positive correlation and negative correlation are calculated separately.
 - **dist_tss**: Distance to nearby gene TSS. The value is proportion of DMRs in current set which have distance less in 1kb, between 1kb and 5kb, between 5kb and 10kb and more than 10kb.
 - **gene_anno**: The proportion of DMRs that overlap to genes or intergenic regions.
 - **cgi_anno**: The proportion of DMRs that overlap to CpG islands or CGI shores.
 - **mat_enrich_gf**: The enrichment to a list of genomic features. Positive values mean over representation.
 - **mat_pct_st**: The proportion of DMRs that overlap to chromatin states.
 - **mat_enrich_st**: The enrichment to the chromatin states.

Attach all these variables to the working environment.

attach(lt)

Since we have many statistics to visualize, we first define the colors. We define color mapping functions for the statistics which we want to visualize as heatmaps and color vectors for those we want to visualize as barplots.

```
library(circlize)
meth_col_fun = colorRamp2(c(0, 0.5, 1), c("blue", "white", "red"))
corr_col = c("green", "red")
dist_tss_col = c("#FF0000", "#FF7352", "#FFB299", "#FFD9CB")
gene_anno_col = c("green", "blue")
cgi_anno_col = c("#FFA500", "#FFD191")
z_score_col_fun = colorRamp2(c(-200, 0, 200), c("green", "white", "red"))
state_col = c("#FF0000", "#008000", "#C2E105", "#8A91D0", "#CD5C5C", "#808080", "#000000")
```

The construction of the heatmap list very straightforward. Each statistic is constructed as a heatmap or a row annotation.

```

anno_width = unit(3, "cm")
ht_list = rowAnnotation(text = anno_text(label, location = unit(1, "npc"), just = "right",
  gp = gpar(fontsize = 12)))

ht_list = ht_list + Heatmap(mean_meth, name = "mean_meth", col = meth_col_fun,
  cluster_rows = FALSE, row_title = NULL, cluster_columns = FALSE, show_row_names = FALSE,
  heatmap_legend_param = list(title = "Methylation"), width = ncol(mean_meth)*unit(4, "mm")) +
rowAnnotation("n_gr" = anno_barplot(n_gr, bar_width = 1, width = anno_width),
  show_annotation_name = FALSE) +

```

```

rowAnnotation("n_corr" = anno_barplot(n_corr, bar_width = 1, gp = gpar(fill = corr_col,
  width = anno_width), show_annotation_name = FALSE) +
rowAnnotation("dist_tss" = anno_barplot(dist_tss, bar_width = 1, gp = gpar(fill = dist_col,
  width = anno_width), show_annotation_name = FALSE) +
rowAnnotation("gene_anno" = anno_barplot(gene_anno, bar_width = 1, gp = gpar(fill = gene_col,
  width = anno_width), show_annotation_name = FALSE) +
rowAnnotation("cgi_anno" = anno_barplot(cgi_anno, bar_width = 1, gp = gpar(fill = cgi_col,
  width = anno_width), show_annotation_name = FALSE) +
Heatmap(mat_enrich_gf, name = "enrich_gf", col = z_score_col_fun, cluster_columns = FALSE,
  width = unit(ncol(mat_enrich_gf)*4, "mm"), column_title = "", heatmap_legend_param = list(title = "Z-score")) +
rowAnnotation("pct_st" = anno_barplot(mat_pct_st, bar_width = 1, gp = gpar(fill = state_col,
  width = anno_width), show_annotation_name = FALSE) +
Heatmap(mat_enrich_st, name = "enrich_st", col = z_score_col_fun, cluster_columns = FALSE,
  width = unit(ncol(mat_enrich_st)*6, "mm"), column_title = "", show_heatmap_legend = TRUE,
  column_names_gp = gpar(col = state_col), show_row_names = FALSE)

```

Since annotation barplots do not generate legends, we manually construct these legends with `Legend()` function.

```

lgd_list = list(
  Legend(labels = c("gene", "intergenic"), title = "Gene annotation",
    legend_gp = gpar(fill = gene_anno_col)),
  Legend(labels = c("<1kb", "1kb~5kb", "5kb~10kb", ">10kb"), title = "Distance to TSS",
    legend_gp = gpar(fill = dist_tss_col)),
  Legend(labels = c("CGI", "CGI shore"), title = "CGI annotation",
    legend_gp = gpar(fill = cgi_anno_col)),
  Legend(labels = colnames(mat_enrich_st), title = "Chromatin states",
    legend_gp = gpar(fill = state_col)))
)

```

When drawing the heatmap list, the rows of all heatmaps and annotations are split into two major groups. Note in the first `Heatmap()` which corresponds to the mean methylation matrix, we set `row_title = NULL` to remove the row titles which is from row splitting.

Since later we will add titles for the annotations, we allocate white space on top of the whole plotting region by `padding` argument. Also we concatenate the self-defined legend list to the heatmap legend list and put them horizontally at the bottom of the heatmap list.

```

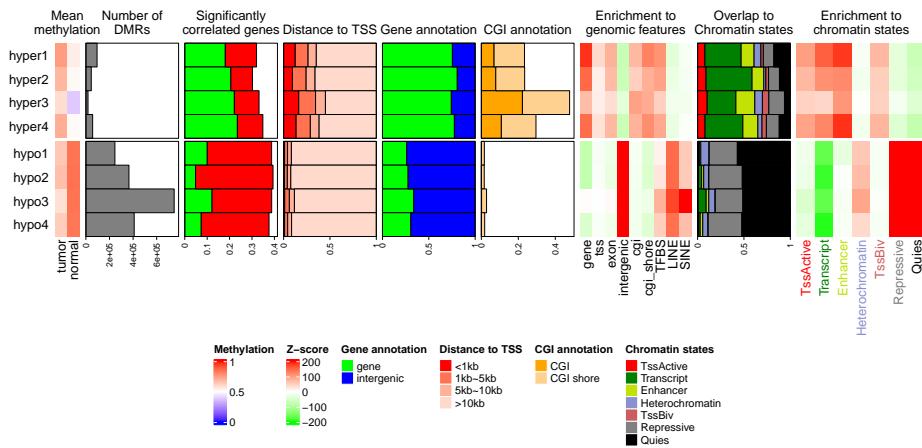
draw(ht_list, padding = unit(c(2, 2, 20, 2), "mm"), row_split = gsub("\\\\d+$", "", labels),
  heatmap_legend_list = lgd_list, heatmap_legend_side = "bottom")
anno_title = c("n_gr" = "Number of nDMRs", "n_corr" = "Significantly\\ncorrelated genes",
  "gene_anno" = "Gene annotation", "dist_tss" = "Distance to TSS",
  "cgi_anno" = "CGI annotation", "pct_st" = "Overlap to\\nChromatin states")
for(an in names(anno_title)) {

```

```

decorate_annotation(an, {
  grid.text(anno_title[an], y = unit(1, "npc") + unit(3, "mm"), just = "bottom")
})
ht_title = c("mean_meth" = "Mean\nmethylation", "enrich_gf" = "Enrichment to\nngenomic features",
            "enrich_st" = "Enrichment to\nchromatin states")
for(an in names(ht_title)) {
  decorate_heatmap_body(an, {
    grid.text(ht_title[an], y = unit(1, "npc") + unit(3, "mm"), just = "bottom")
  })
}

```



Similarly, the multiple statistics can also be arranged vertically. In following example, we visualize several statistics for a list of genomic regions in 40 samples, from four subgroups. The statistics are:

- `prop`: The proportion in the genome.
- `median_length`: The median length of regions in each sample.
- `group`: subgroup labels.

```

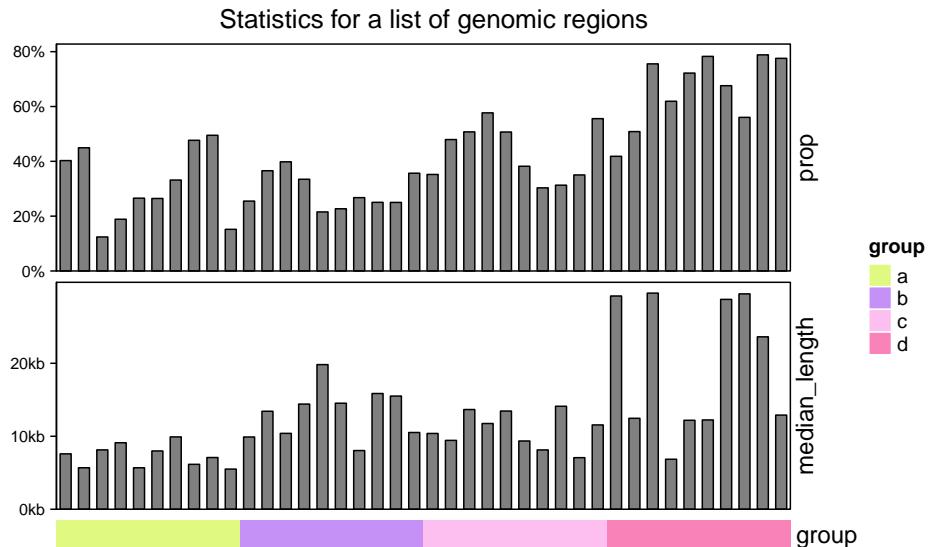
prop = c(
  runif(10, min = 0.1, max = 0.5),
  runif(10, min = 0.2, max = 0.4),
  runif(10, min = 0.3, max = 0.6),
  runif(10, min = 0.4, max = 0.8)
)
median_length = c(
  runif(10, min = 5000, max = 10000),
  runif(10, min = 6000, max = 20000),
  runif(10, min = 7000, max = 15000),
  runif(10, min = 6000, max = 30000)
)

```

```
)
group = rep(letters[1:4], each = 10)
```

Note in following example, there is no heatmap in the list.

```
ht_list = HeatmapAnnotation(prop = anno_barplot(prop, height = unit(4, "cm"),
                                                axis_param = list(at = c(0, 0.2, 0.4, 0.6, 0.8),
                                                                  labels = c("0%", "20%", "40%", "60%", "80%"))),
                                annotation_name_rot = 90) %v%
HeatmapAnnotation(median_length = anno_barplot(median_length, height = unit(4, "cm"),
                                                axis_param = list(at = c(0, 10000, 20000), labels = c("0kb", "10kb", "20kb"))),
                                annotation_name_rot = 90) %v%
HeatmapAnnotation(group = group)
draw(ht_list, column_title = "Statistics for a list of genomic regions")
```



For concatenation of multiple annotations, individual annotations can also be put into one single `HeatmapAnnotation()`. E.g. previous code is almost exactly the same as following code:

```
# code only for demonstration
HeatmapAnnotation(
  prop = anno_barplot(prop, height = unit(4, "cm"),
                      axis_param = list(at = c(0, 0.2, 0.4, 0.6, 0.8),
                                        labels = c("0%", "20%", "40%", "60%", "80%"))),
  median_length = anno_barplot(median_length, height = unit(4, "cm"),
                                axis_param = list(at = c(0, 10000, 20000), labels = c("0kb", "10kb", "20kb"))),
  group = group,
  annotation_name_rot = c(90, 90, 0),
```

```
gap = unit(2, "mm")
) %v% NULL # add NULL to convert single HeatmapAnnotation to HeatmapList
```

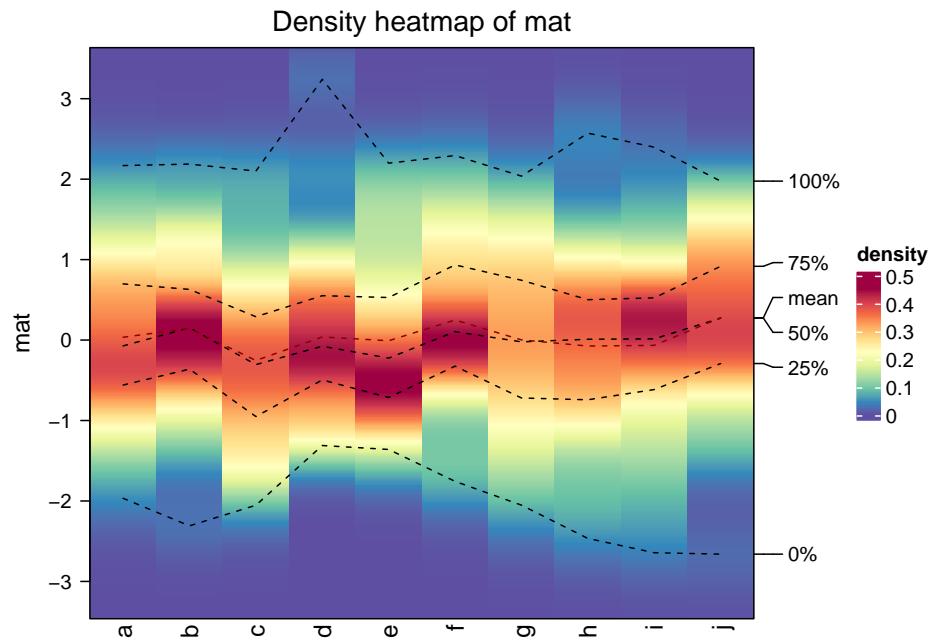

Chapter 12

Three-dimensional ComplexHeatmap

12.1 Motivation

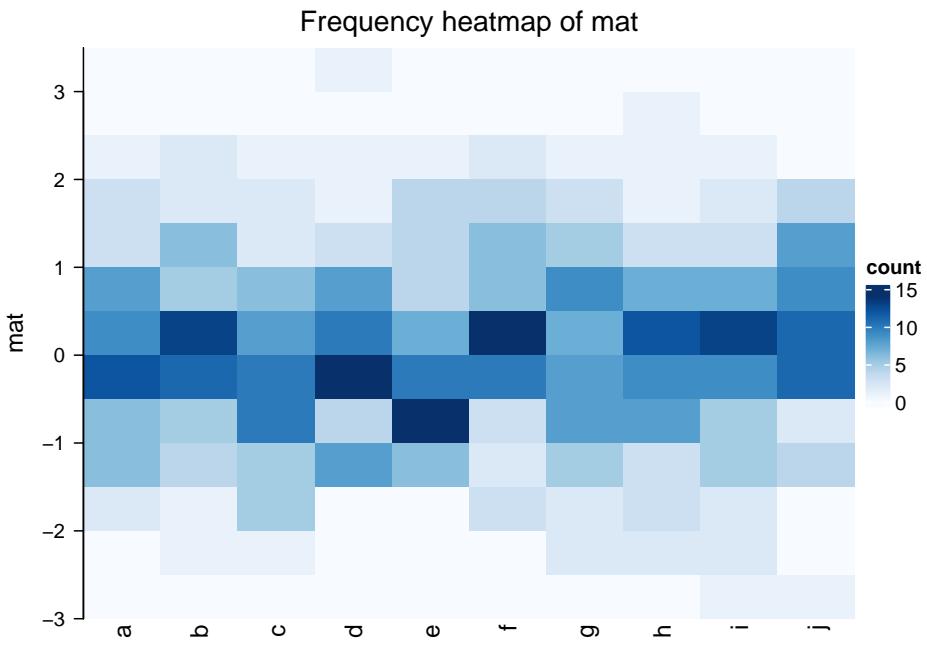
ComplexHeatmap has a `densityHeatmap()` to visualize a list of density distributions, such as in the following example:

```
library(ComplexHeatmap)
set.seed(123)
mat = matrix(rnorm(500), ncol = 10)
colnames(mat) = letters[1:10]
densityHeatmap(mat)
```



In basic R graphics, since distributions can also be visualized by histograms, from **ComplexHeatmap** version 2.7.9, I added a new function `frequencyHeatmap()` which is like a histogram-version of density heatmap. The usage is similar as `densityHeatmap()`:

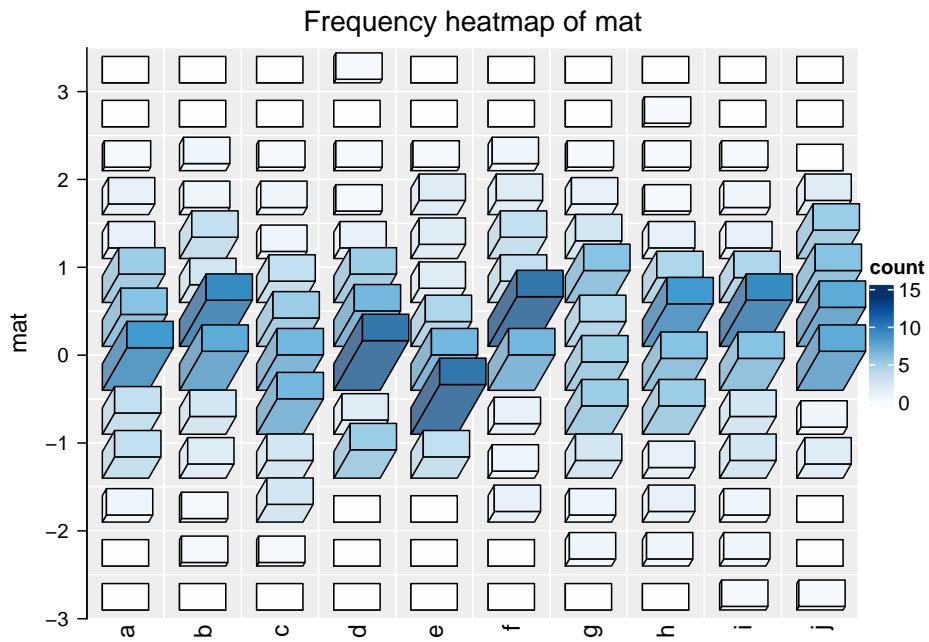
```
frequencyHeatmap(mat)
```



In the previous example, the frequency matrix is visualized as a heatmap. Note you can use different statistic in `frequencyHeatmap()`, i.e., "count", "proportion" or "density".

Well, the frequency heatmap claims to be a histogram-version of density heatmap, but it does not look like histograms at all. Maybe a 3D heatmap with 3D bars is more proper. This can be done by setting argument `use_3d = TRUE` in `frequencyHeatmap()`.

```
frequencyHeatmap(mat, use_3d = TRUE)
```

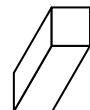


This looks nicer! In the next section I will explain how the 3D Heatmap is implemented.

12.2 Implementation of 3D heatmap

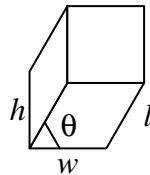
First, we need to draw 3D bars. This can be done by the new function `bar3D()`. The usage is as follows:

```
bar3D(x = 0.5, y = 0.5, w = 0.2, h = 0.2, l = unit(1, "cm"), theta = 60)
```



The arguments are:

- `x`: x coordinate of the center point in the bottom face. Value should be a `unit` object. If it is numeric, the default unit is `npc`.
- `y`: y coordinate of the center point in the bottom face.
- `w`: Width of the bar (in the x direction). See the following figure.
- `h`: Height of the bar (in the y direction). See the following figure.
- `l`: Length of the bar (in the z direction). See the following figure.
- `theta`: Angle for the projection. See the following figure. Note `theta` can only take value between 0 and 90.



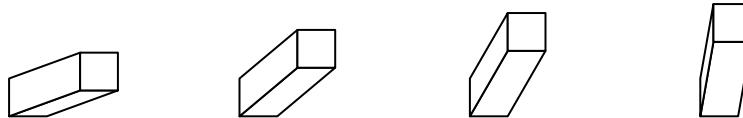
`fill` argument sets the color. To enhance the visual effect of 3D visualization, the three faces actually have slightly different brightness.

```
bar3D(x = seq(0.2, 0.8, length = 4), y = 0.5, w = unit(5, "mm"), h = unit(5, "mm"),
      l = unit(1, "cm"), fill = c("red", "green", "blue", "purple"))
```



`theta` argument sets the angle of the projection. Since I am a right-hand person, my left hand is more free so that it feels like it's giving invisible force to push the bars to the right, thus, `theta` can only take value between 0 and 90. :)

```
bar3D(x = seq(0.2, 0.8, length = 4), y = 0.5, w = unit(5, "mm"), h = unit(5, "mm"),
      l = unit(1, "cm"), theta = c(20, 40, 60, 80))
```

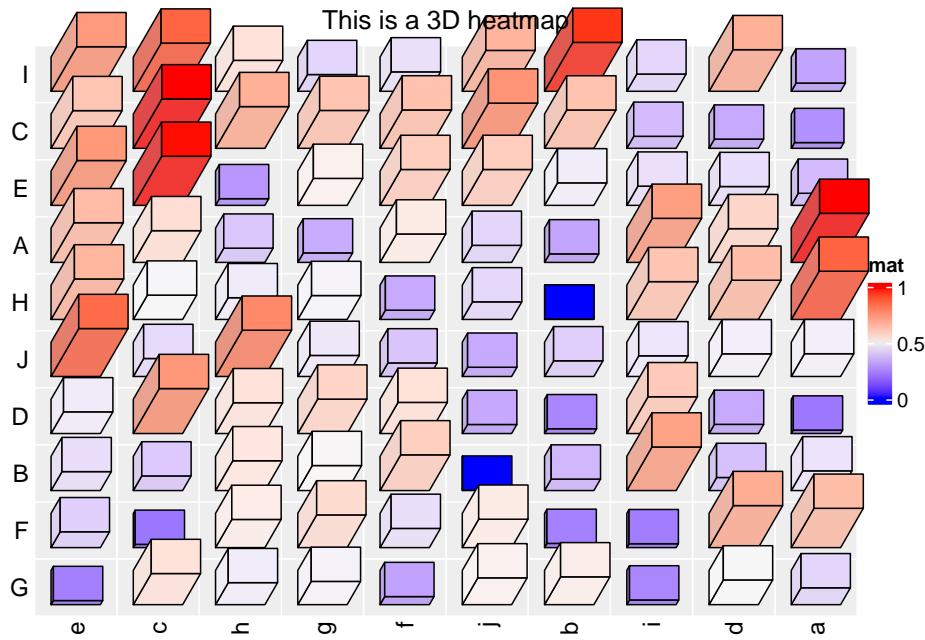


To add bars to heatmap cells, we can simply implement `bar3D()` in `cell_fun` or `layer_fun` where we add each bar to each cell. Here we have a new function `Heatmap3D()` which simplifies this. `Heatmap3D()` accepts almost all the arguments in `Heatmap()` and the only difference is each cell has a 3D bar of which the height corresponds to its value.

`Heatmap3D()` only allows non-negative matrix as input. Also default values are changes for some arguments, such as row names are put on the left side of heatmap, and clusterings are still applied but the dendograms are not drawn by default.

Following is a demonstration of the usage of `Heatmap3D()`:

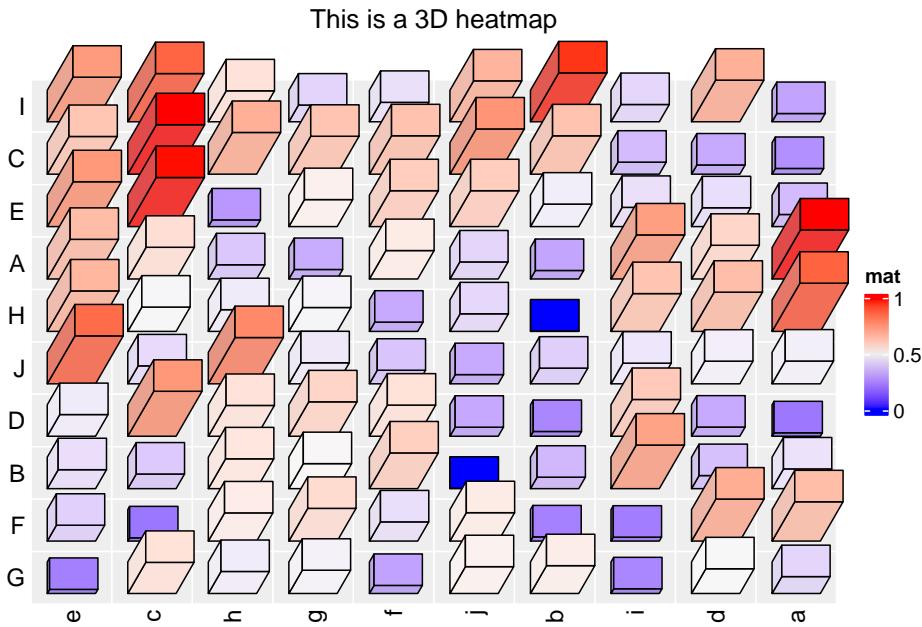
```
set.seed(7)
mat = matrix(runif(100), 10)
rownames(mat) = LETTERS[1:10]
colnames(mat) = letters[1:10]
Heatmap3D(mat, name = "mat", column_title = "This is a 3D heatmap")
```



In the previous example, if bars close to the top of the heatmap or to the right of the heatmap have too large length, they will overlap to the heatmap title or the legend, in this case, we need to manually adjust the space between, *e.g.*, title and the heatmap body.

In **ComplexHeatmap**, there are several global options that control the spaces between heatmap components. To solve the problem in the previous example, we can manually set a proper value for `ht_opt$HEATMAP_LEGEND_PADDING` and `ht_opt$TITLE_PADDING`.

```
ht_opt$HEATMAP_LEGEND_PADDING = unit(5, "mm")
ht_opt$TITLE_PADDING = unit(c(9, 2), "mm") # bottom and top padding
Heatmap3D(mat, name = "mat", column_title = "This is a 3D heatmap")
```



Reset the global options by `ht_opt(RESET = TRUE)`:

```
ht_opt(RESET = TRUE)
```

Next I demonstrate another example which is applied to the well-known measles vaccine dataset. First I show the “normal 2D heatmap.” Code for generating the heatmap can be found [here](#).

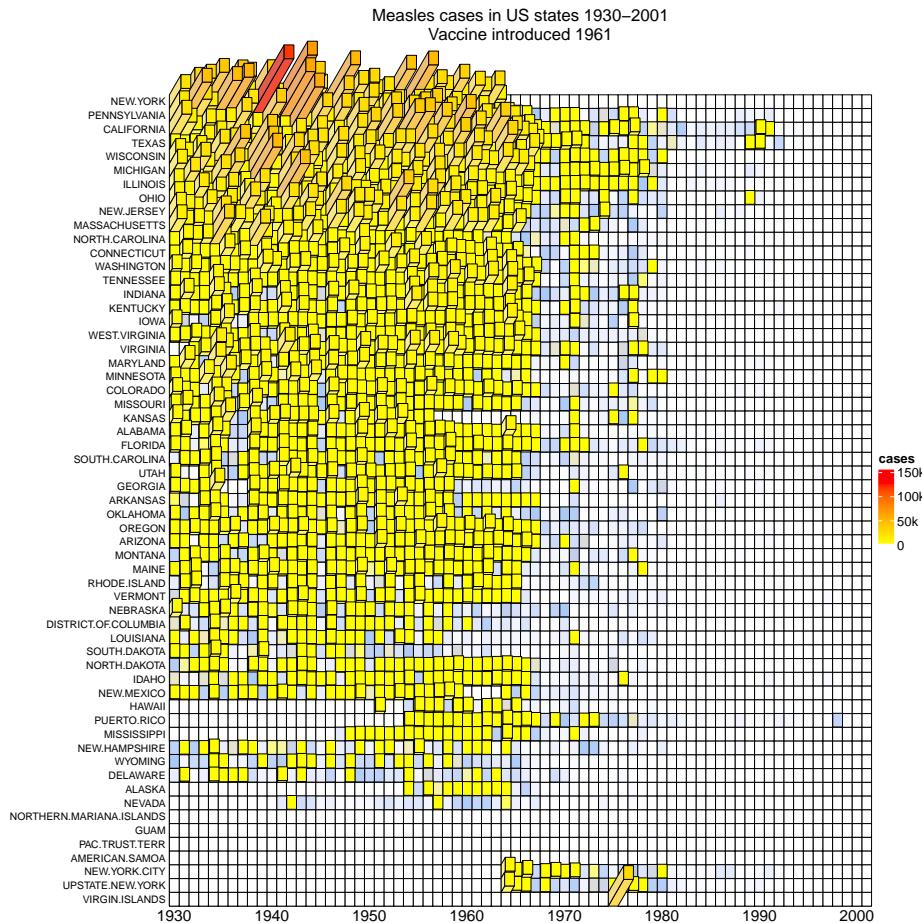
To change it to 3D visualization, simply replace `Heatmap()` with `Heatmap3D()` and most of the original arguments for `Heatmap()` can still be put there. For simplicity, in the 3D heatmap, I removed the top annotation and the right annotation.

```
mat = readRDS(system.file("extdata", "measles.rds", package = "ComplexHeatmap"))
year_text = as.numeric(colnames(mat))
year_text[year_text %% 10 != 0] = ""
ha_column = HeatmapAnnotation(
  year = anno_text(year_text, rot = 0, location = unit(1, "npc"), just = "top")
)
col_fun = circlize::colorRamp2(c(0, 800, 1000, 127000), c("white", "cornflowerblue", "yellow", "red"))
ht_opt$TITLE_PADDING = unit(c(15, 2), "mm")
Heatmap3D(mat, name = "cases", col = col_fun,
  cluster_columns = FALSE, show_row_dend = FALSE,
  show_column_names = FALSE,
  row_names_side = "left", row_names_gp = gpar(fontsize = 8),
  column_title = 'Measles cases in US states 1930-2001\nVaccine introduced 1961',
  bottom_annotation = ha_column,
```

```

heatmap_legend_param = list(at = c(0, 5e4, 1e5, 1.5e5),
                           labels = c("0", "50k", "100k", "150k")),
# new arguments for Heatmap3D()
bar_rel_width = 1, bar_rel_height = 1, bar_max_length = unit(2, "cm")
)

```



By the way, it is also possible to turn the static 3D heatmap to an interactive Shiny application by package **InteractiveComplexHeatmap**. See the following figure:

`Heatmap3D()` can do a lot of things that are the same as `Heatmap()`, such as adding annotations, splitting the heatmap or concatenating more heatmaps by `+/%v%`. But since 3D visualization is in general not a good idea and it actually won't give you more information than what you can get from 2D visualization, thus, if you want to use `Heatmap3D()`, you better keep it as simple as possible. Also, please apply it to small matrices, it will take long time to generate for

large matrices.

Chapter 13

Genome-level heatmap

Many people are interested in making genome-scale heatmap with multiple tracks, like examples here and here. In this chapter, I will demonstrate how to implement it with **ComplexHeatmap**.

To make genome-scale plot, we first need the ranges on chromosome-level. There are many ways to obtain this information. In following, I use `circlize::read.chromInfo()` function.

```
library(circlize)
library(GenomicRanges)
chr_df = read.chromInfo()$df
chr_df = chr_df[chr_df$chr %in% paste0("chr", 1:22), ]
chr_gr = GRanges(seqnames = chr_df[, 1], ranges = IRanges(chr_df[, 2] + 1, chr_df[, 3]))
chr_gr

## GRanges object with 22 ranges and 0 metadata columns:
##      seqnames      ranges strand
##      <Rle>    <IRanges>  <Rle>
## [1]   chr1  1-249250621    *
## [2]   chr2  1-243199373    *
## [3]   chr3  1-198022430    *
## [4]   chr4  1-191154276    *
## [5]   chr5  1-180915260    *
## ...
## [18]  chr18 1-78077248     *
## [19]  chr19 1-59128983     *
## [20]  chr20 1-63025520     *
## [21]  chr21 1-48129895     *
## [22]  chr22 1-51304566     *
## -----
```

```
## seqinfo: 22 sequences from an unspecified genome; no seqlengths
```

In the final heatmap, each row (if the genomic direction is vertical) or each column (if the genomic direction is horizontal) actually represents a genomic window, thus we need to split the genome with equal-width windows. Here I use `EnrichedHeatmap::makeWindows()` function to split the genome by 1MB window (The two meta-columns in `chr_window` can be ignored here).

```
library(EnrichedHeatmap)
chr_window = makeWindows(chr_gr, w = 1e6)
chr_window
```

```
## GRanges object with 2875 ranges and 2 metadata columns:
##           seqnames      ranges strand | .i_query .i_window
##           <Rle>      <IRanges>  <Rle> | <integer> <integer>
## [1]   chr1    1-1000000 * |       1     1
## [2]   chr1  1000001-2000000 * |       1     2
## [3]   chr1  2000001-3000000 * |       1     3
## [4]   chr1  3000001-4000000 * |       1     4
## [5]   chr1  4000001-5000000 * |       1     5
## ...
## [2871] chr22 46000001-47000000 * |      22    47
## [2872] chr22 47000001-48000000 * |      22    48
## [2873] chr22 48000001-49000000 * |      22    49
## [2874] chr22 49000001-50000000 * |      22    50
## [2875] chr22 50000001-51000000 * |      22    51
## -----
## seqinfo: 22 sequences from an unspecified genome; no seqlengths
```

To visualize genome-scale signals as a heatmap as well as other tracks, now the task is to calculate the average signals in the 1MB windows by overlapping the genomic windows and the genomic signals. Here I implement a function `average_in_window()`. This function is adapted from `HilbertCurve` package since there is similar task there.

```
average_in_window = function(window, gr, v, method = "weighted", empty_v = NA) {

  if(missing(v)) v = rep(1, length(gr))
  if(is.null(v)) v = rep(1, length(gr))
  if(is.atomic(v) && is.vector(v)) v = cbind(v)

  v = as.matrix(v)
  if(is.character(v) && ncol(v) > 1) {
    stop("`v` can only be a character vector.")
  }

  if(length(empty_v) == 1) {
    empty_v = rep(empty_v, ncol(v))
  }
}
```

```

}

u = matrix(rep(empty_v, each = length(window)), nrow = length(window), ncol = ncol(v))

mtch = as.matrix(findOverlaps(window, gr))
intersect = pintersect(window[mtch[,1]], gr[mtch[,2]])
w = width(intersect)
v = v[mtch[,2], , drop = FALSE]
n = nrow(v)

ind_list = split(seq_len(n), mtch[, 1])
window_index = as.numeric(names(ind_list))
window_w = width(window)

if(is.character(v)) {
  for(i in seq_along(ind_list)) {
    ind = ind_list[[i]]
    if(is.function(method)) {
      u>window_index[i], ] = method(v[ind], w[ind], window_w[i])
    } else {
      tb = tapply(w[ind], v[ind], sum)
      u>window_index[i], ] = names(tb[which.max(tb)])
    }
  }
} else {
  if(method == "w0") {
    gr2 = reduce(gr, min.gapwidth = 0)
    mtch2 = as.matrix(findOverlaps(window, gr2))
    intersect2 = pintersect(window[mtch2[, 1]], gr2[mtch2[, 2]])

    width_intersect = tapply(width(intersect2), mtch2[, 1], sum)
    ind = unique(mtch2[, 1])
    width_setdiff = width(window[ind]) - width_intersect

    w2 = width(window[ind])

    for(i in seq_along(ind_list)) {
      ind = ind_list[[i]]
      x = colSums(v[ind, , drop = FALSE]*w[ind])/sum(w[ind])
      u>window_index[i], ] = (x*width_intersect[i] + empty_v*width_setdiff[i])/w2[i]
    }
  } else if(method == "absolute") {
    for(i in seq_along(ind_list)) {
      u>window_index[i], ] = colMeans(v[ind_list[[i]]], , drop = FALSE)
    }
  }
}

```

```

        }

    } else if(method == "weighted") {
        for(i in seq_along(ind_list)) {
            ind = ind_list[[i]]
            u>window_index[i], ] = colSums(v[ind, , drop = FALSE]*w[ind])/sum(w[ind])
        }
    } else {
        if(is.function(method)) {
            for(i in seq_along(ind_list)) {
                ind = ind_list[[i]]
                u>window_index[i], ] = method(v[ind], w[ind], window_w[i])
            }
        } else {
            stop("wrong method.")
        }
    }
}

return(u)
}

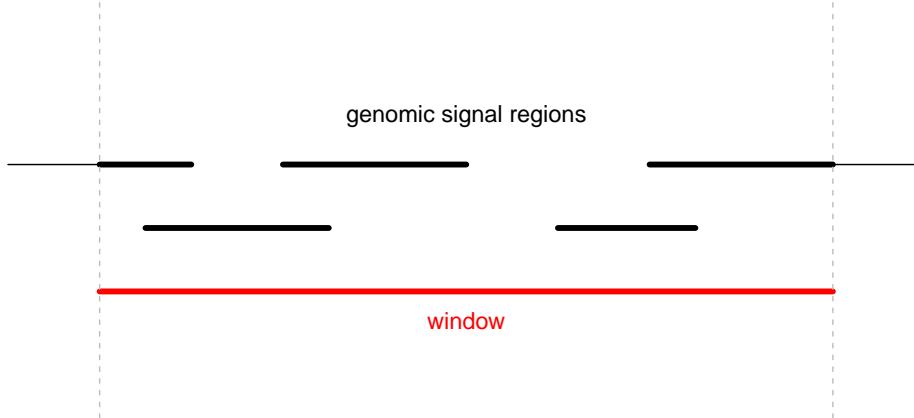
```

In `average_in_window()` function, there are following arguments:

- `window`: A `GRanges` object of the genomic windows.
- `gr`: A `GRanges` object of the genomic signals.
- `v`: A vector or a matrix. This is the value associated with `gr` and it should have the same length or `nrow` as `gr`. `v` can be numeric or character. If it is `missing` or `NULL`, a value of one is assign to every region in `gr`. If `v` is numeric, it can be a vector or a matrix, and if `v` is character, it can only be a vector.
- `method`: Method to summarize the signals for every genomic window.
- `empty_v`: The default value for the window if no region in `gr` overlaps to it.

The function returns a matrix with the same row length and order as `window`.

The overlapping model is illustrated in the following plot. The red line in the bottom represents a certain genomic window. Black lines on the top are the regions for genomic signals that overlap with the window. The thick lines indicate the intersected part between the signal regions and the window.



For a given window, n is the number of signal regions which overlap with the window (it is 5 in the above plot), w_i is the width of the intersected segments (black thick lines), and x_i is the signal value associated with the original regions.

1. If the signals are numeric, either as a vector or a matrix, there are three pre-defined methods:

The “absolute” method is denoted as v_a and is simply calculated as the mean of all signal regions regardless of their width.

$$v_a = \frac{\sum_i^n x_i}{n}$$

The “weighted” method is denoted as v_w and is calculated as the mean of all signal regions weighted by the width of their intersections. This is the default method for numeric signals.

$$v_w = \frac{\sum_i^n x_i w_i}{\sum_i^n w_i}$$

“Absolute” and “weighted” methods should be applied when background values should not be taken into consideration. For example, when summarizing the mean methylation in a small window, non-CpG background should be ignored, because methylation is only associated with CpG sites and not with other positions.

The “w0” method is the weighted mean between the intersected parts and un-intersected parts.

$$v_{w0} = \frac{v_w W}{W + W'}$$

W is sum of width of the intersected parts ($\sum_i^n w_i$) and W' is the sum of width for the non-intersected parts.

2. **If the signals are as a character vector**, denote all levels encoded in x_i as A and a certain level of A is denoted as a , the final value assigned to the window is the level of which the corresponding segments have the maximal sum of widths.

$$\arg \max_{a \in A} \sum_i^n I(x_i = a) \cdot w_i$$

According to these rules, when the signal value `v` is numeric, the argument `method` can be one of `weighted` (default), `absolute` and `w0`, and when `v` is character, the value for `method` is ignored.

Besides the pre-defined values, `method` can also be a user-defined function and it works both for numeric signals and character signals. The user-defined function should accept three arguments: `x`, `w` and `gw`. This function is applied to every genomic window. The three arguments are:

- `x`: The signal values that fall in the genomic window (as shown in the previous plot).
- `w`: The associated segment widths.
- `gw`: The width of the current genomic window.

The user-defined function should only return a scalar variable.

OK, now with the function `average_in_window()`, I can convert the genomic signals to a window-based matrix. In the following example, I generate approximately 1000 random genomic regions with 10 columns of random values (to simulate 10 samples).

```
bed1 = generateRandomBed(nr = 1000, nc = 10) # generateRandomBed() is from circlize package
# convert to a GRanges object
gr1 = GRanges(seqnames = bed1[, 1], ranges = IRanges(bed1[, 2], bed1[, 3]))

num_mat = average_in_window(chr_window, gr1, bed1[, -(1:3)])
dim(num_mat)

## [1] 2875    10

head(num_mat)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
## [2,]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
## [3,]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
## [4,]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
## [5,]   NA   NA   NA   NA   NA   NA   NA   NA   NA   NA
```

```
## [6,] NA NA NA NA NA NA NA NA NA
```

The first five genomic windows have no value associated because no region in `gr1` overlaps to them, thus, they take the value from `empty_v` which is by default NA.

The second data to visualize is 10 lists of genomic regions with character signals (let's assume they are copy number variation results from 10 samples). In each random regions, I additionally sample 20 from them, just to make them sparse in the genome.

```
bed_list = lapply(1:10, function(i) {
  generateRandomBed(nr = 1000, nc = 1,
    fun = function(n) sample(c("gain", "loss"), n, replace = TRUE))
})
char_mat = NULL
for(i in 1:10) {
  bed = bed_list[[i]]
  bed = bed[sample(nrow(bed), 20), , drop = FALSE]
  gr_cnv = GRanges(seqnames = bed[, 1], ranges = IRanges(bed[, 2], bed[, 3]))

  char_mat = cbind(char_mat, average_in_window(chr_window, gr_cnv, bed[, 4]))
}
```

The third data to visualize is simply genomic regions with two numeric columns where both columns will be visualized as a point track and the first column will be visualized as a barplot track.

```
bed2 = generateRandomBed(nr = 100, nc = 2)
gr2 = GRanges(seqnames = bed2[, 1], ranges = IRanges(bed2[, 2], bed2[, 3]))

v = average_in_window(chr_window, gr2, bed2[, 4:5])
```

The fourth data to visualize is a list of gene symbols that we want to mark in the plot. `gr3` contains genomic positions for the genes as well as their symbols. The variable `at` contains the row indice of the corresponding windows in `chr_window` and `labels` contains the gene names. As shown in the following code, I simply use `findOverlaps()` to associate gene regions to genomic windows.

```
bed3 = generateRandomBed(nr = 40, nc = 0)
gr3 = GRanges(seqnames = bed3[, 1], ranges = IRanges(bed3[, 2], bed3[, 2]))
gr3$gene = paste0("gene_", 1:length(gr3))

mtch = as.matrix(findOverlaps(chr_window, gr3))
at = mtch[, 1]
labels = mcols(gr3)[mtch[, 2], 1]
```

Now I have all the variables and are ready for making the heatmaps. Before doing that, to better control the heatmap, I set `chr` as a factor to control the order of chromosomes in the final plot and I create a variable `subgroup` to

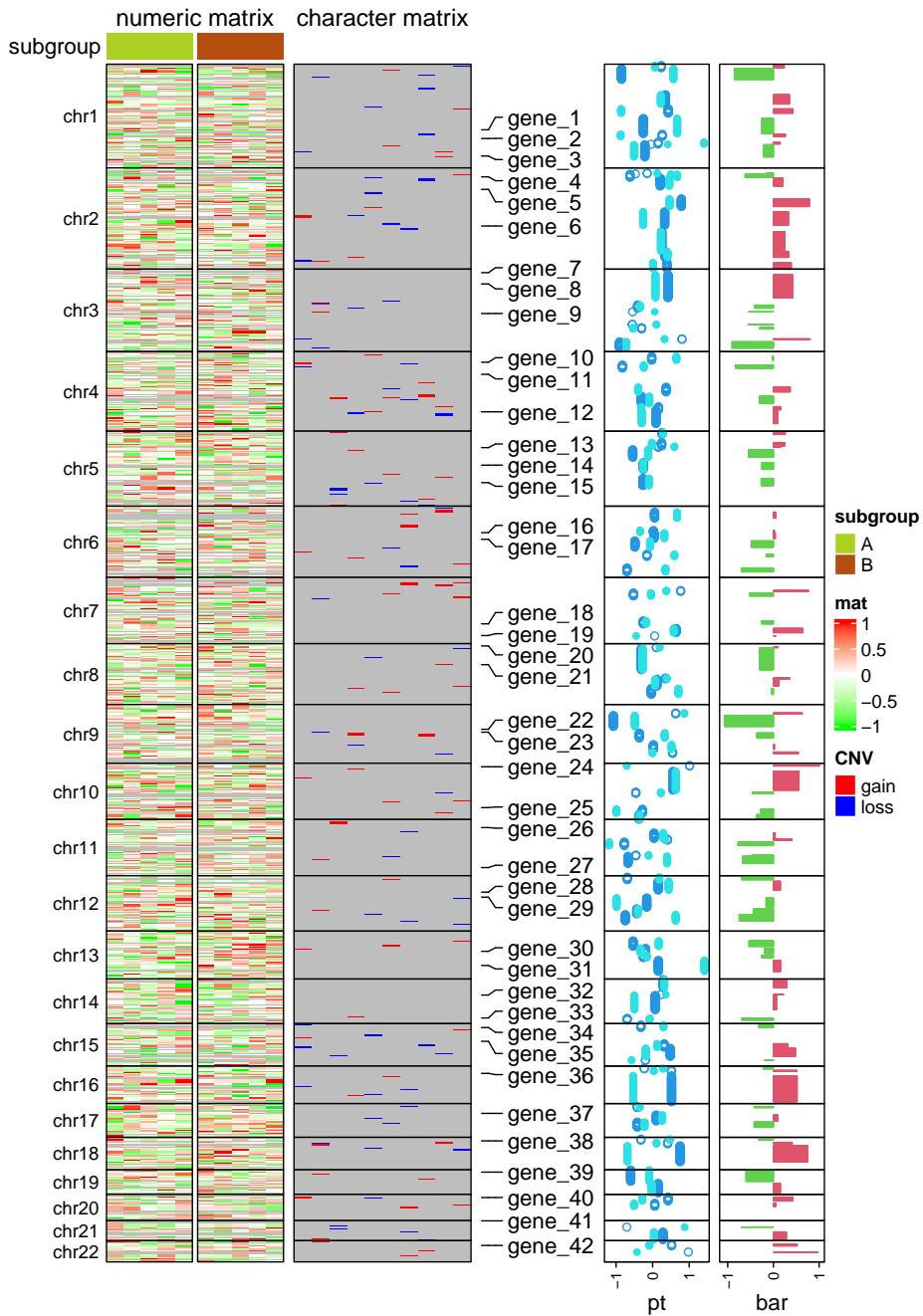
simulate the 10 columns in the matrix for two subgroups.

```
chr = as.vector(seqnames(chr_window))
chr_level = paste0("chr", 1:22)
chr = factor(chr, levels = chr_level)

subgroup = rep(c("A", "B"), each = 5)
```

The following code makes the heatmap with additional tracks. The plot is a combination of two heatmaps and three row annotations. Don't be scared by the massive number of arguments. If you have been using **ComplexHeatmap** for more than a week, I believe you've already get used to it :).

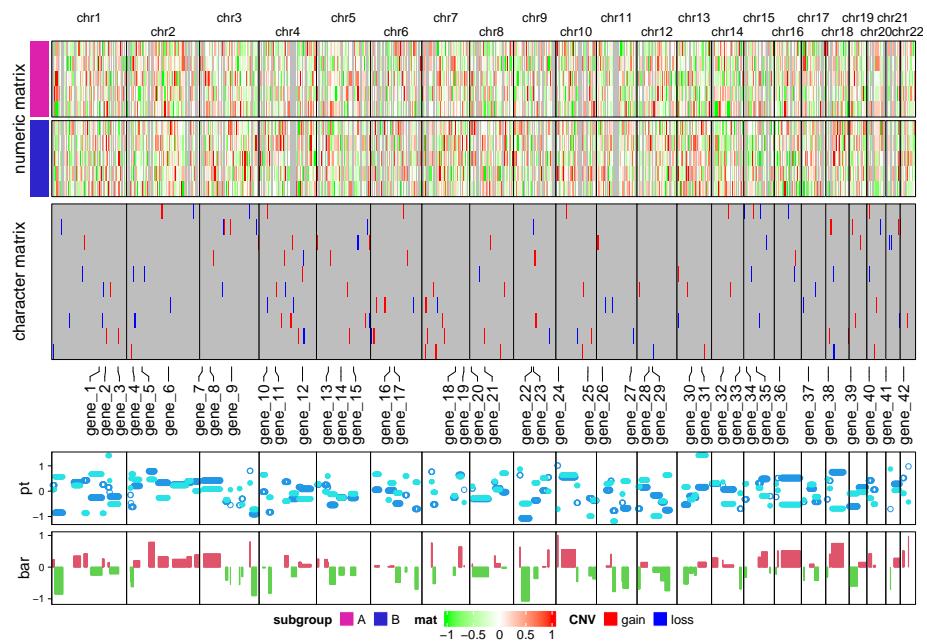
```
library(ComplexHeatmap)
ht_opt$TITLE_PADDING = unit(c(4, 4), "points")
ht_list = Heatmap(num_mat, name = "mat", col = colorRamp2(c(-1, 0, 1), c("green", "white")),
  row_split = chr, cluster_rows = FALSE, show_column_dend = FALSE,
  column_split = subgroup, cluster_column_slices = FALSE,
  column_title = "numeric matrix",
  top_annotation = HeatmapAnnotation(subgroup = subgroup, annotation_name_side = "left",
    row_title_rot = 0, row_title_gp = gpar(fontsize = 10), border = TRUE,
    row_gap = unit(0, "points")) +
  Heatmap(char_mat, name = "CNV", col = c("gain" = "red", "loss" = "blue"),
    border = TRUE, column_title = "character matrix") +
  rowAnnotation(label = anno_mark(at = at, labels = labels)) +
  rowAnnotation(pt = anno_points(v, gp = gpar(col = 4:5), pch = c(1, 16)),
    width = unit(2, "cm")) +
  rowAnnotation(bar = anno_barplot(v[, 1], gp = gpar(col = ifelse(v[, 1] > 0, 2, 3))),
    width = unit(2, "cm"))
draw(ht_list, merge_legend = TRUE)
```



It is easy to make the arrangement of heatmaps vertical (use `%v%` to concatenate heatmaps!). Just carefully switch the row-related parameters and column-related parameters. Here I additionally adjust the legends to make them look nicer in the plot.

Note I use a trick to arrange the chromosome names. Since the chromosome names will overlap for small chromosomes, I simply add \n before or after for the neighbour chromosome names (see how I set `column_title` argument in the first heatmap).

```
ht_list = Heatmap(t(num_mat), name = "mat", col = colorRamp2(c(-1, 0, 1), c("green", "red")),
  column_split = chr, cluster_columns = FALSE, show_row_dend = FALSE,
  row_split = subgroup, cluster_row_slices = FALSE,
  row_title = "numeric matrix",
  left_annotation = rowAnnotation(subgroup = subgroup, show_annotation_name = FALSE,
    annotation_legend_param = list(
      subgroup = list(direction = "horizontal", title_position = "lefttop", nrow = 1),
      column_title_gp = gpar(fontsize = 10), border = TRUE,
      column_gap = unit(0, "points"),
      column_title = ifelse(1:22 %% 2 == 0, paste0("\n", chr_level), paste0(chr_level, "\n"))
    ),
    heatmap_legend_param = list(direction = "horizontal", title_position = "lefttop")))
Heatmap(t(char_mat), name = "CNV", col = c("gain" = "red", "loss" = "blue"),
  border = TRUE, row_title = "character matrix",
  heatmap_legend_param = list(direction = "horizontal", title_position = "lefttop", nrow = 1))
HeatmapAnnotation(label = anno_mark(at = at, labels = labels, side = "bottom")) %v%
HeatmapAnnotation(pt = anno_points(v, gp = gpar(col = 4:5), pch = c(1, 16)),
  annotation_name_side = "left", height = unit(2, "cm")) %v%
HeatmapAnnotation(bar = anno_barplot(v[, 1], gp = gpar(col = ifelse(v[, 1] > 0, 2, 3))),
  annotation_name_side = "left", height = unit(2, "cm"))
draw(ht_list, heatmap_legend_side = "bottom", merge_legend = TRUE)
```



Chapter 14

More Examples

14.1 Add more information for gene expression matrix

Heatmaps are very popular to visualize gene expression matrix. Rows in the matrix correspond to genes and more information on these genes can be attached after the expression heatmap.

In following example, the big heatmap visualizes relative expression for genes (expression for each gene is scaled). On the right we put the absolute expression level of genes as a single-column heatmap. The gene length and gene type (i.e. protein coding or lincRNA) are also put as heatmap annotations or heatmaps.

On the very left of the heatmaps, there are colored rectangles drawn by `anno_block()` to identify the five clusters from k-means clustering. On top of the “base mean” and “gene type” heatmaps, there are summary plots (barplots and boxplots) showing the statistics or distributions of the data points in the five clusters.

```
library(ComplexHeatmap)
library(circlize)

expr = readRDS(system.file(package = "ComplexHeatmap", "extdata", "gene_expression.rds"))
mat = as.matrix(expr[, grep("cell", colnames(expr))])
base_mean = rowMeans(mat)
mat_scaled = t(apply(mat, 1, scale))

type = gsub("s\\\\d+_", "", colnames(mat))
ha = HeatmapAnnotation(type = type, annotation_name_side = "left")

ht_list = Heatmap(mat_scaled, name = "expression", row_km = 5,
```

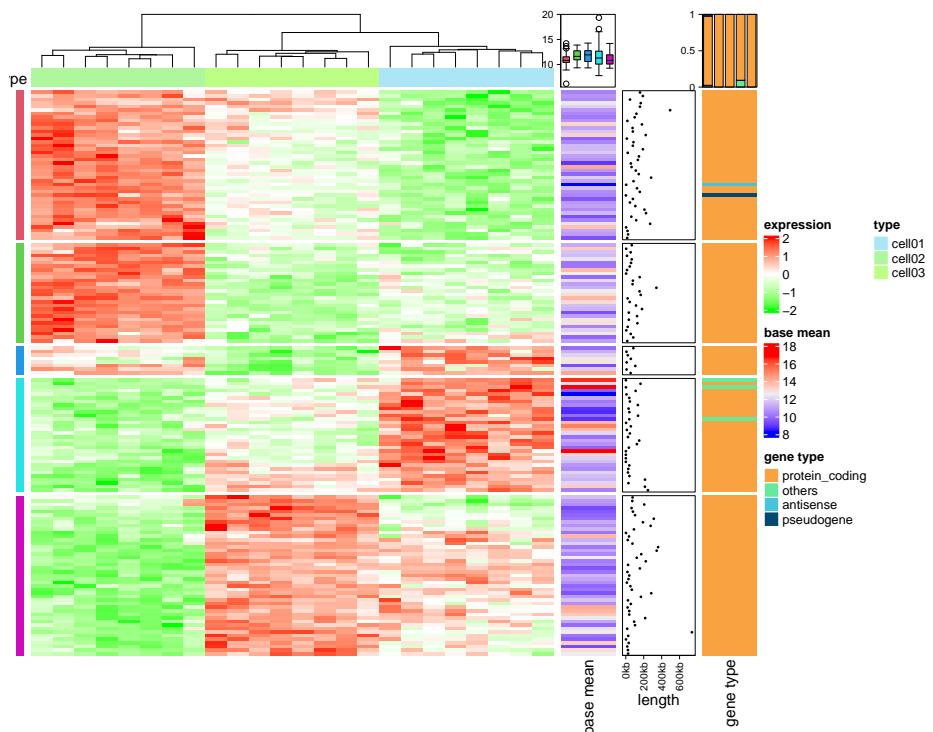
```

col = colorRamp2(c(-2, 0, 2), c("green", "white", "red")),
top_annotation = ha,
show_column_names = FALSE, row_title = NULL, show_row_dend = FALSE) +
Heatmap(base_mean, name = "base mean",
        top_annotation = HeatmapAnnotation(summary = anno_summary(gp = gpar(fill = 2:6),
                                                               height = unit(2, "cm"))),
        width = unit(15, "mm")) +
rowAnnotation(length = anno_points(expr$length, pch = 16, size = unit(1, "mm")),
              axis_param = list(at = c(0, 2e5, 4e5, 6e5),
                                 labels = c("0kb", "200kb", "400kb", "600kb")),
              width = unit(2, "cm")) +
Heatmap(expr$type, name = "gene type",
        top_annotation = HeatmapAnnotation(summary = anno_summary(height = unit(2, "cm"))),
        width = unit(15, "mm"))

ht_list = rowAnnotation(block = anno_block(gp = gpar(fill = 2:6, col = NA)),
                        width = unit(2, "mm")) + ht_list

draw(ht_list)

```



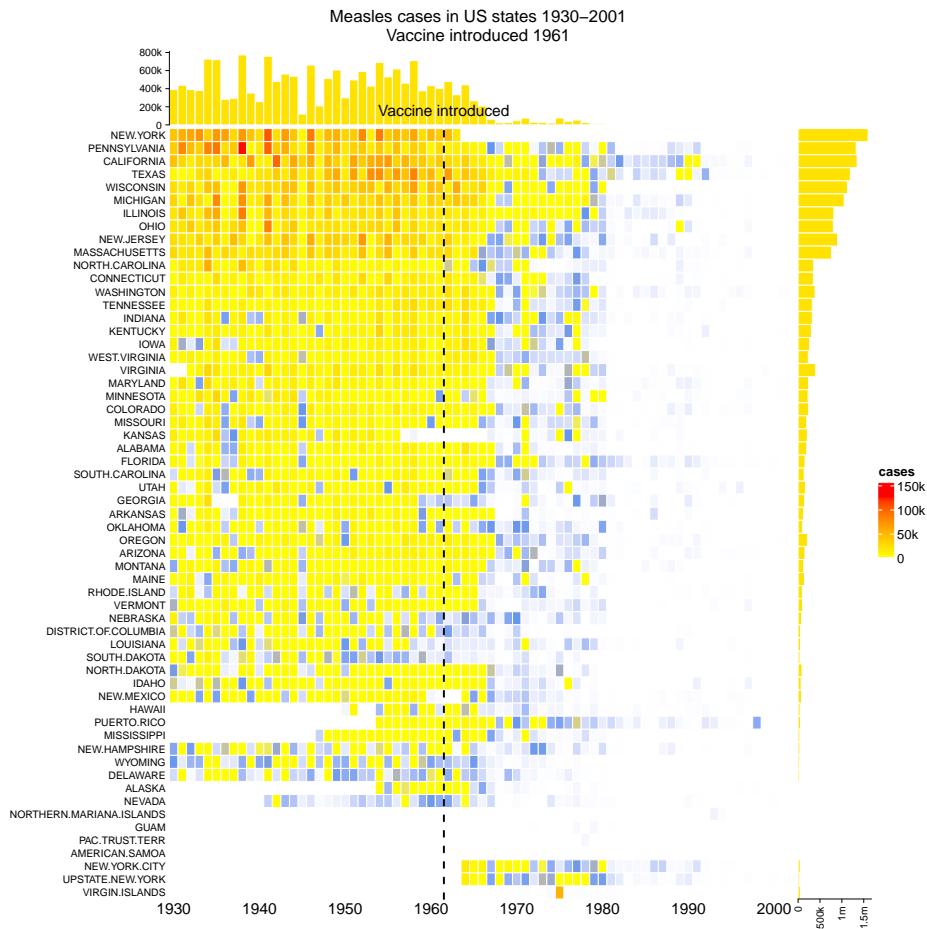
14.2 The measles vaccine heatmap

Following code reproduces the heatmap introduced here and here.

```

mat = readRDS(system.file("extdata", "measles.rds", package = "ComplexHeatmap"))
ha1 = HeatmapAnnotation(
  dist1 = anno_barplot(
    colSums(mat),
    bar_width = 1,
    gp = gpar(col = "white", fill = "#FFE200"),
    border = FALSE,
    axis_param = list(at = c(0, 2e5, 4e5, 6e5, 8e5),
                      labels = c("0", "200k", "400k", "600k", "800k")),
    height = unit(2, "cm")
  ), show_annotation_name = FALSE)
ha2 = rowAnnotation(
  dist2 = anno_barplot(
    rowSums(mat),
    bar_width = 1,
    gp = gpar(col = "white", fill = "#FFE200"),
    border = FALSE,
    axis_param = list(at = c(0, 5e5, 1e6, 1.5e6),
                      labels = c("0", "500k", "1m", "1.5m")),
    width = unit(2, "cm")
  ), show_annotation_name = FALSE)
year_text = as.numeric(colnames(mat))
year_text[year_text %% 10 != 0] = ""
ha_column = HeatmapAnnotation(
  year = anno_text(year_text, rot = 0, location = unit(1, "npc"), just = "top")
)
col_fun = colorRamp2(c(0, 800, 1000, 127000), c("white", "cornflowerblue", "yellow", "red"))
ht_list = Heatmap(mat, name = "cases", col = col_fun,
  cluster_columns = FALSE, show_row_dend = FALSE, rect_gp = gpar(col= "white"),
  show_column_names = FALSE,
  row_names_side = "left", row_names_gp = gpar(fontsize = 8),
  column_title = 'Measles cases in US states 1930-2001\nVaccine introduced 1961',
  top_annotation = ha1, bottom_annotation = ha_column,
  heatmap_legend_param = list(at = c(0, 5e4, 1e5, 1.5e5),
                               labels = c("0", "50k", "100k", "150k")) + ha2
draw(ht_list, ht_gap = unit(3, "mm"))
decorate_heatmap_body("cases", {
  i = which(colnames(mat) == "1961")
  x = i/ncol(mat)
  grid.lines(c(x, x), c(0, 1), gp = gpar(lwd = 2, lty = 2))
  grid.text("Vaccine introduced", x, unit(1, "npc") + unit(5, "mm"))
})
}

```



14.3 Visualize Cell Heterogeneity from Single Cell RNASeq

In this example, single cell RNA-Seq data for mouse T-cells is visualized to show the heterogeneity of cells. The data (`mouse_scRNAseq_corrected.txt`) is from Buettner et al., 2015, supplementary data 1, sheet “Cell-cycle corrected gene expr.” You can get `mouse_scRNAseq_corrected.txt` here.

In following code, duplicated genes are removed.

```
expr = read.table("data/mouse_scRNAseq_corrected.txt", sep = "\t", header = TRUE)
expr = expr[!duplicated(expr[[1]]), ]
rownames(expr) = expr[[1]]
expr = expr[-1]
```

14.3. VISUALIZE CELL HETEROGENEITY FROM SINGLE CELL RNASEQ467

```
expr = as.matrix(expr)
```

Genes that are not expressed in more than half of the cells are filtered out.

```
expr = expr[apply(expr, 1, function(x) sum(x > 0)/length(x) > 0.5), , drop = FALSE]
```

The `get_correlated_variable_rows()` function is defined here. It extracts signature genes that are variably expressed between cells and correlate to other genes.

```
get_correlated_variable_genes = function(mat, n = nrow(mat), cor_cutoff = 0, n_cutoff = 0) {
  ind = order(apply(mat, 1, function(x) {
    q = quantile(x, c(0.1, 0.9))
    x = x[x < q[1] & x > q[2]]
    var(x)/mean(x)
  }), decreasing = TRUE)[1:n]
  mat2 = mat[ind, , drop = FALSE]
  dt = cor(t(mat2), method = "spearman")
  diag(dt) = 0
  dt[abs(dt) < cor_cutoff] = 0
  dt[dt < 0] = -1
  dt[dt > 0] = 1

  i = colSums(abs(dt)) > n_cutoff

  mat3 = mat2[i, ,drop = FALSE]
  return(mat3)
}
```

Signature genes are defined as a list of genes where each gene correlates to more than 20 genes with an absolute correlation larger than 0.5.

`mat2` contains expression values scaled per gene, which means it contains relative expression across cells for every gene. Since single cell RNASeq data is highly variable and outliers are frequent, gene expression is only scaled within the 10th and 90th quantiles.

```
mat = get_correlated_variable_genes(expr, cor_cutoff = 0.5, n_cutoff = 20)
mat2 = t(apply(mat, 1, function(x) {
  q10 = quantile(x, 0.1)
  q90 = quantile(x, 0.9)
  x[x < q10] = q10
  x[x > q90] = q90
  scale(x)
}))
colnames(mat2) = colnames(mat)
```

Load cell cycle genes and ribonucleoprotein genes. The cell cycle gene list is

from Buettner et al., 2015, supplementary table 1, sheet “**Union of Cyclebase and GO genes**.” Ribonucleoprotein genes are from GO:0030529. Gene list are stored in `mouse_cell_cycle_gene.rds` and `mouse_ribonucleoprotein.rds`. The two files can be found [here](#) and [here](#).

```
cc = readRDS("data/mouse_cell_cycle_gene.rds")
ccl = rownames(mat) %in% cc
cc_gene = rownames(mat)[ccl]

rp = readRDS("data/mouse_ribonucleoprotein.rds")
rpl = rownames(mat) %in% rp
```

Since with scaling the expression values per gene the expression level of a gene relative to other genes has been lost, we calculate the base mean as the mean expression of a gene throughout all samples. The base mean can be used to compare expression levels between genes.

```
base_mean = rowMeans(mat)
```

Now the following information is available:

1. scaled expression, `mat2`,
2. base mean, `base_mean`,
3. whether genes are ribonucleoprotein genes, `rpl`,
4. whether genes are cell cycle genes, `ccl`,
5. symbols for cell cycle genes, `cc_gene`,

In the next step, we can put the information together and visualize it as a list of heatmaps. A gene-gene correlation heatmap is added at the end and defined to be the `main_heatmap`, meaning that the row order of all heatmaps/row annotations are based on the clustering of this correlation matrix.

For cell cycle genes with relatively high expression levels (larger than the 25% quantile of all genes), the gene name is indicated as text labels. In the first heatmap, the column dendrogram is underlaid with two different colours based in the two main groups derived by hierarchical clustering to highlight the two subpopulations.

```
library(GetoptLong)
ht_list = Heatmap(mat2, col = colorRamp2(c(-1.5, 0, 1.5), c("blue", "white", "red")),
  name = "scaled_expr", column_title = qq("relative expression for @{nrow(mat)} genes"),
  show_column_names = FALSE, width = unit(8, "cm"),
  heatmap_legend_param = list(title = "Scaled expr")) +
  Heatmap(base_mean, name = "base_expr", width = unit(5, "mm")),
  heatmap_legend_param = list(title = "Base expr")) +
  Heatmap(rpl + 0, name = "ribonucleoprotein", col = c("0" = "white", "1" = "purple"),
  show_heatmap_legend = FALSE, width = unit(5, "mm")) +
  Heatmap(ccl + 0, name = "cell_cycle", col = c("0" = "white", "1" = "red"),
  show_heatmap_legend = FALSE, width = unit(5, "mm")) +
```

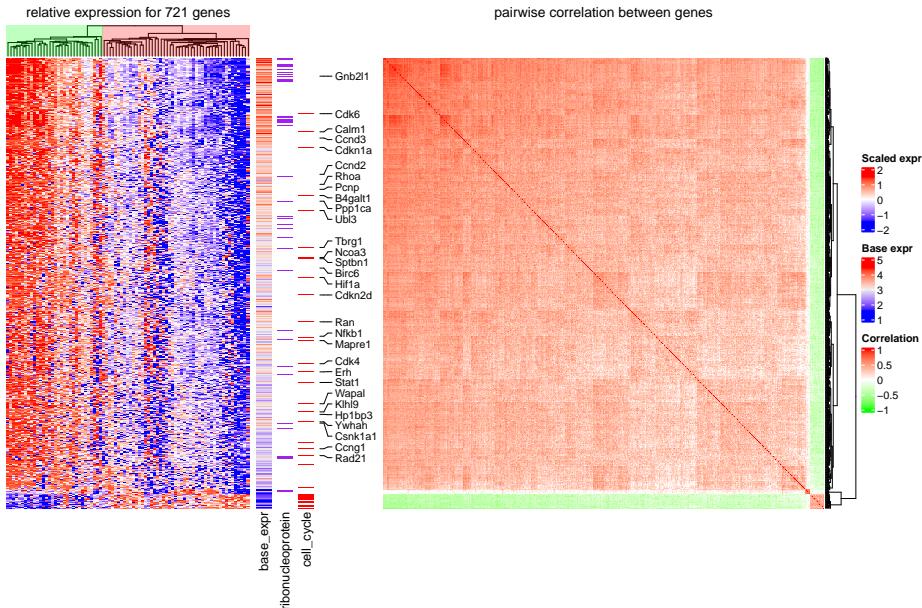
14.3. VISUALIZE CELL HETEROGENEITY FROM SINGLE CELL RNASEQ469

```

rowAnnotation(link = anno_mark(at = which(ccl & base_mean > quantile(base_mean, 0.25)),
  labels = rownames(mat)[ccl & base_mean > quantile(base_mean, 0.25)],
  labels_gp = gpar(fontsize = 10), padding = unit(1, "mm")) +
Heatmap(cor(t(mat2)), name = "cor",
  col = colorRamp2(c(-1, 0, 1), c("green", "white", "red")),
  show_row_names = FALSE, show_column_names = FALSE, row_dend_side = "right",
  show_column_dend = FALSE, column_title = "pairwise correlation between genes",
  heatmap_legend_param = list(title = "Correlation"))
ht_list = draw(ht_list, main_heatmap = "cor")
decorate_column_dend("scaled_expr", {
  tree = column_dend(ht_list)$scaled_expr
  ind = cutree(as.hclust(tree), k = 2)[order.dendrogram(tree)]
}

first_index = function(l) which(l)[1]
last_index = function(l) { x = which(l); x[length(x)] }
x1 = c(first_index(ind == 1), first_index(ind == 2)) - 1
x2 = c(last_index(ind == 1), last_index(ind == 2))
grid.rect(x = x1/length(ind), width = (x2 - x1)/length(ind), just = "left",
  default.units = "npc", gp = gpar(fill = c("#FF000040", "#00FF0040"), col = NA))
})

```



The heatmap clearly reveals that the cells are separated into two sub-populations. The population on the left in the first heatmap exhibits high expression of a subset of cell cycle genes (cell cycle genes are indicated in “`cell_cycle`” heatmap). However, the overall expression level for these genes is relatively low (see “`base_expr`” heatmap). The population on the right has higher

expression in the other signature genes. Interestingly, the signature genes which are higher expressed in this subpopulation are enriched for genes coding for ribonucleoproteins (see “**ribonucleoprotein**” heatmap). A subset of the ribonucleoprotein genes shows strong coexpression (see correlation heatmap) and overall high expression levels (“**base_expr**” heatmap).

14.4 Correlations between methylation, expression and other genomic features

In the following example, data is randomly generated based on patterns found in an unpublished analysis. First we load the data. `meth.rds` can be found here.

```
res_list = readRDS("data/meth.rds")
type = res_list$type
mat_meth = res_list$mat_meth
mat_expr = res_list$mat_expr
direction = res_list$direction
cor_pvalue = res_list$cor_pvalue
gene_type = res_list$gene_type
anno_gene = res_list$anno_gene
dist = res_list$dist
anno_enhancer = res_list$anno_enhancer
```

The different sources of information and corresponding variables are:

1. `type`: the label which shows whether the sample is tumor or normal.
2. `mat_meth`: a matrix in which rows correspond to differentially methylated regions (DMRs). The value in the matrix is the mean methylation level in the DMR in every sample.
3. `mat_expr`: a matrix in which rows correspond to genes which are associated to the DMRs (i.e. the nearest gene to the DMR). The value in the matrix is the expression level for each gene in each sample. Expression is scaled for every gene across samples.
4. `direction`: direction of the methylation change (hyper meaning higher methylation in tumor samples, hypo means lower methylation in tumor samples).
5. `cor_pvalue`: p-value for the correlation test between methylation and expression of the associated gene.
6. `gene_type`: type of the genes (e.g. protein coding genes or lincRNAs).
7. `anno_gene`: annotation to the gene models (intergenic, intragenic or TSS).
8. `dist`: distance from DMRs to TSS of the associated genes.
9. `anno_enhancer`: fraction of the DMR that overlaps enhancers.

The data only includes DMRs for which methylation and expression of the associated gene are negatively correlated.

The clustering of columns for the methylation matrix are calculated first so that

columns in the expression matrix can be adjusted to have the same column order as in the methylation matrix.

```
column_tree = hclust(dist(t(mat_meth)))
column_order = column_tree$order

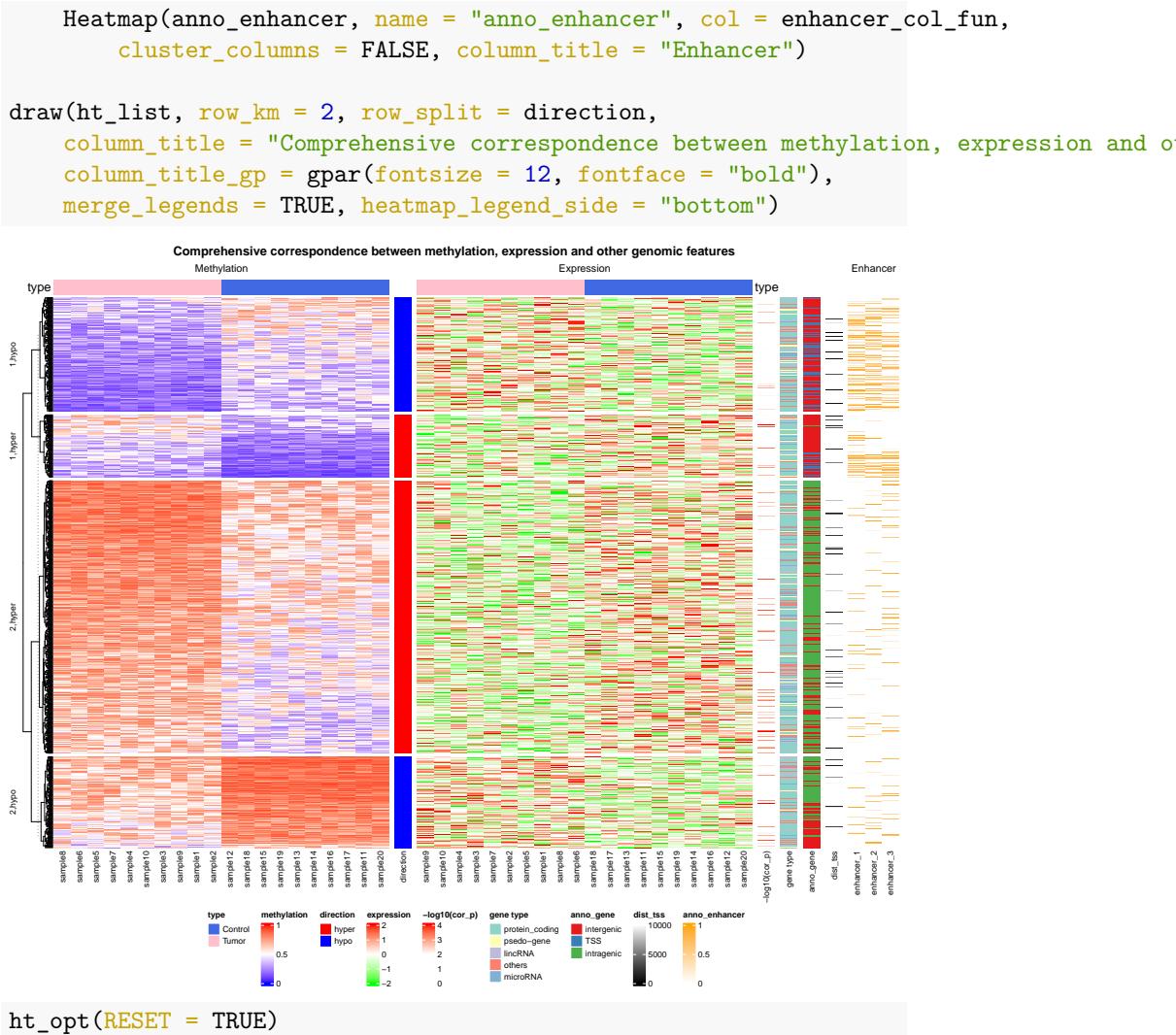
library(RColorBrewer)
meth_col_fun = colorRamp2(c(0, 0.5, 1), c("blue", "white", "red"))
direction_col = c("hyper" = "red", "hypo" = "blue")
expr_col_fun = colorRamp2(c(-2, 0, 2), c("green", "white", "red"))
pvalue_col_fun = colorRamp2(c(0, 2, 4), c("white", "white", "red"))
gene_type_col = structure(brewer.pal(length(unique(gene_type)), "Set3"),
                           names = unique(gene_type))
anno_gene_col = structure(brewer.pal(length(unique(annotation)), "Set1"),
                           names = unique(annotation))
dist_col_fun = colorRamp2(c(0, 10000), c("black", "white"))
enhancer_col_fun = colorRamp2(c(0, 1), c("white", "orange"))
```

We first define two column annotations and then make the complex heatmaps.

```
ht_opt(
  legend_title_gp = gpar(fontsize = 8, fontface = "bold"),
  legend_labels_gp = gpar(fontsize = 8),
  heatmap_column_names_gp = gpar(fontsize = 8),
  heatmap_column_title_gp = gpar(fontsize = 10),
  heatmap_row_title_gp = gpar(fontsize = 8)
)

ha = HeatmapAnnotation(type = type,
                       col = list(type = c("Tumor" = "pink", "Control" = "royalblue")),
                       annotation_name_side = "left")
ha2 = HeatmapAnnotation(type = type,
                        col = list(type = c("Tumor" = "pink", "Control" = "royalblue")),
                        show_legend = FALSE)

ht_list = Heatmap(mat_meth, name = "methylation", col = meth_col_fun,
                  column_order= column_order,
                  top_annotation = ha, column_title = "Methylation") +
  Heatmap(direction, name = "direction", col = direction_col) +
  Heatmap(mat_expr[, column_tree$order], name = "expression",
          col = expr_col_fun,
          column_order = column_order,
          top_annotation = ha2, column_title = "Expression") +
  Heatmap(cor_pvalue, name = "-log10(cor_p)", col = pvalue_col_fun) +
  Heatmap(gene_type, name = "gene type", col = gene_type_col) +
  Heatmap(annotation, name = "annotation", col = anno_gene_col) +
  Heatmap(dist, name = "dist_tss", col = dist_col_fun) +
```



The complex heatmaps reveal that highly methylated DMRs are enriched in intergenic and intragenic regions and rarely overlap with enhancers. In contrast, lowly methylated DMRs are enriched for transcription start sites (TSS) and enhancers.

14.5 Visualize Methylation Profile with Complex Annotations

In this example, Figure 1 in Strum et al., 2012 is re-implemented with some adjustments.

Some packages need to be loaded firstly.

```
library(matrixStats)
library(GenomicRanges)
```

Methylation profiles can be download from GEO database. The **GEOquery** package is used to retrieve data from GEO.

```
library(GEOquery)
gset = getGEO("GSE36278")
```

The methylation profiles have been measured by Illumina HumanMethylation450 BeadChip arrays. We load probe data via the **IlluminaHumanMethylation450kanno.ilmn12.hg19** package.

Adjust row names in the matrix to be the same as the probes.

```
library(IlluminaHumanMethylation450kanno.ilmn12.hg19)
data(Locations)

mat = exprs(gset[[1]])
colnames(mat) = phenoData(gset[[1]])@data$title
mat = mat[rownames(Locations), ]
```

`probe` contains locations of probes and also information whether the CpG sites overlap with SNPs. Here we remove probes that are on sex chromosomes and probes that overlap with SNPs.

```
data(SNPs.137CommonSingle)
data(Islands.UCSC)
l = Locations$chr %in% paste0("chr", 1:22) & is.na(SNPs.137CommonSingle$Probe_rs)
mat = mat[l, ]
```

Get subsets for locations of probes and the annotation to CpG Islands accordingly.

```
cgi = Islands.UCSC$Relation_to_Island[1]
loc = Locations[l, ]
```

Separate the matrix into a matrix for tumor samples and a matrix for normal samples. Also modify column names for the tumor samples to be consistent with the phenotype data which we will read later.

```
mat1 = as.matrix(mat[, grep("GBM", colnames(mat))]) # tumor samples
mat2 = as.matrix(mat[, grep("CTRL", colnames(mat))]) # normal samples
colnames(mat1) = gsub("GBM", "dkfz", colnames(mat1))
```

Phenotype data is from Sturm et al., 2012, supplementary table S1 and can be found [here](#).

The rows of phenotype data are adjusted to be the same as the columns of the methylation matrix.

```

phenotype = read.table("data/450K_annotation.txt", header = TRUE, sep = "\t",
  row.names = 1, check.names = FALSE, comment.char = "", stringsAsFactors = FALSE)
phenotype = phenotype[colnames(mat1), ]

```

Please note that we only use the 136 samples which are from DKFZ, while in Sturm et al., 2012, additional 74 TCGA samples have been used.

Extract the top 8000 probes with most variable methylation in the tumor samples, and also subset other information correspondingly.

```

ind = order(rowVars(mat1, na.rm = TRUE), decreasing = TRUE)[1:8000]
m1 = mat1[ind, ]
m2 = mat2[ind, ]
cgi2 = cgi[ind]
cgi2 = ifelse(grepl("Shore", cgi2), "Shore", cgi2)
cgi2 = ifelse(grepl("Shelf", cgi2), "Shelf", cgi2)
loc = loc[ind, ]

```

For each probe, find the distance to the closest TSS. pc_tx_tss.bed contains positions of TSS from protein coding genes.

```

gr = GRanges(loc[, 1], ranges = IRanges(loc[, 2], loc[, 2]+1))
tss = read.table("data/pc_tx_tss.bed", stringsAsFactors = FALSE)
tss = GRanges(tss[[1]], ranges = IRanges(tss[, 2], tss[, 3]))

tss_dist = distanceToNearest(gr, tss)
tss_dist = tss_dist@elementMetadata$distance

```

Because there are a few NA in the matrix (`sum(is.na(m1))/length(m1) = 0.0011967`) which will break the `cor()` function, we replace NA to the intermediate methylation (0.5). Note that although **ComplexHeatmap** allows NA in the matrix, removal of NA will speed up the clustering.

```

m1[is.na(m1)] = 0.5
m2[is.na(m2)] = 0.5

```

The following annotations will be added to the columns of the methylation matrix:

1. age
2. subtype classification by DKFZ
3. subtype classification by TCGA
4. subtype classification by TCGA, based on expression profile
5. IDH1 mutation
6. H3F3A mutation
7. TP53 mutation
8. chr7 gain
9. chr10 loss
10. CDKN2A deletion

11. EGFR amplification
12. PDGFRA amplification

In following code we define the column annotation in the `ha` variable. Also we customize colors, legends and height of the annotations.

```
mutation_col = structure(names = c("MUT", "WT", "G34R", "G34V", "K27M"),
                        c("black", "white", "#4DAF4A", "#4DAF4A", "#377EB8"))
cnv_col = c("gain" = "#E41A1C", "loss" = "#377EB8", "amp" = "#E41A1C",
           "del" = "#377EB8", "normal" = "white")
ha = HeatmapAnnotation(
  age = anno_points(phenotype[[13]]),
  gp = gpar(col = ifelse(phenotype[[13]] > 20, "black", "red")),
  height = unit(3, "cm")),
  dkfz_cluster = phenotype[[1]],
  tcga_cluster = phenotype[[2]],
  tcga_expr = phenotype[[3]],
  IDH1 = phenotype[[5]],
  H3F3A = phenotype[[4]],
  TP53 = phenotype[[6]],
  chr7_gain = ifelse(phenotype[[7]] == 1, "gain", "normal"),
  chr10_loss = ifelse(phenotype[[8]] == 1, "loss", "normal"),
  CDKN2A_del = ifelse(phenotype[[9]] == 1, "del", "normal"),
  EGFR_amp = ifelse(phenotype[[10]] == 1, "amp", "normal"),
  PDGFRA_amp = ifelse(phenotype[[11]] == 1, "amp", "normal"),
  col = list(dkfz_cluster = structure(names = c("IDH", "K27", "G34", "RTK I PDGFRA",
                                                "Mesenchymal", "RTK II Classic"), brewer.pal(6, "Set1")),
             tcga_cluster = structure(names = c("G-CIMP+", "Cluster #2", "Cluster #3"),
                                       brewer.pal(3, "Set1")),
             tcga_expr = structure(names = c("Proneural", "Classical", "Mesenchymal"),
                                   c("#377EB8", "#FFFF33", "#FF7F00"))),
  IDH1 = mutation_col,
  H3F3A = mutation_col,
  TP53 = mutation_col,
  chr7_gain = cnv_col,
  chr10_loss = cnv_col,
  CDKN2A_del = cnv_col,
  EGFR_amp = cnv_col,
  PDGFRA_amp = cnv_col),
  na_col = "grey", border = TRUE,
  show_legend = c(TRUE, TRUE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, FALSE, FALSE),
  show_annotation_name = FALSE,
  annotation_legend_param = list(
    dkfz_cluster = list(title = "DKFZ Methylation"),
    tcga_cluster = list(title = "TCGA Methylation"),
    tcga_expr = list(title = "TCGA Expression"))
```

```

    H3F3A = list(title = "Mutations"),
    chr7_gain = list(title = "CNV"))
)

```

In the final plot, there are four heatmaps added. From left to right, there are

1. heatmap for methylation in tumor samples
2. methylation in normal samples
3. distance to nearest TSS
4. CpG Island (CGI) annotation.

The heatmaps are split by rows according to CGI annotations.

After the heatmaps are plotted, additional graphics such as labels for annotations are added by `decorate_*`() functions.

```

col_fun = colorRamp2(c(0, 0.5, 1), c("#377EB8", "white", "#E41A1C"))
ht_list = Heatmap(m1, col = col_fun, name = "Methylation",
  clustering_distance_columns = "spearmann",
  show_row_dend = FALSE, show_column_dend = FALSE,
  show_column_names = FALSE,
  bottom_annotation = ha, column_title = qq("GBM samples (n = @{ncol(m1)}"),
  row_split = factor(cgi2, levels = c("Island", "Shore", "Shelf", "OpenSea")),
  row_title_gp = gpar(col = "#FFFFFF00")) +
Heatmap(m2, col = col_fun, show_column_names = FALSE,
  show_column_dend = FALSE, column_title = "Controls",
  show_heatmap_legend = FALSE, width = unit(1, "cm")) +
Heatmap(tss_dist, name = "tss_dist", col = colorRamp2(c(0, 2e5), c("white", "black")),
  width = unit(5, "mm")),
  heatmap_legend_param = list(at = c(0, 1e5, 2e5), labels = c("0kb", "100kb", "200kb"))
Heatmap(cgi2, name = "CGI", show_row_names = FALSE, width = unit(5, "mm"),
  col = structure(names = c("Island", "Shore", "Shelf", "OpenSea"), c("red", "blue")),
draw(ht_list, row_title = paste0("DNA methylation probes (n = ", nrow(m1), ")"),
  annotation_legend_side = "left", heatmap_legend_side = "left"))

annotation_titles = c(dkfz_cluster = "DKFZ Methylation",
  tcga_cluster = "TCGA Methylation",
  tcga_expr = "TCGA Expression",
  IDH1 = "IDH1",
  H3F3A = "H3F3A",
  TP53 = "TP53",
  chr7_gain = "Chr7 gain",
  chr10_loss = "Chr10 loss",
  CDKN2A_del = "Chr10 loss",
  EGFR_amp = "EGFR amp",
  PDGFRA_amp = "PDGFRA amp")
for(an in names(annotation_titles)) {

```

```

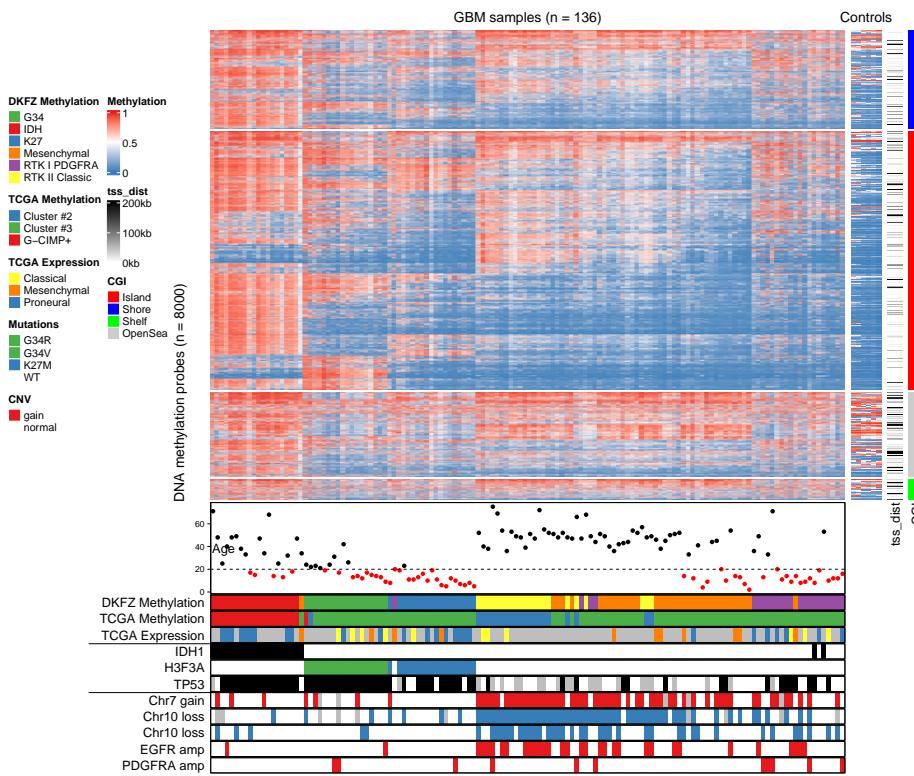
decorate_annotation(an, {
    grid.text(annotation_titles[an], unit(-2, "mm"), just = "right")
    grid.rect(gp = gpar(fill = NA, col = "black"))
})
}

decorate_annotation("age", {
    grid.text("Age", unit(8, "mm"), just = "right")
    grid.rect(gp = gpar(fill = NA, col = "black"))
    grid.lines(unit(c(0, 1), "npc"), unit(c(20, 20), "native"), gp = gpar(lty = 2))
})
}

decorate_annotation("IDH1", {
    grid.lines(unit(c(-40, 0), "mm"), unit(c(1, 1), "npc"))
})

decorate_annotation("chr7_gain", {
    grid.lines(unit(c(-40, 0), "mm"), unit(c(1, 1), "npc"))
})
}

```



14.6 Add multiple boxplots for single row

The annotation function `anno_boxplot()` only draws one boxplot for a single row. When multiple heatmaps are concatenated, or the groups for columns have already been defined, for each row, we want to compare between heatmaps or column groups, thus, multiple boxplots need to be drawn for each single row.

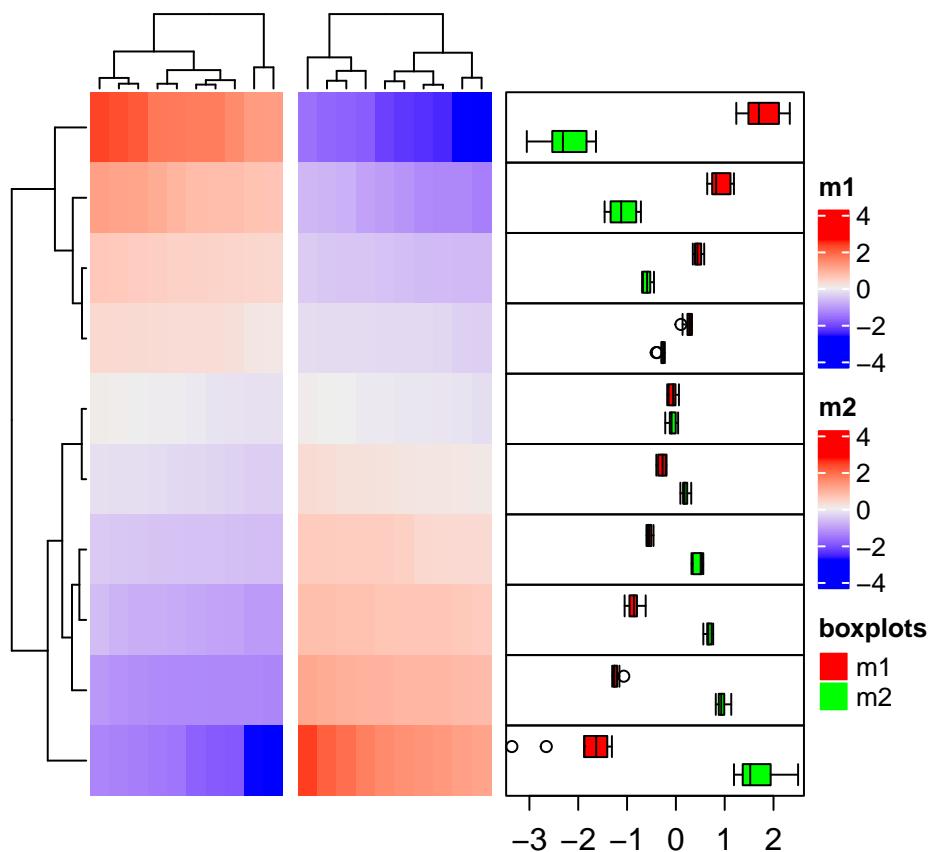
In following example, we demonstrate how to implement an annotation function which draws multiple boxplots for single rows. The `grid.boxplot()` function is from **ComplexHeatmap** package which makes it easy to draw boxplot under `grid` system.

```
m1 = matrix(sort(rnorm(100)), 10, byrow = TRUE)
m2 = matrix(sort(rnorm(100), decreasing = TRUE), 10, byrow = TRUE)

ht_list = Heatmap(m1, name = "m1") + Heatmap(m2, name = "m2")

rg = range(c(m1, m2))
rg[1] = rg[1] - (rg[2] - rg[1])* 0.02
rg[2] = rg[2] + (rg[2] - rg[1])* 0.02
anno_multiple_boxplot = function(index) {
    nr = length(index)
    pushViewport(viewport(xscale = rg, yscale = c(0.5, nr + 0.5)))
    for(i in seq_along(index)) {
        grid.rect(y = nr-i+1, height = 1, default.units = "native")
        grid.boxplot(m1[ index[i], ], pos = nr-i+1 + 0.2, box_width = 0.3,
                     gp = gpar(fill = "red"), direction = "horizontal")
        grid.boxplot(m2[ index[i], ], pos = nr-i+1 - 0.2, box_width = 0.3,
                     gp = gpar(fill = "green"), direction = "horizontal")
    }
    grid.xaxis()
    popViewport()
}

ht_list = ht_list + rowAnnotation(boxplot = anno_multiple_boxplot, width = unit(4, "cm",
    show_annotation_name = FALSE)
lgd = Legend(labels = c("m1", "m2"), title = "boxplots",
             legend_gp = gpar(fill = c("red", "green")))
draw(ht_list, padding = unit(c(20, 2, 2, 2), "mm"), heatmap_legend_list = list(lgd))
```



Chapter 15

Other Tricks

15.1 Set the same cell size for different heatmaps with different dimensions

Assume you have a list of heatmaps/oncoPrints that you want to save as different e.g. png or pdf files, one thing you might want to do is to make the size of each grid/cell in the heatmap identical across heatmaps, thus, you need to calculate the size of png/pdf file according to the number of rows or columns in the heatmap. In the heatmap generated by **ComplexHeatmap**, all the heatmap components have absolute size and only the size of the heatmap body (or the size of the cells) is changable (or in other words, if you change the size of the final graphic device, e.g. by dragging the graphics window if you plot in, only the size of the heatmap body is adjusted), which means, the size of the whole plot is linearly related to the number of rows or columns in the heatmap. This implies we can actually fit a linear model $y = a*x + b$ where e.g. y is the height of the whole plot and x is the number of rows.

In following example, we simply demonstrate how to establish the relation between the plot height and the number of rows in the heatmap. We first define a function which generates a 10-column matrix with specific number of rows. Note the values in the matrix is of no importance in this demonstration.

```
random_mat = function(nr) {  
  m = matrix(rnorm(10*nr), nc = 10)  
  colnames(m) = letters[1:10]  
  return(m)  
}
```

Since the relation is absolutely linear, we only need to test two heatmaps with different number of rows where the height of a single row is `unit(5, "mm")`. In the heatmap, there are also column title, column dendrogram, column annotation

and the column names.

There are several things that needs to be noted in following code:

1. The heatmap object should be returned by `draw()` because the layout of the heatmap is calculated only after the execution of `draw()`.
2. `component_height()` returns a vector of units which correspond to the height of all heatmap components from top to bottom in the heatmap. (`component_width()` returns the width of heatmap components).
3. When calculating `ht_height`, we add `unit(4, "mm")` because on top and bottom of the final plot, there are 2mm white borders.
4. `ht_height` needs to be converted to a simple unit in `cm` or `inch`.

In following, `y` contains values which are measured in `inch` unit.

```
y = NULL
for(nr in c(10, 20)) {
  ht = draw(Heatmap(random_mat(nr), height = unit(5, "mm")*nr,
    column_title = "foo", # one line text
    top_annotation = HeatmapAnnotation(bar = 1:10)))
  ht_height = sum(component_height(ht)) + unit(4, "mm")
  ht_height = convertHeight(ht_height, "inch", valueOnly = TRUE)
  y = c(y, ht_height)
}
```

Then we can fit a linear relation between `y` and the number of rows:

```
x = c(10, 20)
lm(y ~ x)
```

```
##
## Call:
## lm(formula = y ~ x)
##
## Coefficients:
## (Intercept)          x
##       1.2222      0.1969
```

This means the relation between the number of rows `x` and the height of the plot `y` is: $y = 0.1969*x + 1.3150$.

You can test whether the height of single rows are the same for heatmaps with different rows by following code. Note all the heatmap configurations should be the same as the ones you prepare `y`.

```
for(nr in c(10, 20)) {
  png(paste0("test_heatmap_nr_", nr, ".png"), width = 5, height = 0.1969*nr + 1.3150
    units = "in", res = 100)
  draw(Heatmap(random_mat(nr), height = unit(5, "mm")*nr,
    column_title = "foo", # column title can be any one-line string
```

```
    top_annotation = HeatmapAnnotation(bar = 1:10)))
dev.off()
}
```

