ggalign: Bridging the Grammar of Graphics and Complex layout

Yun Peng

2024-11-24

Table of contents

6   quad-layout   **96**

7   A list of `quad_layout()`   **107**

Preface

Welcome to ggalign documents. Examples in the book are generated under version 0.0.5.9000.

In the world of data visualization, aligning multiple plots in a coherent and organized layout is often a challenging task, especially when dealing with complex datasets that require precise alignment across rows, columns, and even within plot elements. While existing tools provide some solutions, they often fall short in offering the flexibility, control, and simplicity that users need to create intricate and beautiful plots. This is where `ggalign` comes in.

The `ggalign` package, built on top of the powerful ggplot2 framework, is designed to solve this very problem. It offers a suite of functions specifically crafted for aligning and organizing plots with minimal effort. Whether you need to align observations based on statistical measures, group plots by categorical factors, or fine-tune the layout to match the precise needs of your data, `ggalign` gives you the tools you need to create polished, publication-ready visualizations.

This book serves as both an introduction to the `ggalign` package and a comprehensive guide to mastering its features. Whether you're a beginner or an experienced user of ggplot2, you'll find detailed explanations, step-by-step tutorials, and real-world examples to help you leverage the full potential of `ggalign` in your work.

Throughout this book, we will cover everything from basic concepts to advanced layout customizations, focusing on key functions like `stack_layout()`, `align_*` series (including `align_group()`, `align_order()`, and `align_hclust()`), and how to combine them with other ggplot2 layers to create aligned plots. Additionally, you'll learn how to adapt `ggalign` for different data types and scenarios, allowing you to develop flexible, dynamic visualizations tailored to your specific needs.

By the end of this book, you will be equipped to use `ggalign` effectively in your own projects, whether for scientific research, data analysis, or any other field where data visualization is key. Our goal is to provide you with the knowledge and confidence to tackle complex visualization challenges and transform your datasets into clear, impactful, and visually appealing plots.

Thank you for choosing `ggalign`. We hope this book will inspire you to explore the endless possibilities that come with aligned data visualization.

1 Introduction

ggalign extends **ggplot2** by providing advanced tools for aligning and organizing multiple plots, particularly those that automatically reorder observations, such as dendrogram. It offers fine control over layout adjustment and plot annotations, enabling you to create complex visualizations while leveraging the familiar grammar of **ggplot2**.

1.1 Installation

You can install **ggalign** from **CRAN** using:

```
install.packages("ggalign")
```

Alternatively, install the development version from r-universe with:

```
install.packages("ggalign",
    repos = c("https://yunuuuu.r-universe.dev", "https://cloud.r-project.org")
)
```

or from GitHub with:

```
# install.packages("remotes")
remotes::install_github("Yunuuuu/ggalign")
```

1.2 General design

The core feature of **ggalign** lies in its integration of the **grammar of graphics** into advanced visualization through its object-oriented **Layout** system. The package provides two main **Layout** classes:

- the **StackLayout** class: Put plots horizontally or vertically.
- the **QuadLayout** class: Arranges plots in the four quadrants (top, left, bottom, right) around a main plot. This layout is ideal for designs that require supplementary plots or annotations surrounding a central figure.

Both `Layout` classes support the alignment of observations (ordinal variable). Observations refer to data points or samples, allowing for consistent alignment of corresponding data across multiple plots when using the same axis values. Depending on whether you want to align observations across multiple plots within the layout, the following variants are available:

For `StackLayout`:

- `stack_align()`: Align the observations along the stack.
- `stack_free()`: Does not align the observations.

For `QuadLayout`:

- `quad_free`/`ggside`: Never align observations.
- `quad_alignh`: Align observations in the horizontal direction.
- `quad_alignv`: Align observations in the vertical direction.
- `quad_alignb`: Align observations in both horizontal and vertical directions.
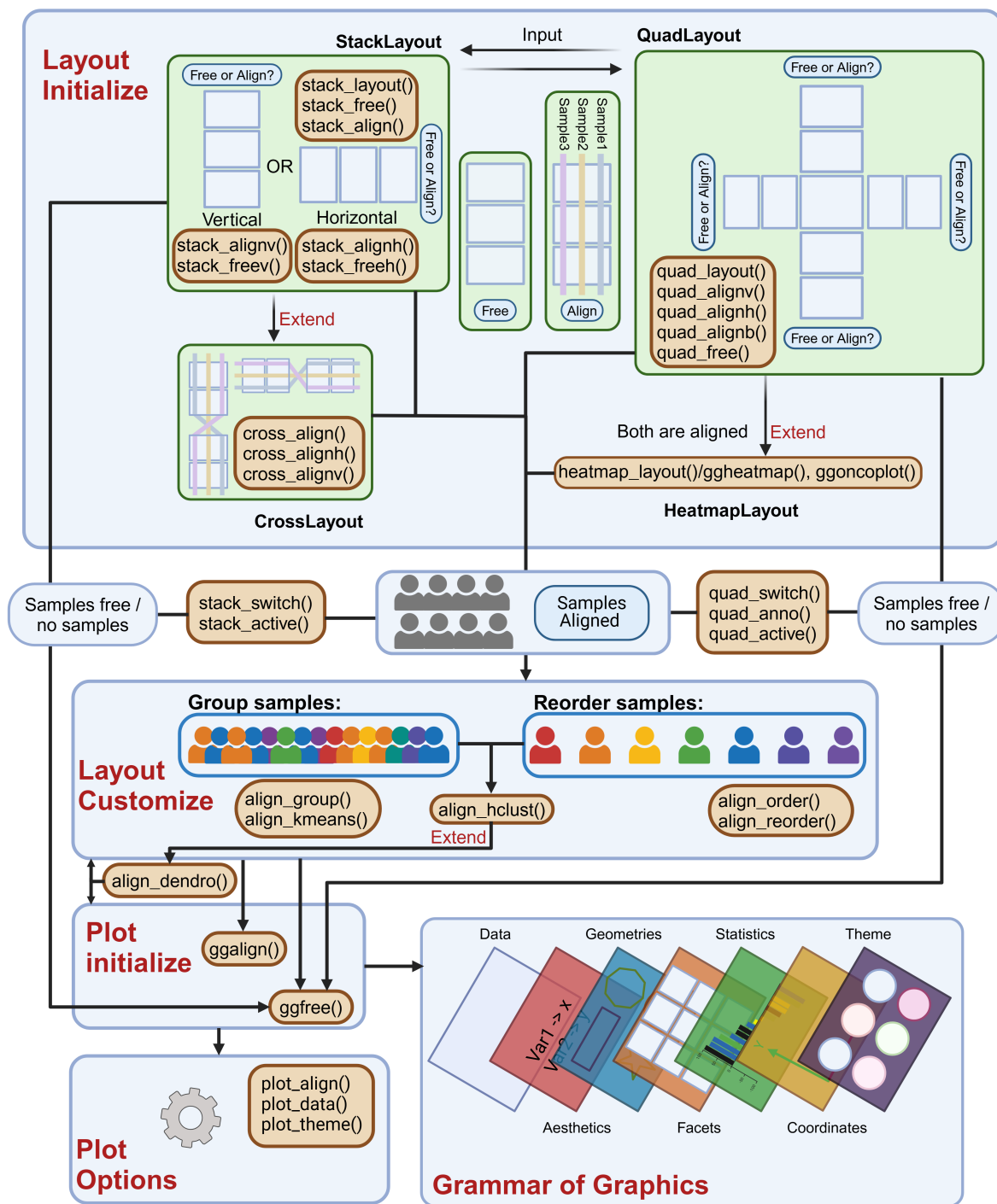
Figure 1.1: General design of ggalign

1.3 Getting Started

```
library(ggalign)
#> Loading required package: ggplot2
```

The usage of `ggalign` is simple if you're familiar with `ggplot2` syntax, the typical workflow includes:

1. Initialize the layout using:

   - `stack_layout()`: Arrange Plots Horizontally or Vertically

     - `cross_align`: Arrange Plots Crosswise Horizontally or Vertically

   - `quad_layout()`: Arrange Plots in the Quad-Side of a main plot

     - `ggheatmap()`: Create a Complex Heatmap.
     - `ggoncoplot()`: Create `OncoPrint` Visualizations from Genetic Alteration Data

2. Customize the layout with:

   - `align_group()`: Group observations into panel with a group variable.
   - `align_kmeans()`: Group observations into panel by kmeans.
   - `align_order()`: Reorder layout observations based on statistical weights or by manually specifying the observation index.
   - `align_hclust()`: Reorder or group observations based on hierarchical clustering.
   - `align_reorder`: Reorder observations using an arbitrary statistical function.

3. Adding plots with `align_dendro()`, `ggalign()` or `ggfree()`, and then layer additional ggplot2 elements such as geoms, stats, or scales.

```
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))
```

Every `*_layout()` function accepts default data, which will be inherited by all plots within the layout.

Here's a simple example:

```
stack_alignv(small_mat) +                                    ①
    align_dendro() +                                         ②
    theme(axis.text.y = element_text())                      ③
```

① We initialize a vertical stack.
② Reorder the observations based on hierarchical clustering and add a dendrogram tree.
③ Add y-axis text.
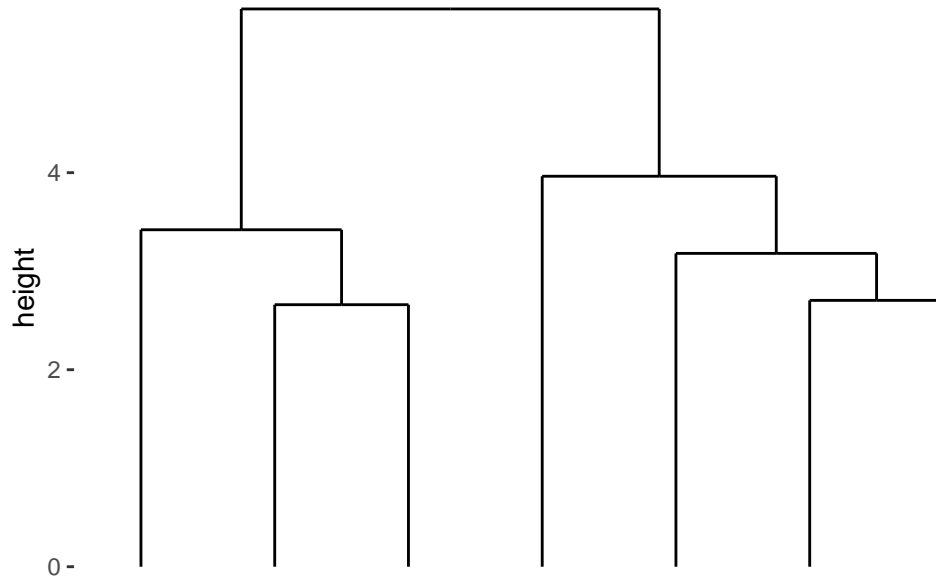
This produces a simple dendrogram. By default, `stack_alignv()` removes the axis text on the axis used for aligning observations. This is because it's often unclear which plot should display the axis text, as typically, we want it to appear in only one plot. However, you can easily use the `theme()` function to control where the axis text appears.

Internally, `align_dendro()` will reorder the observations based on the dendrogram, and other plots in the layout will follow this ordering.

```
stack_alignv(small_mat) +                                    ①
    align_dendro() +                                         ②
    ggalign(data = rowSums) +                                ③
    geom_bar(aes(.names, value), stat = "identity") +        ④
    theme(axis.text.y = element_text())                      ⑤
```

① We initialize a vertical stack.
② Reorder the observations based on hierarchical clustering and add a dendrogram tree.
③ Create a new ggplot in the layout, and use data based on the sum of the layout data.
④ Add a bar layer.
⑤ Add y-axis text.
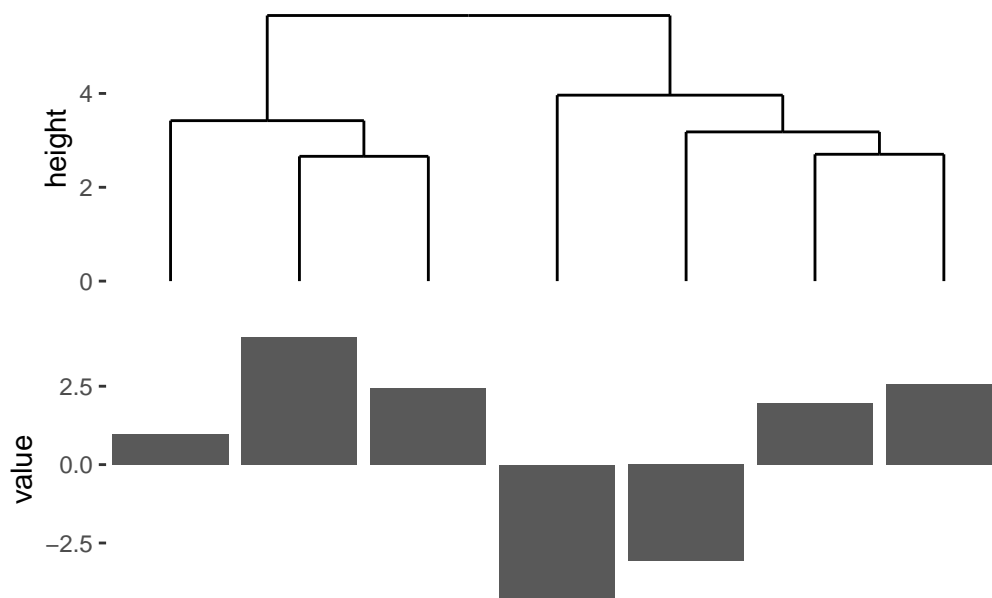
The data in the underlying `ggplot` object of `ggalign()` function contains at least following columns (more details will be introduced in the Section 5.1):

- `.panel`: the group panel for the aligned axis. It means `x-axis` for vertical stack layout, `y-axis` for horizontal stack layout.
- `.x` or `.y`: the x or y coordinates
- `.names` and `.index`: A factor of the names (only applicable when names exists) and an integer of index of the original data.
- `value`: the actual value (only applicable if `data` is a `matrix` or atomic vector).

It is recommended to use `.x`/`.y`, or `.names` as the x/y mapping.

`align_dendro()` can also split the observations into groups.

```
stack_alignv(small_mat) +                                                    ①
    align_dendro(k = 3) +                                                    ②
    ggalign(data = rowSums) +                                                ③
    geom_bar(aes(.names, value, fill = .panel), stat = "identity") +         ④
    scale_fill_brewer(palette = "Dark2", name = "Group") +                   ⑤
    theme(axis.text.y = element_text())                                      ⑥
```

① We initialize a vertical stack.
② Reorder and group the observations based on hierarchical clustering, and add a dendrogram tree.

10

③ Create a new ggplot in the layout, and use data based on the sum of the layout data.
④ Add a bar layer.
⑤ Add fill mapping scale.
⑥ Add y-axis text.
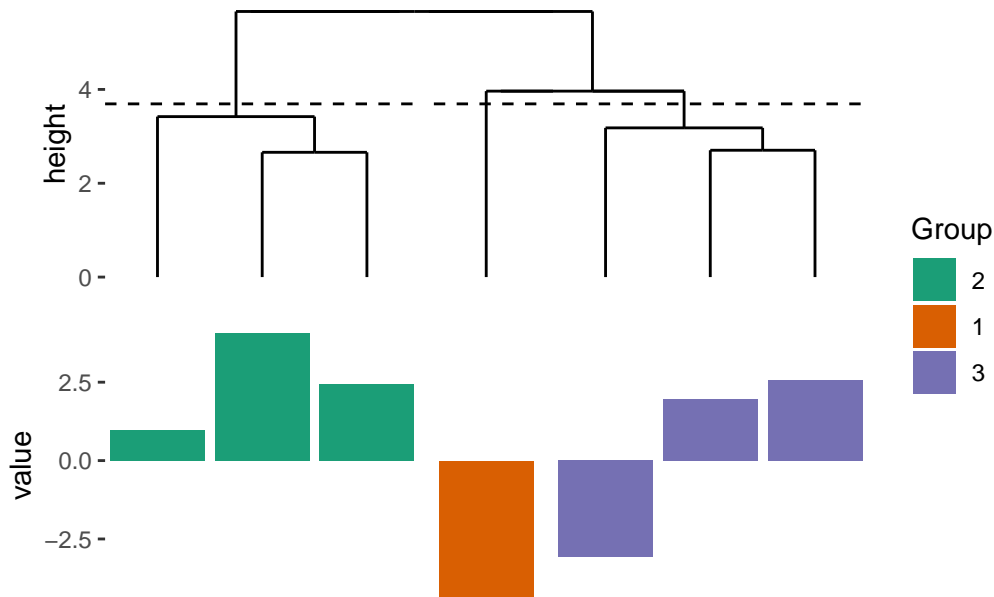
One common visualization associated with the dendrogram is the heatmap. You can use ggheatmap() to initialize a heatmap layout. When grouping the observations using align_dendro(k = 3), a special column named branch is added, which you can use to color the dendrogram tree.

```
ggheatmap(small_mat) +                                      ①
    anno_left() +                                           ②
    align_dendro(aes(color = branch), k = 3) +              ③
    scale_fill_brewer(palette = "Dark2")                    ④
#> > heatmap built with `geom_tile()`
```

① We initialize a heatmap layout.
② we initialize an annotation in the left side of the heatmap body, and set it as the active context, in this way, all following addition will be directed to the left annotation.
③ Reorder and group the observations based on hierarchical clustering, and add a dendrogram tree, coloring the tree by branch.
④ Add fill mapping scale.
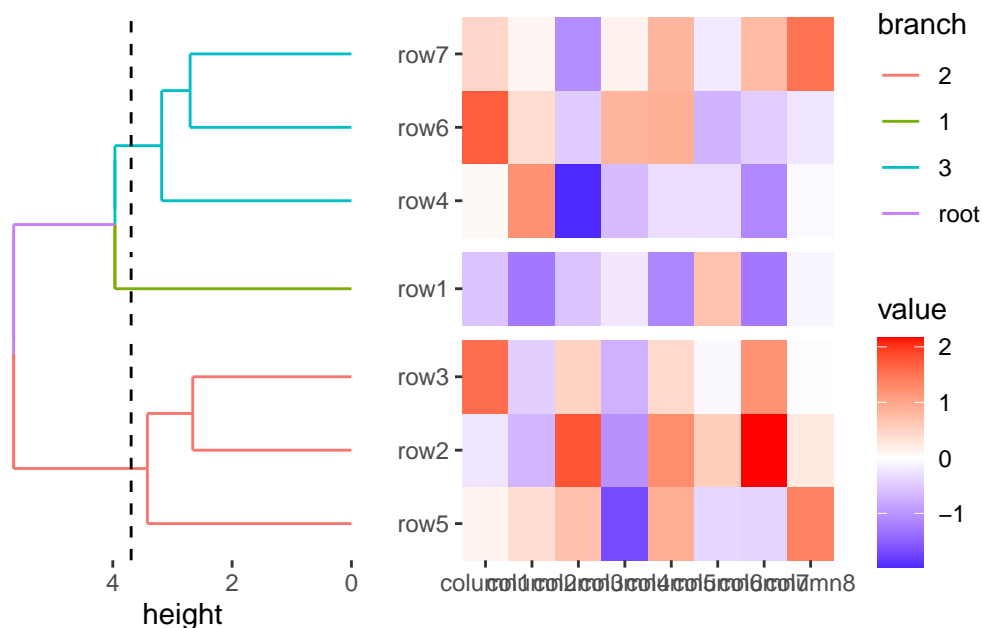
`ggheatmap()` will automatically add axis text in the heatmap body, so you don't need to manually adjust axis text visibility using `theme(axis.text.x = element_text())`/`theme(axis.text.y = element_text())`.

Having explored the core principles of **ggalign**, you should now be familiar with its basic workflow. In the next chapter, we'll introduce the `StackLayout` functionality, a powerful tool for arranging multiple plots in a stacked fashion—either horizontally or vertically—while maintaining full control over their alignment. We'll explore how `stack_layout()` and its various functions can give you even greater flexibility in creating sophisticated layouts.

2 stack layout

`stack_layout()` arranges plots either horizontally or vertically, and we can also use the alias `ggstack()`. Based on whether we want to align the observations, there are two types of stack layouts:

- `stack_align()`: align the observations along the stack.
- `stack_free()`: don't align the observations.

Several aliases are available for convenience:

- `stack_alignv`: Aligns the stack vertically (special case of `stack_align()`).
- `stack_alignh`: Aligns the stack horizontally (special case of `stack_align()`).
- `stack_freev`: A vertical version of `stack_free()`.
- `stack_freeh`: A horizontal version of `stack_free()`.


```
library(ggalign)
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))
```

2.1 Input data

When aligning observations, we typically use a matrix, as it is easy to melt the matrix into a long-formatted data frame. Additionally, matrices are used to fit the observation concept, as they can be transposed (rows to columns, columns to rows), which is necessary for use in functions like `quad_layout()` and `ggheatmap()`, where observations may be aligned in both directions simultaneously.

- For `stack_free()`, a data frame is required, and the input will be automatically converted using `fortify_data_frame()` if needed.
- For `stack_align()`, a matrix is required, and the input will be automatically converted using `fortify_matrix()` if needed.

By default, `fortify_data_frame()` will invoke the `ggplot2::fortify()` function for conversion. Note, for matrix, it will be converted to a long-formatted data frame.

`stack_align()`/`stack_free()` will set up the layout, but no plot will be drawn until you add a plot element:

```
stack_alignh(small_mat) +                                              ①
    layout_annotation(                                                 ②
        theme = theme(plot.background = element_rect(color = "red"))
    )
# the same for `stack_free()`
```

① initialize a vertical stack layout.
② Add a plot background in for the entire layout.
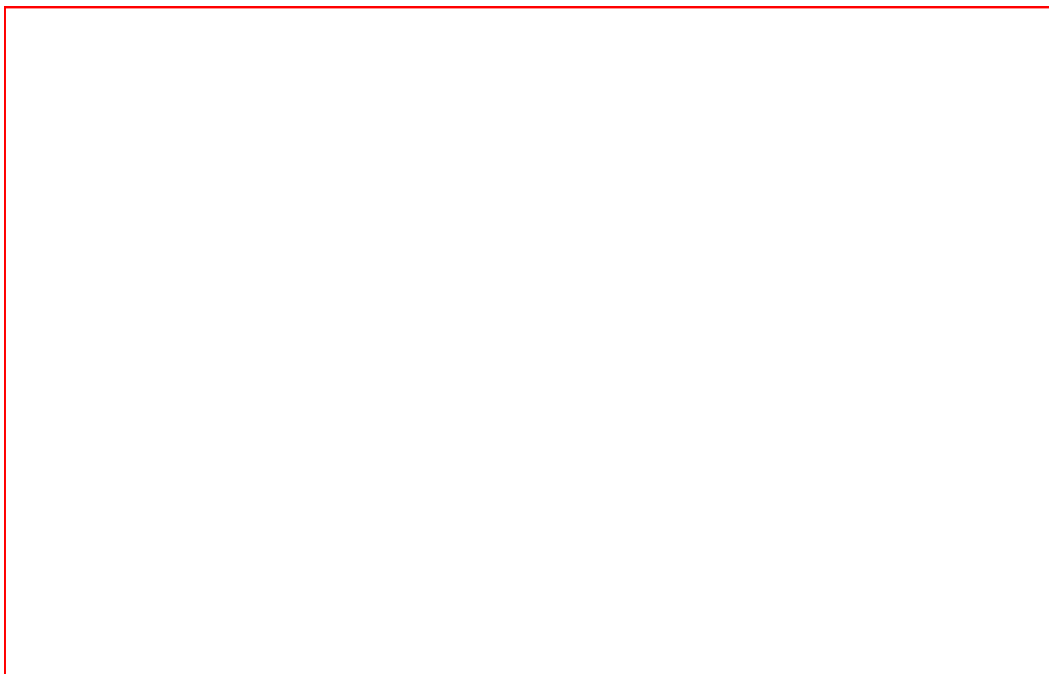
In this example, we use `layout_annotation()` to insert a plot background in the entire layout, it can be also used to control the theme of title, subtitle, caption (`layout_title()`), guides, margins, panel.border.

2.2 Layout Customize

When we use `stack_align()`, it aligns the observations across multiple plots along the specified direction:

- For `stack_alignh()`: Alignment occurs along the horizontal direction (y-axis).
- For `stack_alignv()`: Alignment occurs along the vertical direction (x-axis).

The package offers a suite of `align_*` functions designed to give you precise control over the observations. These functions enable you to reorder the observations or partition the observations into multiple groups. Instead of detailing each `align_*` function individually, we will focus on the general usage and how to combine them with `stack_align()`.

Here, we remain take `align_dendro()` as an example, it can reorder the observations, split them into groups, and can add a plot for visualization.
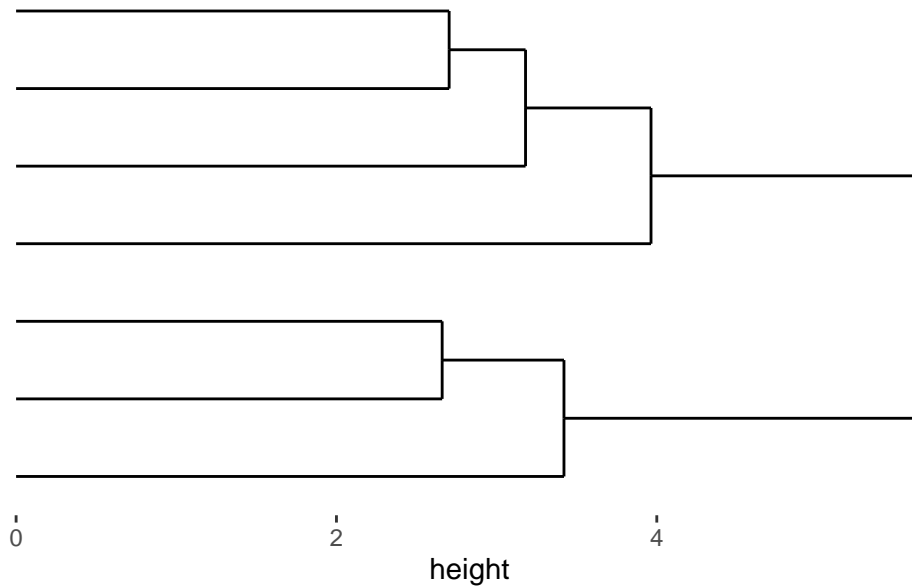
When used for `stack_alignh()`, the observations are aligned along the `y-axis`:

```
stack_alignh(small_mat) +                                               ①
    align_dendro()                                                      ②
```

① initialize a horizontal stack layout.
② reorder the observations based on the hierarchical clustering, add a dendrogram tree, and
set the active plot to this plot.

When used for `stack_alignv()`, the observations are aligned along the `x-axis`:

```
stack_alignv(small_mat) +                                               ①
    align_dendro()                                                      ②
```
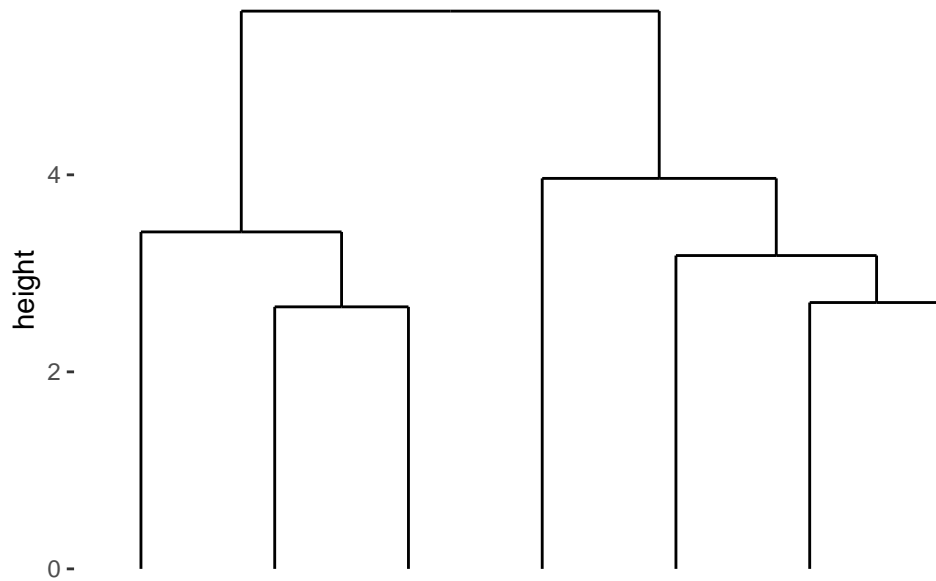
① initialize a vertical stack layout.
② reorder the observations based on the hierarchical clustering, add a dendrogram tree, and
set the active plot to this plot.

When `align_dendro()` is added to the layout, it performs following actions:

1. reorder the observations.
2. set the active plot to the dendrogram.

The active plot refers to the plot that subsequent ggplot2 components will target. In this case, the active plot is the dendrogram, and any new layers added will be applied to it. For instance, we can add additional layers to visualize the dendrogram's structure or data. The default data underlying the `ggplot` object of `align_dendro()` consists of the dendrogram node data. It is also possible to use the dendrogram's edge data for customization, which I will introduce in Section 5.4.

```
stack_alignh(small_mat) +                                    ①
    align_dendro() +                                         ②
    geom_point()                                             ③
```

① initialize a horizontal stack layout.
② reorder the observations based on the hierarchical clustering, add a dendrogram tree, and set the active plot to this plot.
③ add a point layer to the dendrogram
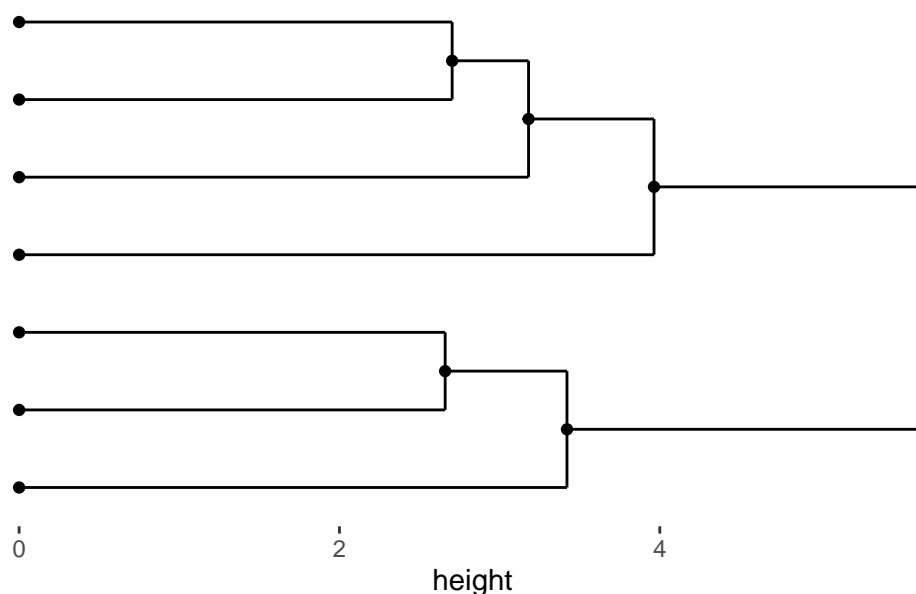
The `active` argument controls whether a plot should be set as the active plot. It accepts an `active()` object with the `use` argument to specify if the plot should be active when added.

```
stack_alignh(small_mat) +                                          ①
    align_dendro(active = active(use = FALSE)) +                   ②
    geom_point()                                                   ③
#> Error in `stack_layout_add()`:
#> ! Cannot add `geom_point()` to `stack_align()`
#> i No active plot component
#> i Did you forget to initialize a <ggplot> object with `ggalign()` or
#>   `ggfree()`?
```

① initialize a horizontal stack layout.
② reorder the observations based on the hierarchical clustering, add a dendrogram tree, but don't set the active plot to this plot.
③ try to add a point layer to the dendrogram, should fail due to no active plot

Usually, you don't need to set this manually, as the active context is automatically applied only for functions that add plot areas. You can inspect whether a `align_*` function will add a plot by print it:

```
align_dendro()
#> `align_dendro()` object:
```

```
#>    plot: yes
#>    reorder: yes
#>    split: no
```

You might find it confusing that we mentioned `align_dendro()` will split observations into groups, while the print output shows `split = "no"`. This happens because we haven't speci-fied the `k/h` argument in `align_dendro()`.

```
align_dendro(k = 3L)
#> `align_dendro()` object:
#>    plot: yes
#>    reorder: yes
#>    split: yes
```

You don't need to explicitly provide `data` to `align_dendro()`. By default, it inherits data from the layout. However, you can always provide another data source, but note that this package uses the concept of `number of observations` (`NROW()`). When aligning the observations, you must ensure the number of observations is consistent across all plots.

```
set.seed(123)
stack_alignh(small_mat) +                                        ①
    align_dendro(data = matrix(rnorm(56), nrow = 8)) +           ②
    theme(axis.text.y = element_text())                          ③
#> Error in `align()`:
#> ! `align_dendro(data = matrix(rnorm(56), nrow = 8))` (nobs: 8) is not
#>   compatible with the `stack_align()` (nobs: 7)
```

① initialize a horizontal stack layout.
② reorder the observations based on hierarchical clustering, add a dendrogram tree, and set the active plot to this one, using self-provided data. This should fail because the number of observations is inconsistent.
③ try to add y-axis text to the dendrogram.

```
set.seed(123)
stack_alignh(small_mat) +                                        ①
    align_dendro(data = matrix(rnorm(70), nrow = 7)) +           ②
    theme(axis.text.y = element_text())                          ③
```

① initialize a horizontal stack layout.
② reorder the observations based on the hierarchical clustering, add a dendrogram tree, and set the active plot to this plot, using self-provided data

③ add y-axis text to the dendrogram.
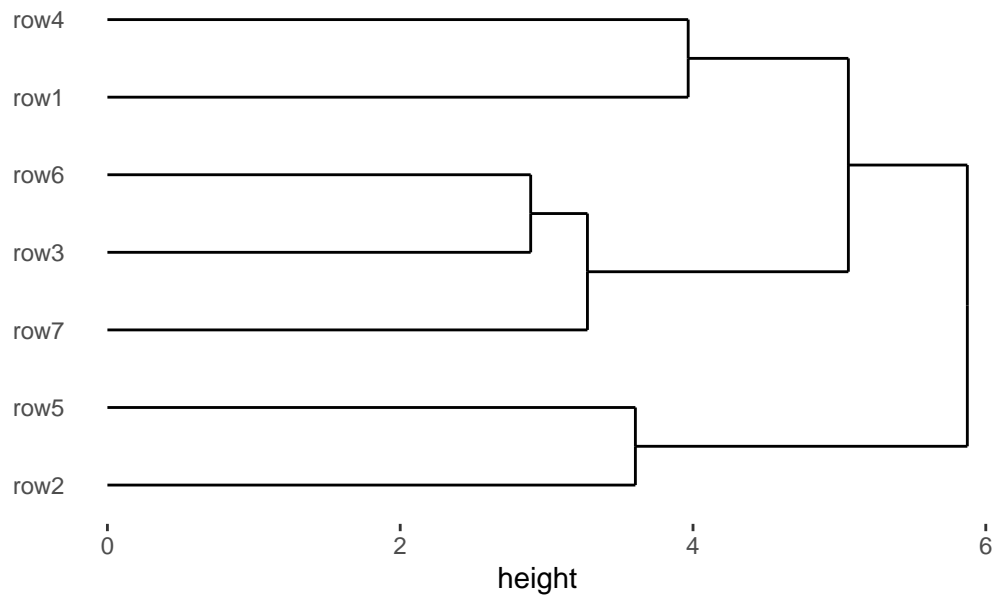
Alternatively, you can provide a function (or purrr-lambda) that will be applied to the layout's matrix. For layouts that align observations, a matrix is always required, so the data input must be in matrix form.

```
set.seed(123)
stack_alignh(small_mat) +
    align_dendro(data = ~ .x[sample(nrow(.x)), ]) +
    theme(axis.text.y = element_text())
```
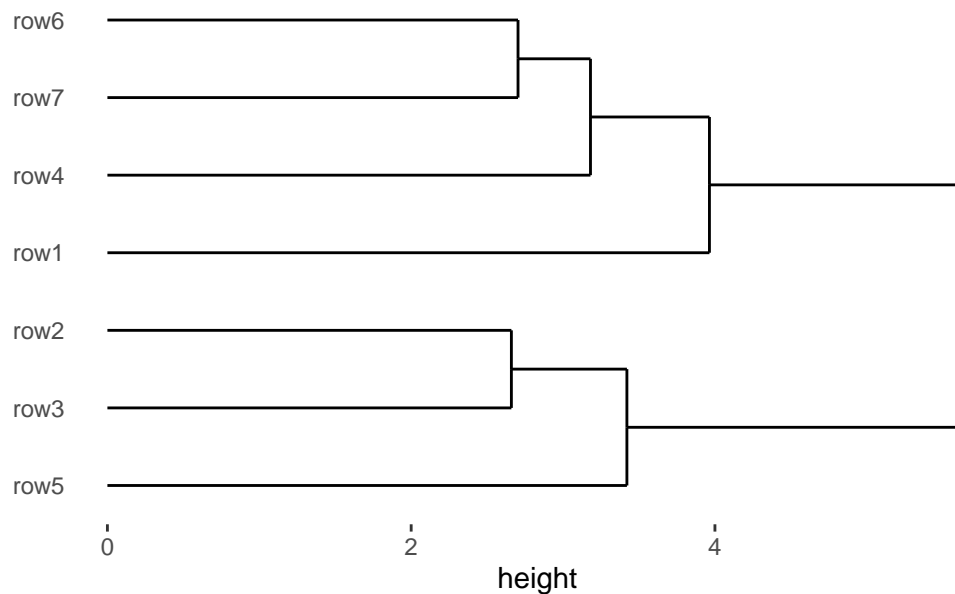
Without adding another plot, it's difficult to appreciate the benefits. Let's now explore how to incorporate a plot.

2.3 Plot initialize

There are two primary functions for adding plots:

- `align_gg()`/`ggalign()`: Create a ggplot object and align with the layout.
- `free_gg()`/`ggfree()`: Create a ggplot object without aligning.

Both functions initialize a `ggplot` object and, by default, set the `active` plot when added to the layout.

For `stack_align()`, plots can be added regardless of whether they need to align observations.

```
stack_alignh(small_mat) +
    align_dendro() +
    ggalign(data = rowSums) +
    geom_bar(aes(value, .names), stat = "identity") +
    theme(axis.text.y = element_text())
```
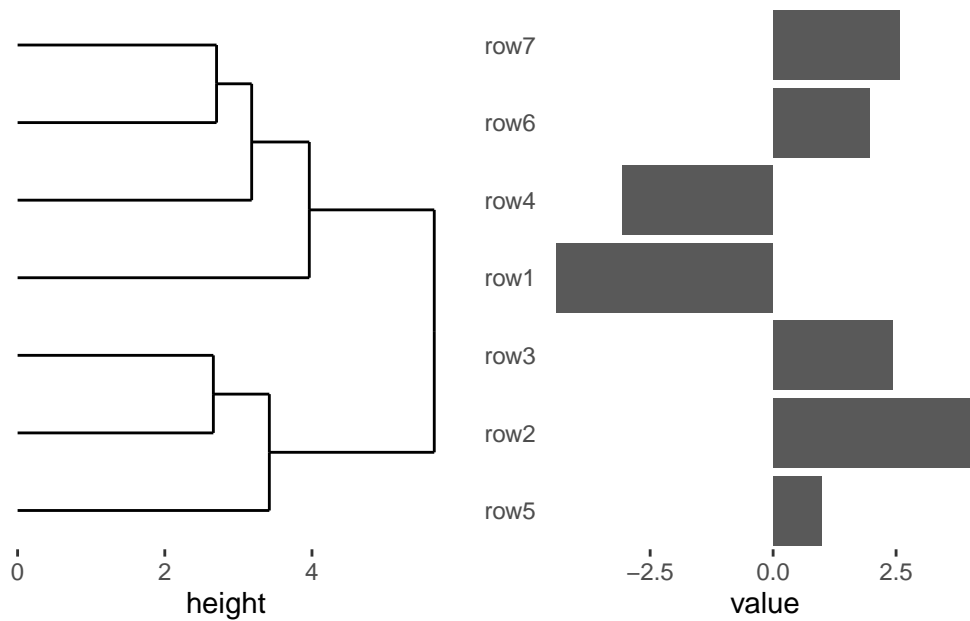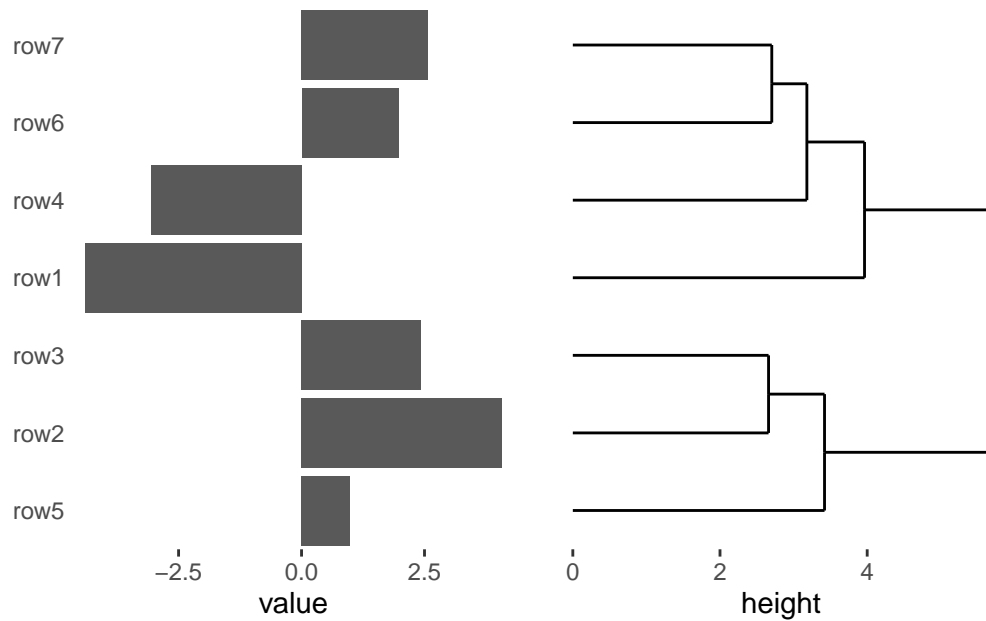
You can build the plot separately and then add it to the layout:

```
my_bar <- ggalign(data = rowSums) +
    geom_bar(aes(value, .names), stat = "identity") +
    theme(axis.text.y = element_text())
stack_alignh(small_mat) +
    align_dendro() +
    my_bar
```
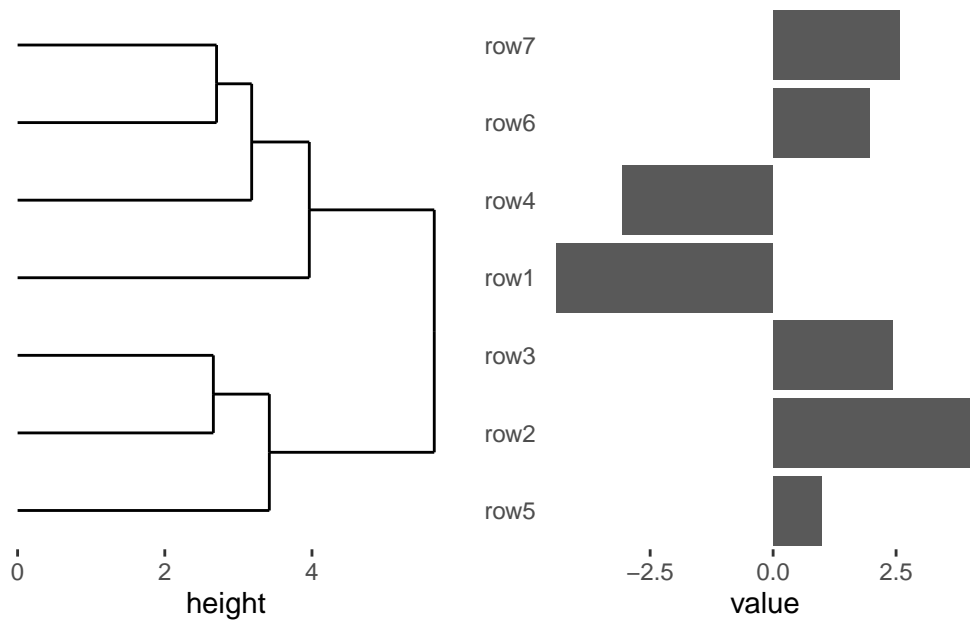
The `active` argument can also control the place of the plot area to be added. It accepts an `active()` object with the `order` argument to specify the order of the plot area.
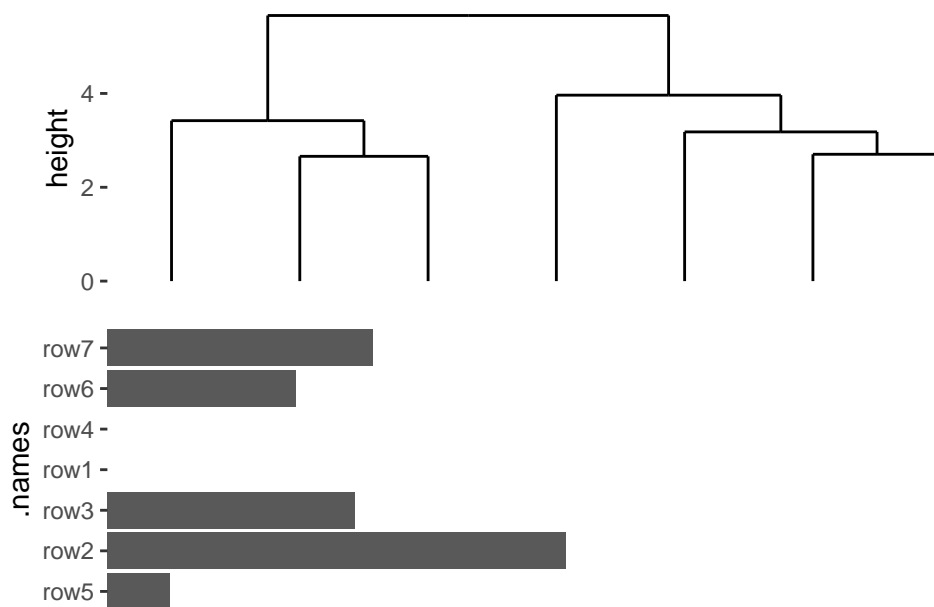
```
stack_alignh(small_mat) +
    align_dendro() +
    ggalign(data = rowSums, active = active(order = 1)) +
    geom_bar(aes(value, .names), stat = "identity") +
    theme(axis.text.y = element_text())
```

You can also stack plots vertically using `stack_alignv()`:

```
stack_alignv(small_mat) +
    align_dendro() +
    ggalign(data = rowSums) +
    geom_bar(aes(value, .names), stat = "identity") +
    theme(axis.text.y = element_text())
```

`stack_align()` can also add plot without aligning observations. `free_gg()` focuses on layout integration without enforcing strict axis alignment. `ggfree()` is an alias for `free_gg`.
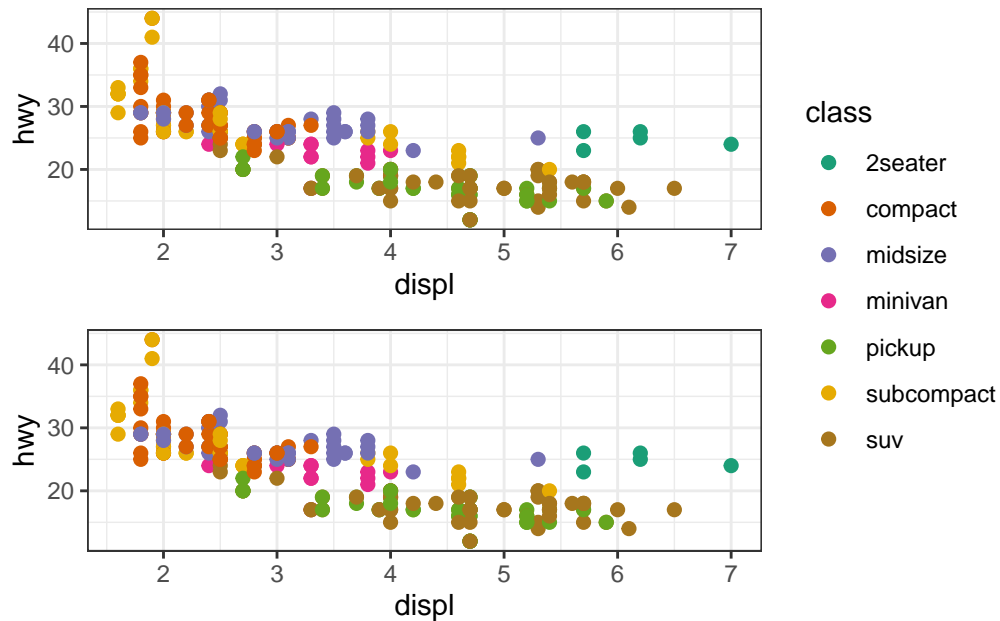
```
stack_alignv() +
    ggfree(mpg, aes(displ, hwy, colour = class)) +
    geom_point(size = 2) +
    ggfree(mpg, aes(displ, hwy, colour = class)) +
    geom_point(size = 2) &
    scale_color_brewer(palette = "Dark2") &
    theme_bw()
```
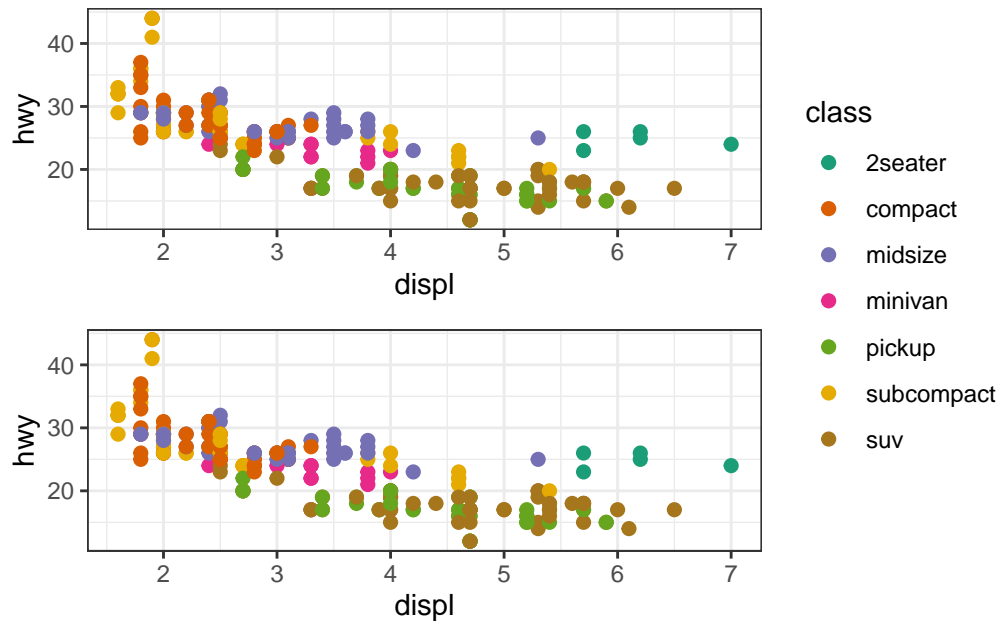
The `&` operator applies the added element to all plots in the layout, similar to its functionality in the `patchwork` package.

For `stack_free()`, only free plots (`ggfree()`) can be added. This layout arranges plots in one row or column without enforcing axis alignment:

```
stack_freev(mpg) +
    ggfree(mapping = aes(displ, hwy, colour = class)) +
    geom_point(size = 2) +
    ggfree(mapping = aes(displ, hwy, colour = class)) +
    geom_point(size = 2) &
    scale_color_brewer(palette = "Dark2") &
    theme_bw()
```

By default, `ggfree()` will also inherit data from the layout and call `fortify_data_frame()` to convert the data to a data frame. So, note that if the layout data is a matrix, it will be converted into a long-formatted data frame.

2.4 Plot Size

Both `ggalign()` and `ggfree()` functions have a `size` argument to control the relative `width` (for horizontal stack layout) or `height` (for vertical stack layout) of the plot's panel area.

```
stack_freev(mpg) +
    ggfree(mapping = aes(displ, hwy, colour = class), size = 2) +
    geom_point(size = 2) +
    ggfree(mapping = aes(displ, hwy, colour = class), size = 1) +
    geom_point(size = 2) &
    scale_color_brewer(palette = "Dark2") &
    theme_bw()
```
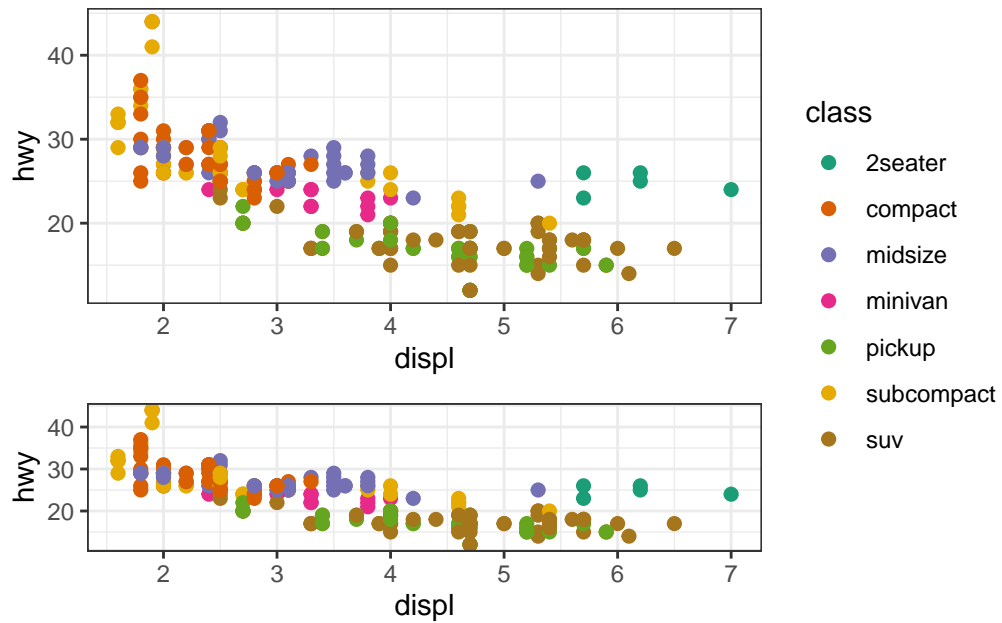
Alternatively, you can define an absolute size by using a `unit()` object:

```
stack_freev(mpg) +
    ggfree(mapping = aes(displ, hwy, colour = class), size = unit(1, "cm")) +
    geom_point(size = 2) +
    ggfree(mapping = aes(displ, hwy, colour = class)) +
    geom_point(size = 2) &
    scale_color_brewer(palette = "Dark2") &
    theme_bw()
```
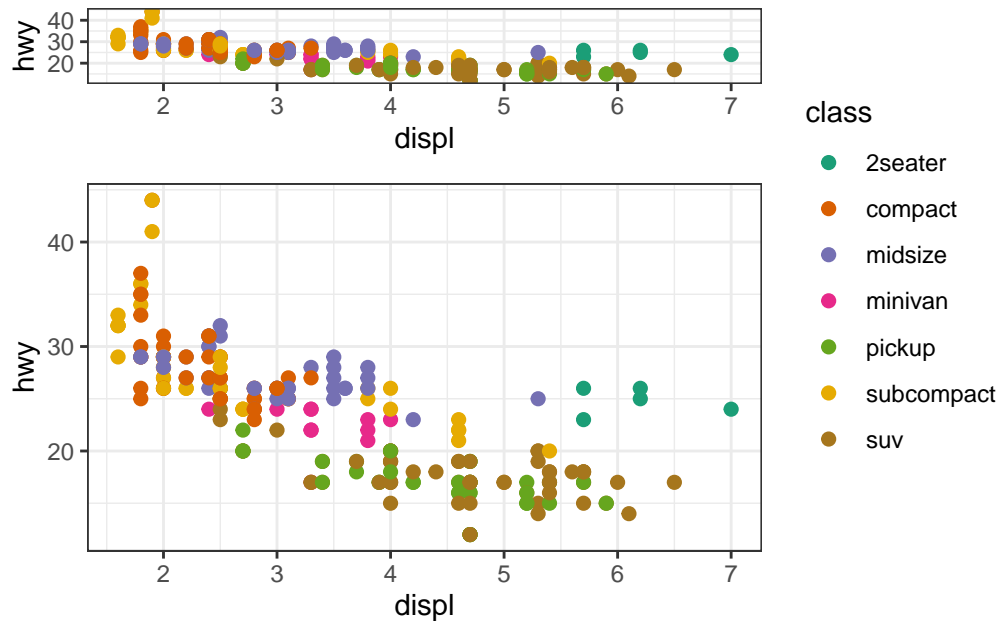
2.5 active plot

As mentioned earlier, the active plot refers to the plot that subsequent ggplot2 components will target. The package provide two functions to work with active plot.
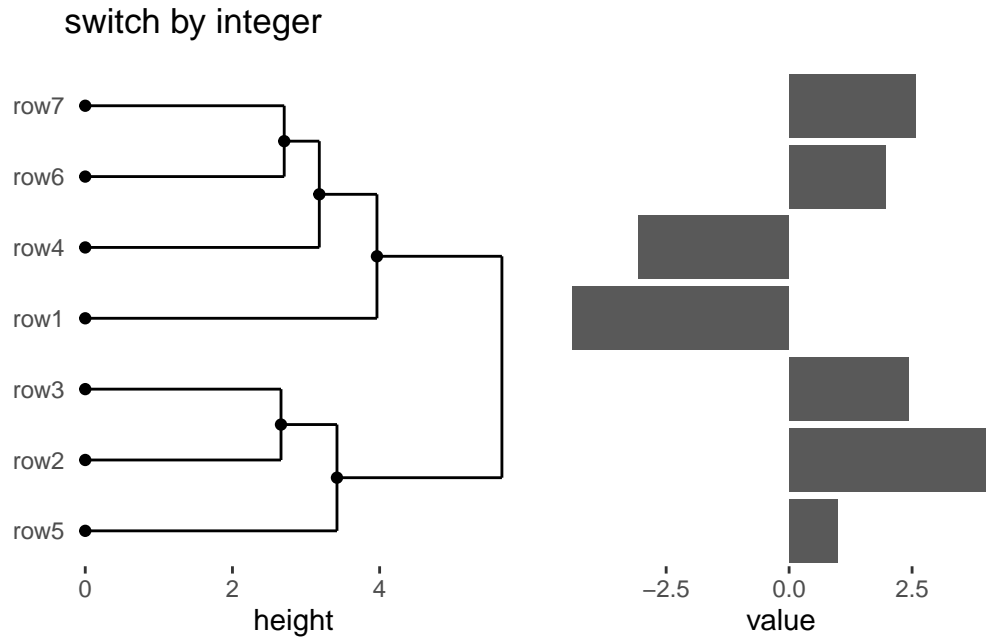
- `stack_switch()`: switch the active context
- `stack_active()`: An alias for `stack_switch()`, which sets `what = NULL`

The `stack_switch()` function accepts the `what` argument, which can either be the index of the plot added (based on its adding order) or the plot name specified via the `active()` object using the `name` argument.

Note that the `what` argument must be explicitly named, as it is placed second in the function signature. This is because, in most cases, we don't need to switch the active plot manually—adjusting the order of plot additions typically suffices.

```
stack_alignh(small_mat) +
    align_dendro() +
    ggalign(data = rowSums) +
    geom_bar(aes(value, .names), stat = "identity") +
    # switch to the `align_dendro()` plot area
    stack_switch(what = 1) +
```

```
    geom_point() +
    theme(axis.text.y = element_text()) +
    layout_title(title = "switch by integer")
```

switch by integer


```
stack_alignh(small_mat) +
    align_dendro(active = active(name = "tree")) +
    ggalign(data = rowSums) +
    geom_bar(aes(value, .names), stat = "identity") +
    # switch to the `align_dendro()` plot area
    stack_switch(what = "tree") +
    geom_point() +
    theme(axis.text.y = element_text()) +
    layout_title(title = "switch by string")
```

switch by string

In the example, we use `layout_title()` to insert a title for the entire layout. Alternatively, you can add a title to a single plot with `ggtitle()`.

By setting `what = NULL` (or alias `stack_active()`), we remove the active plot. This is particularly useful when the active plot is a nested `Layout` object, as any additions would otherwise be directed to that nested `Layout`. By removing the active plot, you can continue adding components directly to the `StackLayout`.

In the next chapter, we will dive into the `HeatmapLayout`, which can take the `StackLayout` as input. Heatmap layouts offer additional features for aligning observations in both directions. Let's move ahead and explore how heatmaps can be seamlessly integrated into your layout workflows.

3 heatmap layout

The `heatmap_layout()` function provides a powerful way to create customizable heatmaps in R using `ggplot2`. This chapter will guide you through its usage.

`heatmap_layout()` is a specialized version of `quad_alignb()`, which itself is a specific variant of `QuadLayout` (`quad_layout()`) designed to align observations both horizontally and vertically. We introduce `heatmap_layout()` directly, as it is more familiar to many users, especially those experienced with popular heatmap packages like `pheatmap` and `ComplexHeatmap`.

QuadLayout
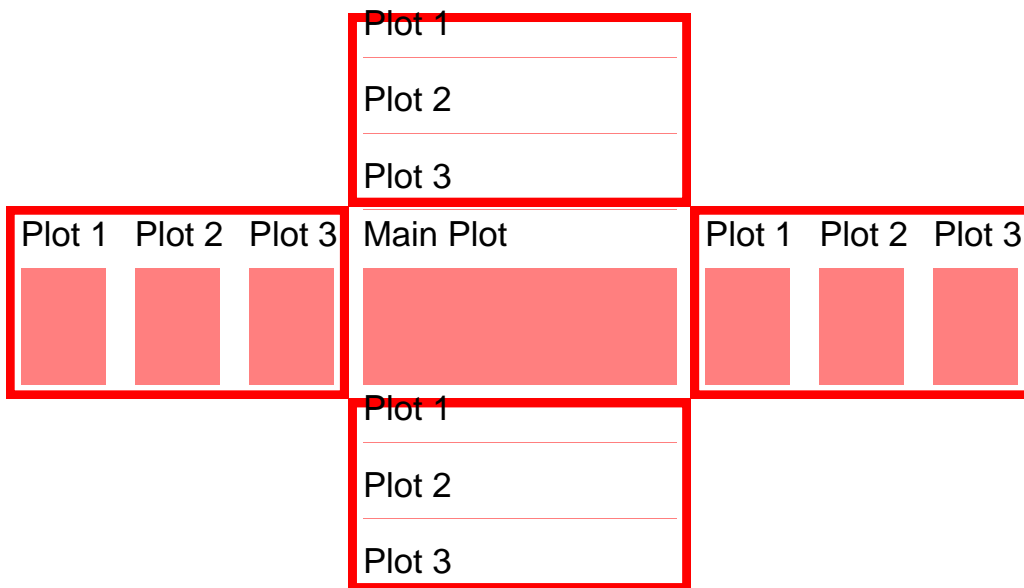
`heatmap_layout()` simplifies the creation of heatmap plots by integrating essential elements for a standard heatmap layout, ensuring that the appropriate data mapping and visualization layers are automatically applied. `ggheatmap()` is an alias for `heatmap_layout()`.

```r
library(ggalign)
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
```

```
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))
```

3.1 input data

As mentioned in Section 2.1, we typically require a matrix for the `Layout` which need align observations. Internally, `fortify_matrix()` will be used to process the data. You can provide a numeric or character vector, a data frame, or any other data type that can be converted into a matrix using `as.matrix()`.

```
ggheatmap(small_mat)
#> > heatmap built with `geom_tile()`
```
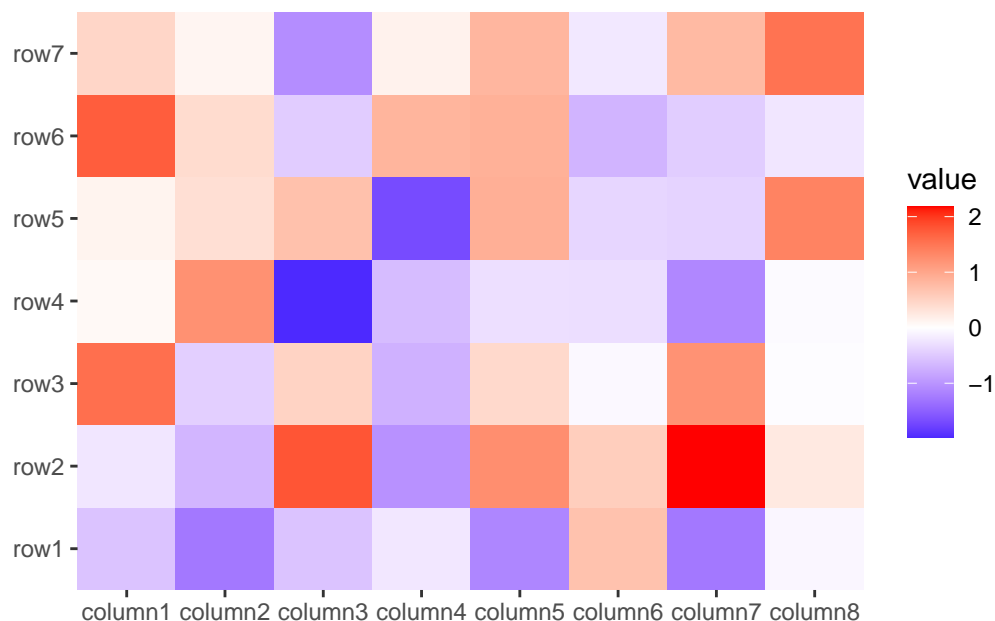

3.2 Main plot (heatmap body)

The `ggheatmap()`/`quad_layout()` functions arrange plots in the Quad-Side layout of the main plot. When the layout is initialized, a `ggplot` object is automatically created for the main plot.

For `ggheatmap()`, the matrix input will be converted into a long-format data frame when drawing. The data in the underlying `ggplot` object includes the following columns:

- `.xpanel` and `.ypanel`: the column and row panel

- `.x` and `.y`: the x and y coordinates

- `.row_names` and `.column_names`: A factor of the row and column names of the original matrix (only applicable when names exist).

- `.row_index` and `.column_index`: the row and column index of the original matrix.

- `value`: the actual matrix value.

The default mapping will use `aes(.data$.x, .data$.y)`, but can be customized using `mapping` argument.

By default, the main plot is regarded as the active plot, meaning you can add ggplot2 elements directly to the main plot.

```
ggheatmap(small_mat) +
    geom_point() +
    scale_fill_viridis_c()
#> > heatmap built with `geom_tile()`
```

By default, `ggheatmap()` adds a heatmap layer. If the matrix has more than 20,000 cells (`nrow * ncol > 20000`), it uses `geom_raster()` for performance efficiency; for smaller matrices, `geom_tile()` is used. You can explicitly choose the layer by providing a single string (`"raster"` or `"tile"`) in the `filling` argument.

```
ggheatmap(small_mat, filling = "raster")
```



```
ggheatmap(small_mat, filling = "tile")
```

Note, the filling layer will always use `mapping` of `aes(.data$.x, .data$.y)`, if you want to customize filling, you can set `filling = NULL`, which will remove the filling layer and allow you to add custom filling geoms.

```
ggheatmap(small_mat, filling = NULL) +
    geom_tile(aes(fill = value), color = "black", width = 0.9, height = 0.9)
```

A heatmap pie charts can be easily drawn:

```
set.seed(123)
ggheatmap(matrix(runif(360L), nrow = 20L), filling = NULL) +
    geom_pie(aes(angle = value * 360, fill = value))
```

For more complex customizations of pie charts, you can try using `ggforce::geom_arc_bar()` instead.

3.3 rasterization

When working with large heatmaps, it's often beneficial to rasterize the heatmap body layer. You can achieve this by using the `raster_magick()` function. The `res` argument controls the resolution of the raster image. By default, the `res` argument matches the resolution of the current device, but specifying a different value can help reduce the resolution of the rasterized heatmap body.

```
ggheatmap(small_mat, filling = NULL) +
    raster_magick(geom_tile(aes(fill = value)), res = 50)
```

By leveraging `raster_magick()`, you can also perform image post-processing using the `magick` package. This allows for custom image resizing with filters.

```
ggheatmap(small_mat, filling = NULL) +
    # Use `magick::filter_types()` to check available `filter` arguments
    raster_magick(geom_raster(aes(fill = value)),
        magick = function(image) {
            magick::image_resize(image,
                # we resize to the 50% of width
                geometry = "50%x", filter = "Lanczos"
            )
        }
    )
```

Note: When using `magick::image_resize()`, you should specify the `geometry` argument to resize the image. If only the `filter` is specified, it will only distort the image data (though subtle). For more information on image resizing, refer to ImageMagick's resize documentation.

You can also rasterize all plots in the layout directly with `raster_magick()`. This method is defined for both `ggheatmap()`/`quad_layout()` and `stack_layout()` objects.

Additionally, You can use external packages like ggrastr or ggfx to rasterize the heatmap body.

```
ggheatmap(small_mat, filling = FALSE) +
    ggrastr::rasterise(geom_tile(aes(fill = value)), dev = "ragg")
```

Likewise, you can also rasterize all plots in the layout directly with `ggrastr::rasterise()` for both `ggheatmap()`/`quad_layout()` and `stack_layout()`.

```
ggrastr::rasterise(ggheatmap(small_mat), dev = "ragg")
#> > heatmap built with `geom_tile()`
```

Furthermore, ggfx offers many image filters that can be applied to ggplot2 layers. See the package for the details.

3.4 annotations

In ggheatmap()/quad_layout(), annotations are handled by a stack_layout() object and can be positioned at the top, left, bottom, or right of the main plot (heatmap body).

By default, ggheatmap()/quad_layout() do not activate an annotation, You can use quad_anno() to activate an annotation, directing all subsequent additions to the specified annotation position. The quad_anno() function has the following aliases:

- anno_top: A special case of quad_anno() with position = "top".
- anno_left: A special case of quad_anno() with position = "left".
- anno_bottom: A special case of quad_anno() with position = "bottom".
- anno_right: A special case of quad_anno() with position = "right".

When quad_anno() is added to a ggheatmap()/quad_layout(), it will try to automatically create a new stack_layout(). For top and bottom annotations, stack_alignv() or stack_freev() will be used; for left and right annotations, stack_alignh() or stack_freeh() will be applied.

quad_anno() will always attempt to initialize a stack_layout() with the same alignment as the current direction. This means that if observations need to be aligned horizontally,

`stack_alignh()` will be used for left and right annotations, otherwise, `stack_freeh()` will be initialized instead. The same logic applies for vertical alignment—`stack_alignv()` or `stack_freev()` will be applied for top and bottom annotations, depending on whether alignment is required. However, you can also manually add a `stack_free()` for directions that require alignment, which I'll cover in the following section.

Additionally, `quad_anno()` will set the active context to the annotation. This means that subsequent additions will be directed to the annotation rather than the main plot. We use the term `active context` in contrast to `active plot` (as described in Section 2.2), since the annotation is a `Layout` object.

```
ggheatmap(small_mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    # we set the active context to the left annotation
    anno_left() +
    align_dendro()
#> > heatmap built with `geom_tile()`
```
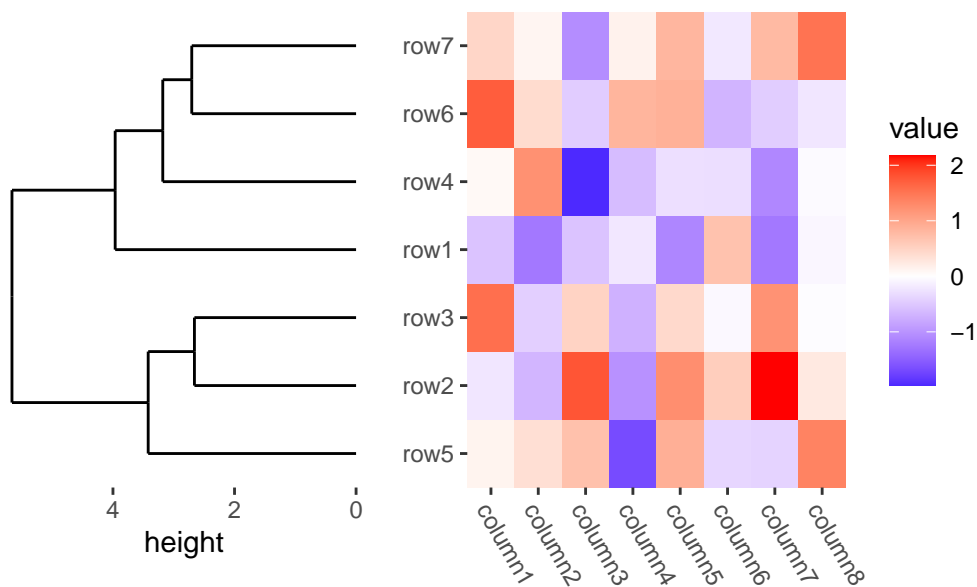

By default, the annotation `stack_layout()` will try to inherit data from `ggheatmap()`/`quad_layout()`. If the observations require alignment vertically, this means the data from `ggheatmap()`/`quad_layout()` should be a matrix, the column annotations will also require a matrix and the matrix from `ggheatmap()`/`quad_layout()` will be transposed for use in the column annotations.

```
ggheatmap(small_mat) +
    # we set the active context to the top annotation
    anno_top() +
    align_dendro()
#> > heatmap built with `geom_tile()`
```


You can further customize the layout design or add new plots in the annotation stack, as described in **?@sec-stack-layout**.

```
ggheatmap(small_mat) +
    # in the heatmap body, we set the axis text theme
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    # we set the active context to the right annotation
    anno_right() +
    # in the right annotation, we add a dendrogram
    align_dendro(k = 3L) +
    # in the dendrogram, we add a point layer
    geom_point(aes(color = factor(branch)))
#> > heatmap built with `geom_tile()`
```
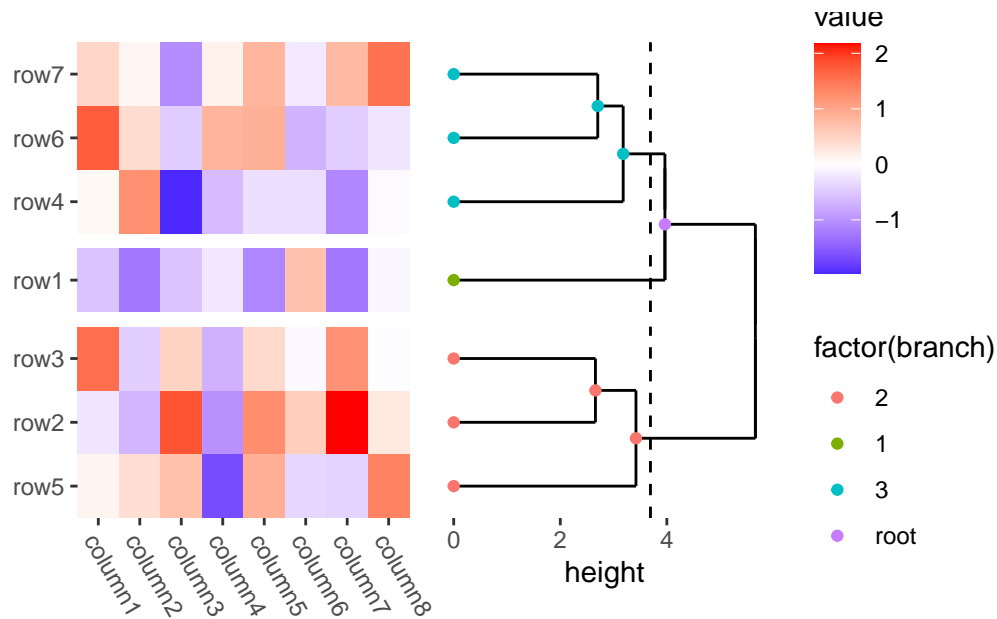
In this example:

- `anno_right()` initialize the right annotation stack, and change the active context to the right of the heatmap.
- `align_dendro(k = 3L)` adds a dendrogram to the annotation and sets itself as the active plot in the annotation stack.
- `geom_point(aes(color = factor(branch)))` is then added to this active plot within the annotation stack, here, it means the `align_dendro()` plot.

`ggheatmap()` aligns observations both horizontally and vertically, so it's safe to always use `quad_anno()` directly, as all annotations require a matrix, and the layout data is also a matrix. However, for `quad_alignh()` and `quad_alignv()` (which I'll discuss in more detail in a Chapter 6), which only align observations in one direction, the data in the layout may not fit the data for the annotation (when the layout requires alignment of observations, we typically use a matrix, regardless of whether alignment is needed in one or two directions)

- `quad_alignh()`: aligning observations in horizontal direction, for column annotations, we ll need a data frame for `stack_free()`.
- `quad_alignv()`: aligning observations in vertical direction, for row annotations, we ll need a data frame for `stack_free()`.

In both cases, `quad_anno()` won't initialize the annotation by default, instead, you must provide the annotation `stack_layout()` manually.

3.5 Adding stack layout

Similar to adding a plot in `stack_layout()` (**?@sec-stack-layout**), when the direction requires alignment, you can add both `stack_align()` and `stack_free()`. However, if the direction does not require alignment, you can add only `stack_free()`.

To add a `stack_layout()` to the `ggheatmap()`, we must prevent the automatical creation of annotation by `quad_anno()` by setting `initialize = FALSE`

```
my_stack_align <- stack_alignh(small_mat) +
    align_dendro(aes(color = branch), k = 3L)
ggheatmap(small_mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_right(initialize = FALSE) +
    my_stack_align +
    layout_title("stack_align()")
#> > heatmap built with `geom_tile()`
```
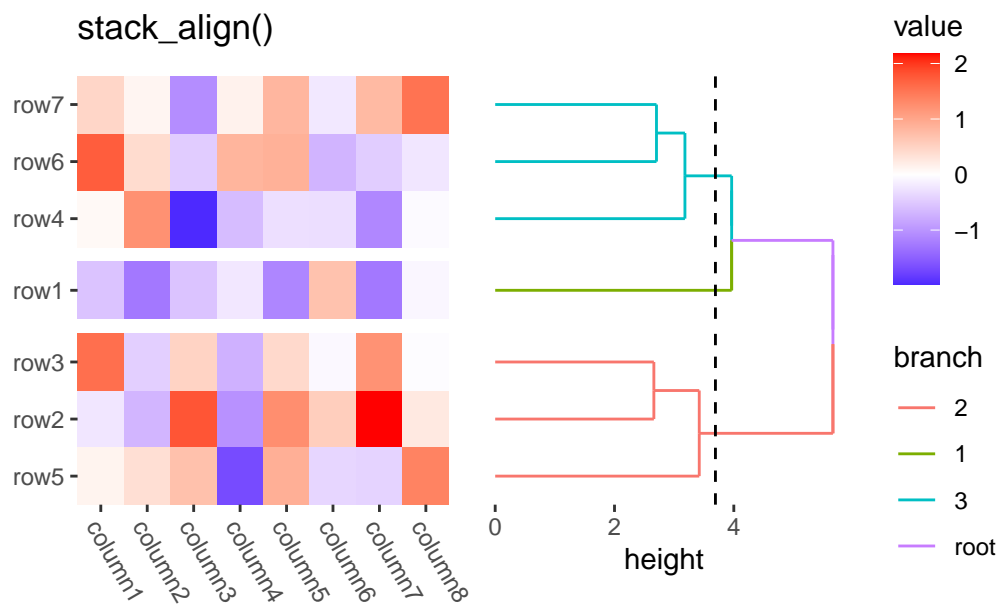


```
my_stack_free <- stack_freeh(mpg) +
    ggfree(mapping = aes(displ, hwy, colour = class)) +
    geom_point(size = 2)
ggheatmap(small_mat) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
```

```
    anno_right(initialize = FALSE) +
    my_stack_free +
    layout_title("stack_free()")
#> > heatmap built with `geom_tile()`
```
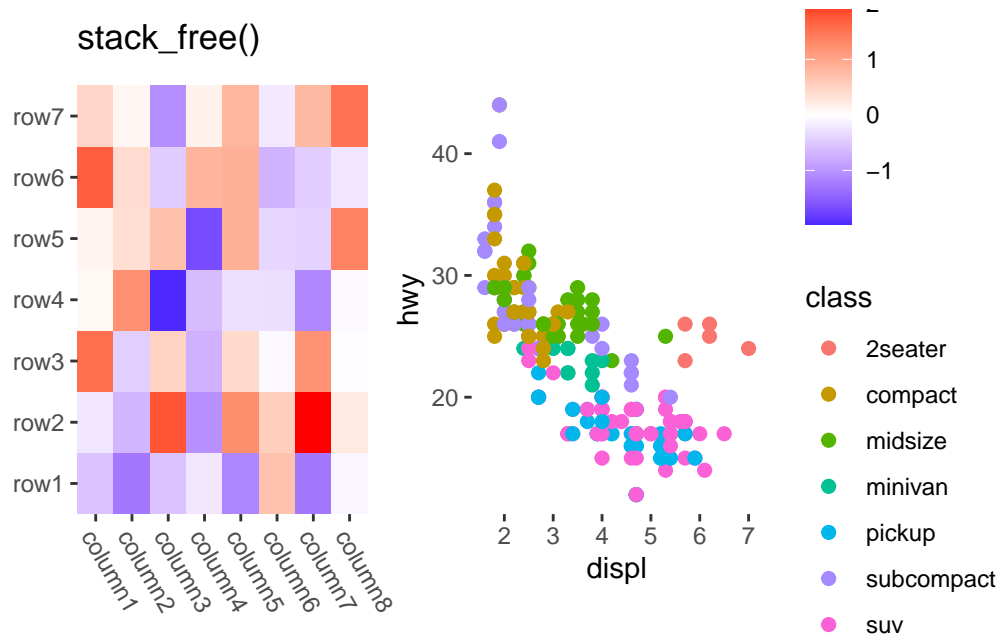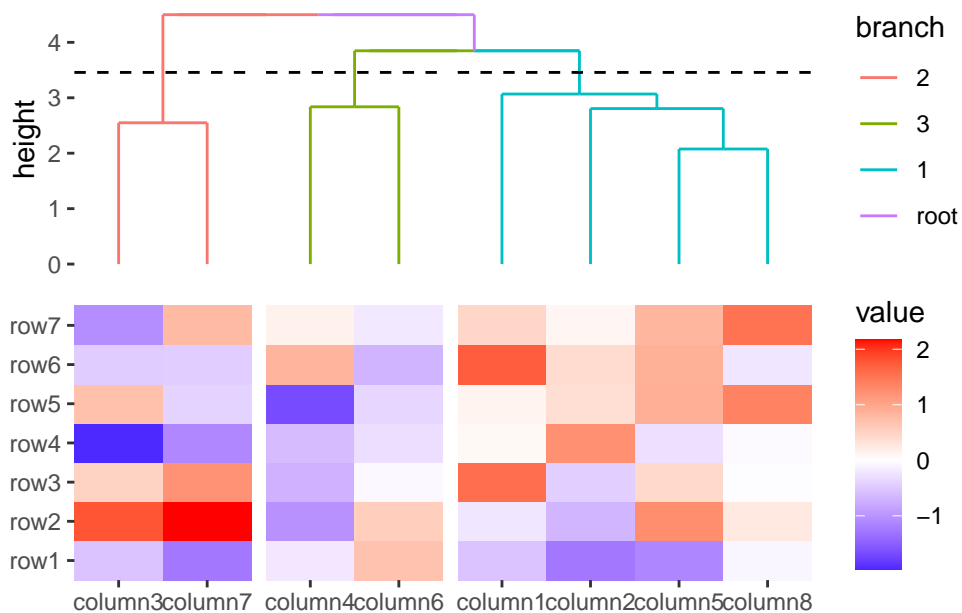

Note when aligning the observations, you must ensure the `number of observations` is consistent in the direction. So for column annotations, you need transpose the data manually.

```
my_stack <- stack_alignv(t(small_mat)) +
    align_dendro(aes(color = branch), k = 3L)
ggheatmap(small_mat) +
    anno_top(initialize = FALSE) +
    my_stack
#> > heatmap built with `geom_tile()`
```

3.6 `quad_active()`

To remove the active context and redirect additions back to the heatmap body, you can use `quad_active()`.

```
ggheatmap(small_mat) +
    # we set the active context to the top annotation
    anno_top() +
    # we split the observations into 3 groups by hierarchical clustering
    align_dendro(k = 3L) +
    # remove any active annotation
    quad_active() +
    # set fill color scale for the heatmap body
    scale_fill_viridis_c()
#> > heatmap built with `geom_tile()`
```
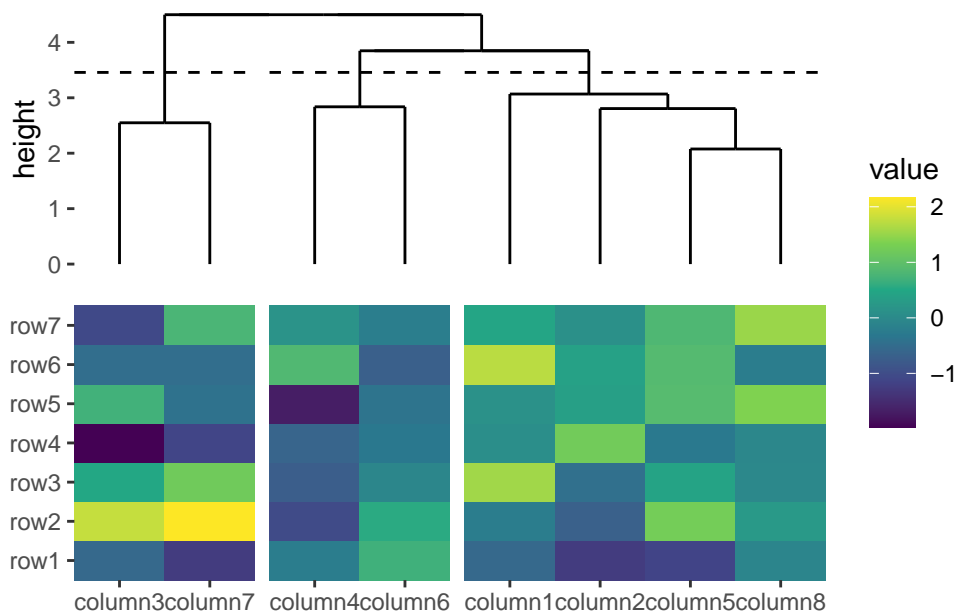
3.7 `quad_switch()`/`hmanno()`

We also provide `quad_switch()`/`hmanno()` (heatmap annotation) which integrates `quad_active()` and `quad_anno()` into one function for ease of use. Feel free to use any of these functions to streamline your annotation process.

```
ggheatmap(small_mat) +
    # we set the active context to the top annotation
    quad_switch("t") +
    # we split the observations into 3 groups by hierarchical clustering
    align_dendro(k = 3L) +
    # remove any active annotation
    quad_switch() +
    # set fill color scale for the heatmap body
    scale_fill_viridis_c() +
    layout_title("quad_switch()")
#> > heatmap built with `geom_tile()`
```

quad_switch()
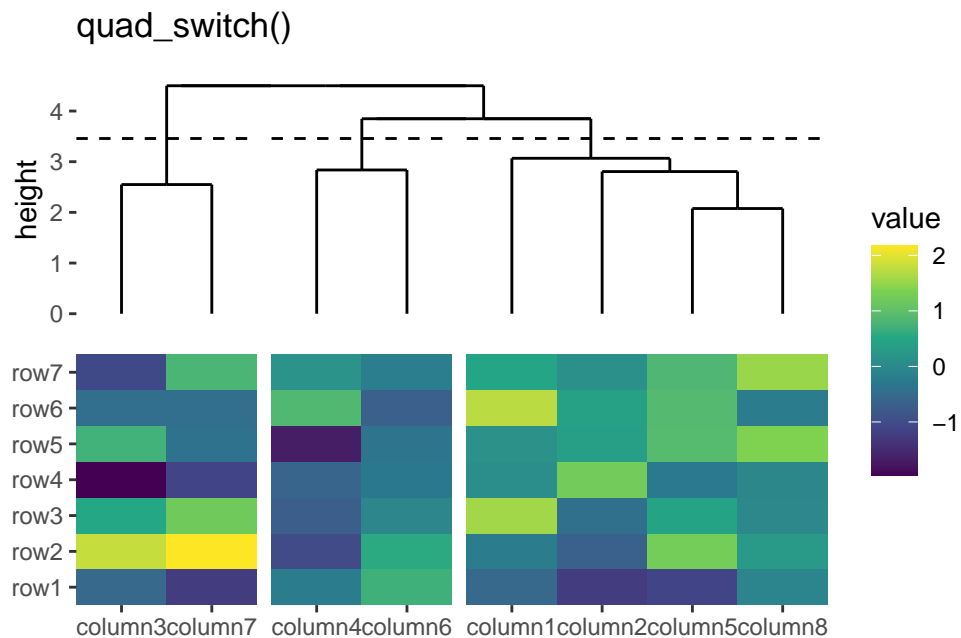

```
ggheatmap(small_mat) +
    # we set the active context to the top annotation
    hmanno("t") +
    # we split the observations into 3 groups by hierarchical clustering
    align_dendro(k = 3L) +
    # remove any active annotation
    hmanno() +
    # set fill color scale for the heatmap body
    scale_fill_viridis_c()+
    layout_title("hmanno()")
#> > heatmap built with `geom_tile()`
```
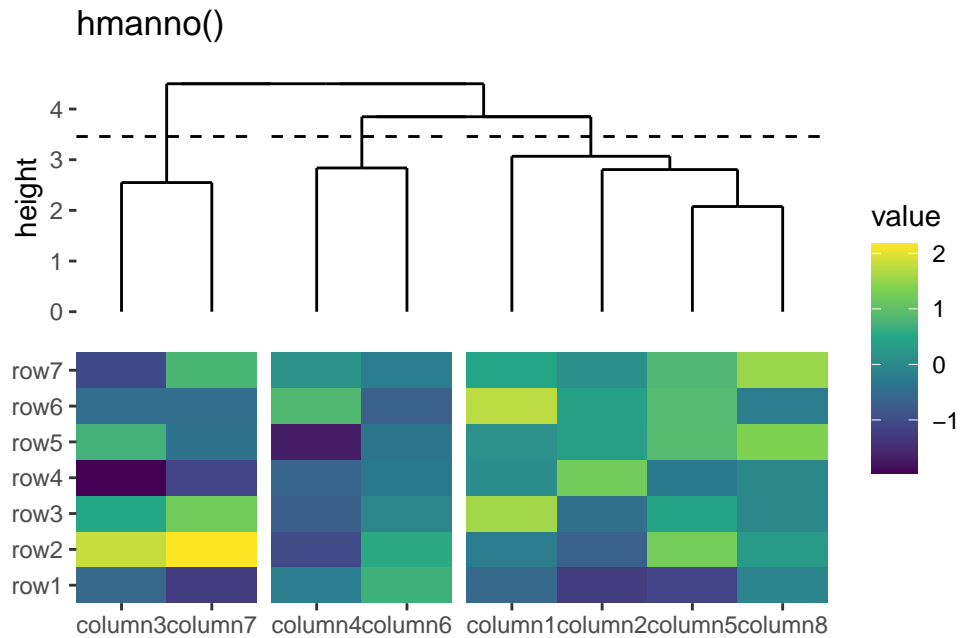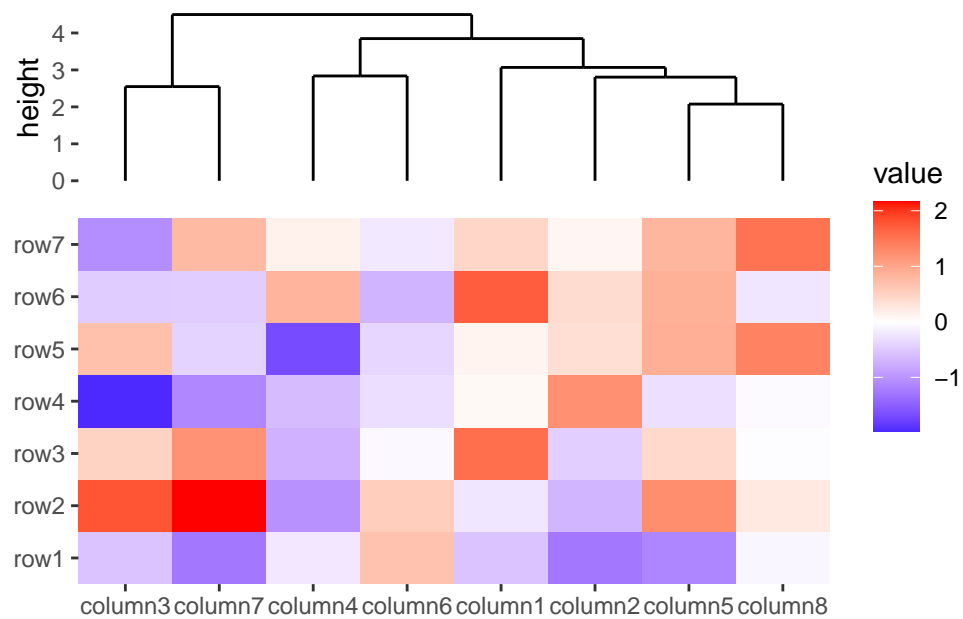
3.8 Plot Size

3.8.1 Heatmap Body Size

You can specify the relative sizes of the heatmap body using the `width` and `height` arguments in the `ggheatmap()` function.

```
ggheatmap(small_mat, height = 2) +
    anno_top() +
    align_dendro()
#> > heatmap built with `geom_tile()`
```

Alternatively, the `quad_active()` function allows you to control the heatmap body sizes.

```
ggheatmap(small_mat) +
    quad_active(height = 2) +
    anno_top() +
    align_dendro()
#> > heatmap built with `geom_tile()`
```
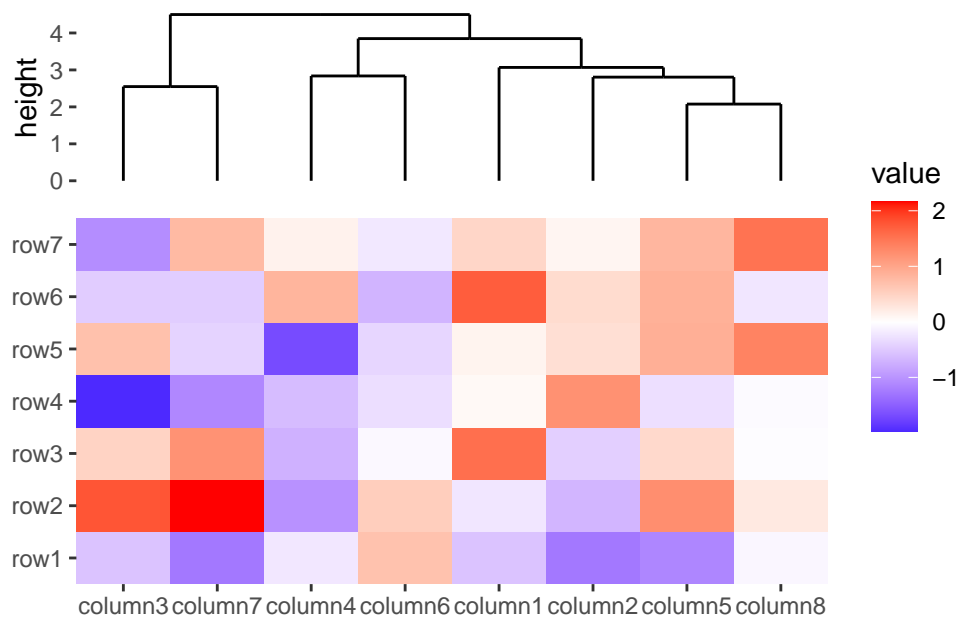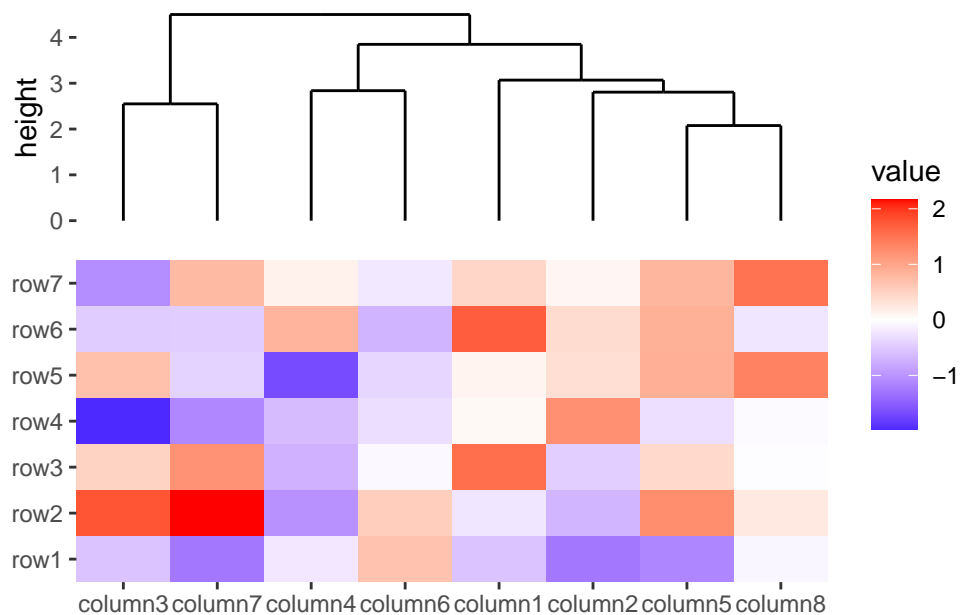
3.8.2 Annotation Stack Size

The `quad_anno()` function allows you to control the total annotation stack size. The `size` argument controls the relative width (for left and right annotations) or height (for top and bottom annotations) of the whole annotation stack.

```
ggheatmap(small_mat) +
    anno_top(size = 1) +
    align_dendro()
#> > heatmap built with `geom_tile()`
```
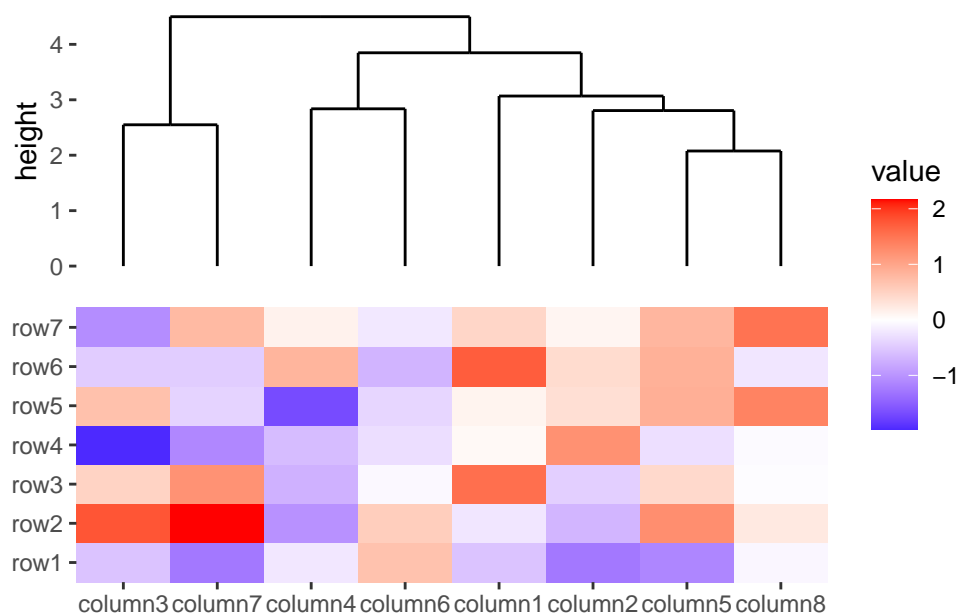
You can also specify it as an absolute size using `unit()`:

```
ggheatmap(small_mat) +
    anno_top(size = unit(30, "mm")) +
    align_dendro()
#> > heatmap built with `geom_tile()`
```

Note that the size of an individual plot (**?@sec-stack-layout__plot__size**) does not affect the total annotation stack size. You must adjust the annotation size using the method described above.

```
ggheatmap(small_mat) +
    anno_top() +
    align_dendro(size = unit(30, "mm")) +
    layout_title("plot size")
#> > heatmap built with `geom_tile()`
```

plot size
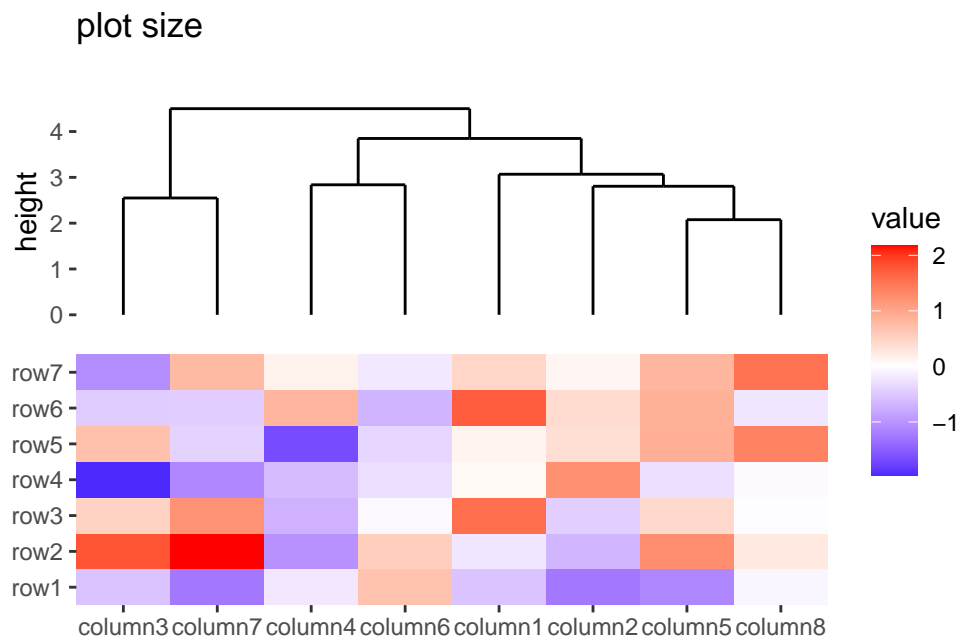

```
ggheatmap(small_mat) +
    anno_top(size = unit(30, "mm")) +
    align_dendro() +
    layout_title("annotation size")
#> > heatmap built with `geom_tile()`
```
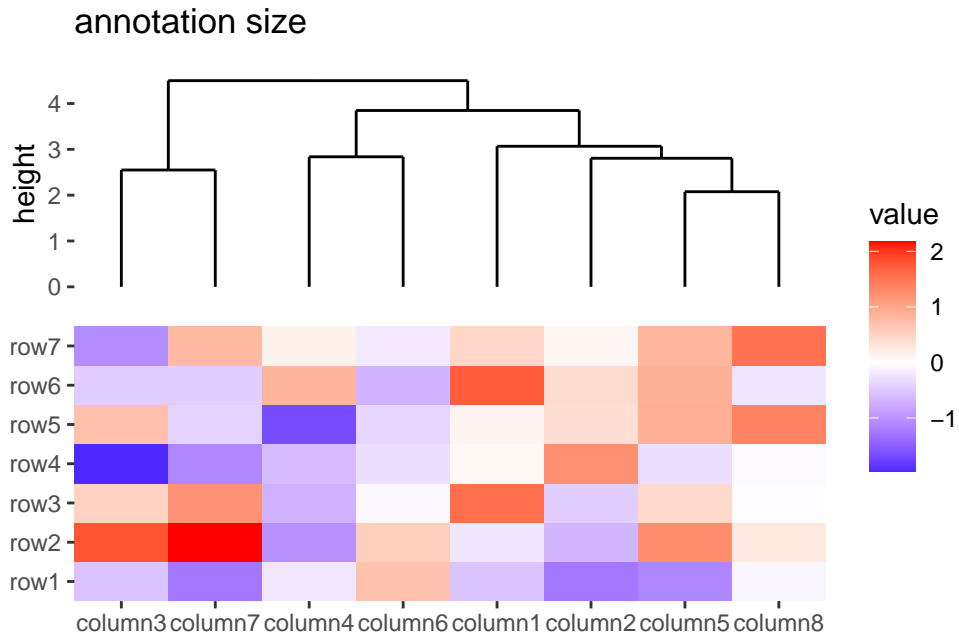
In this chapter, we explored the usage of heatmap layout. These features provide a strong foundation for visualizing matrix-based data in a structured way. However, as your visualization needs grow more complex, the ability to further customize and fine-tune the layout becomes essential.

In the next chapter, we will dive into the Layout Customize functionalities, where you can gain full control over your plot's layout.

4 Layout customize

For layouts that can align observations, the package provides a suite of `align_*` functions designed to give you precise control over the observations. These functions allow you to reorder observations or partition them into multiple groups.

Currently, there are four key `align_*` functions available for layout customization:

- `align_group`: Group and align plots based on categorical factors.
- `align_order`: Reorder layout observations based on statistical weights or allows for manual reordering based on user-defined ordering index.
- `align_kmeans`: Group observations by k-means clustering results.
- `align_hclust`: Reorder or group observations based on hierarchical clustering.
- `align_reorder`: Reorder observations using an arbitrary statistical function.

Note that these functions do not add plot areas and set the active context, meaning you cannot incorporate `ggplot2` elements directly into these objects. You can inspect the behavior of any `align_*` function by printing it.

```
library(ggalign)
#> Loading required package: ggplot2
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))
```

4.1 `align_group()`

The `align_group()` function allows you to split the observations into groups.

```
set.seed(1234)
ggheatmap(small_mat) +
    anno_top() +
    align_group(sample(letters[1:4], ncol(small_mat), replace = TRUE))
#> > heatmap built with `geom_tile()`
```
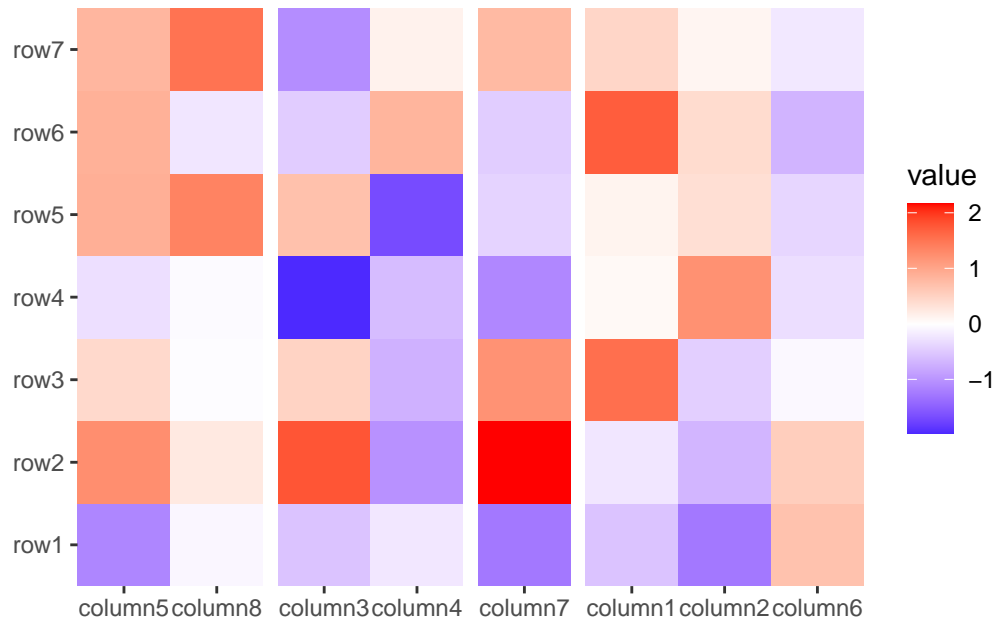
Note that all `align_*` functions which split observations into groups must not break the previous established groups. This means the new groups must nest in the old groups, usually they cannot be used if groups already exist.

```
set.seed(1234)
ggheatmap(small_mat) +
    anno_top() +
    align_group(sample(letters[1:4], ncol(small_mat), replace = TRUE)) +
    align_group(sample(letters[1:5], ncol(small_mat), replace = TRUE))
#> Error in `align()`:
#> ! align_group(sample(letters[1:5], ncol(small_mat), replace = TRUE))
#>   disrupt the previously established panel groups of the top annotation
#>   `stack_align()`
```

4.2 `align_order()`

The `align_order()` function reorder the observations based on the summary weights.

In this example, we order the rows based on their means. By default, the ordering is in ascending order according to the summary weights. You can reverse the order by setting `reverse = TRUE`.

```
ggheatmap(small_mat) +
    anno_left() +
    align_order(rowMeans) +
    layout_title(title = "reverse = FALSE")
#> > heatmap built with `geom_tile()`
```


reverse = FALSE
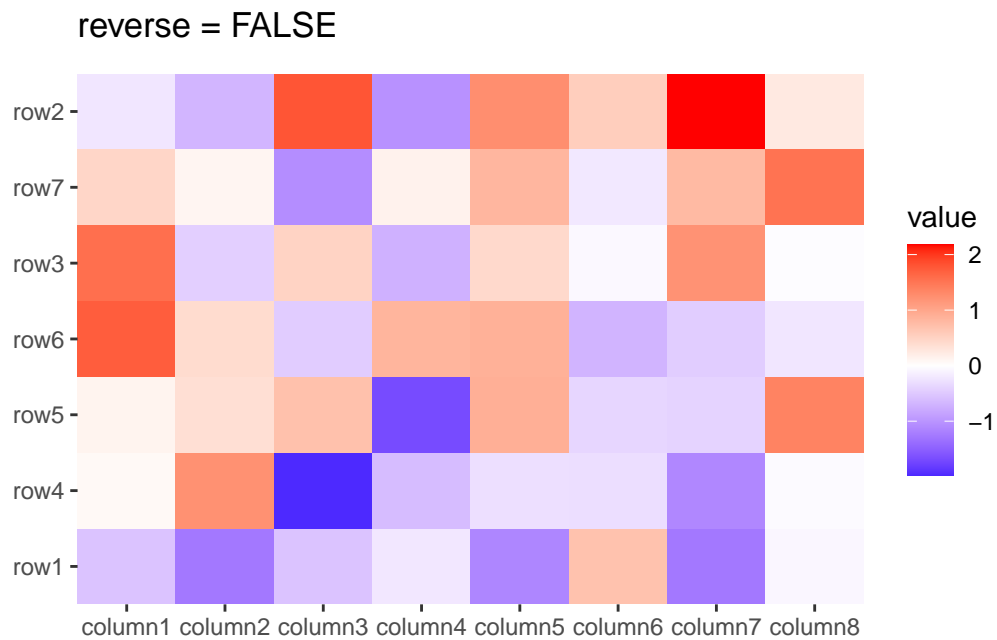
```
ggheatmap(small_mat) +
    anno_left() +
    align_order(rowMeans, reverse = TRUE) +
    layout_title(title = "reverse = TRUE")
#> > heatmap built with `geom_tile()`
```

reverse = TRUE

Additionally, you can provide the ordering integer index or character index directly:

```
set.seed(1234)
my_order <- sample(nrow(small_mat))
print(rownames(small_mat)[my_order])
#> [1] "row4" "row2" "row5" "row7" "row3" "row1" "row6"
```

```
ggheatmap(small_mat) +
    anno_left() +
    align_order(my_order)
#> > heatmap built with `geom_tile()`
```

```
ggheatmap(small_mat) +
    anno_left() +
    align_order(rownames(small_mat)[my_order])
#> > heatmap built with `geom_tile()`
```

Some `align_*` functions also accept a `data` argument. It's important to note that all `align_*` functions treat rows as the observations. This means `NROW(data)` must match the `number of observations` along the axis used for alignment. The `data` argument can also accept a function (supporting purrr-like lambda syntax), which will be applied to the layout matrix.

As mentioned in Section 3.4, for top and bottom annotations, the data matrix of `quad_layout()`/`ggheatmap()` is transposed to create the annotation `stack_layout()`. Therefore, you can use `rowMeans()` to calculate the mean value across all columns.

```
ggheatmap(small_mat) +
    anno_top() +
    align_order(rowMeans)
#> > heatmap built with `geom_tile()`
```

Some `align_*` functions that reorder observations include an argument called `strict`. This argument is especially useful when previous groups have already been established. If previous groups have been created and `strict = FALSE`, the function will reorder the observations within each group.

```
set.seed(1234)
ggheatmap(small_mat) +
    anno_top() +
    align_group(sample(letters[1:4], ncol(small_mat), replace = TRUE))+
    align_order(rowMeans, strict = FALSE)
#> > heatmap built with `geom_tile()`
```

Note that we always prevent reordering the observations into two different orderings. If you want to apply two different orderings, you should use `cross_layout()` instead, which I'll introduce in a later chapter:

```r
set.seed(1234)
another_order <- sample(ncol(small_mat))
ggheatmap(small_mat) +
    anno_top() +
    align_order(rowMeans) +
    align_order(another_order)
#> Error in `align()`:
#> ! align_order(another_order) disrupt the previously established ordering
#>   index of the top annotation `stack_align()`
```

4.3 `align_kmeans()`

The `align_kmeans()` function split the observations into groups based on k-means clustering.

```r
set.seed(1234)
ggheatmap(small_mat) +
```

```
    anno_top() +
    align_kmeans(3L)
#> > heatmap built with `geom_tile()`
```
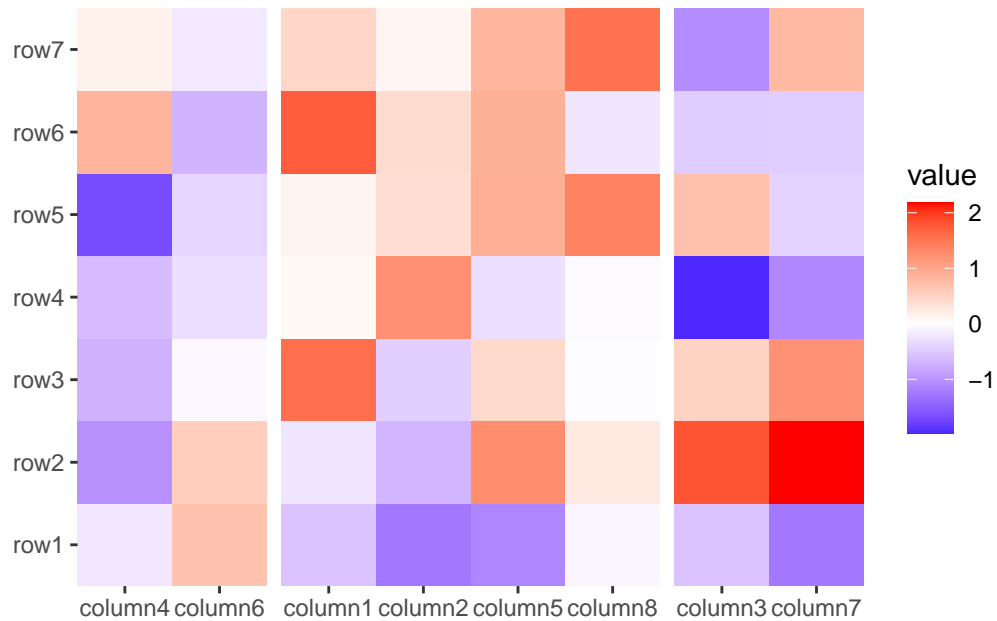

4.4 `align_hclust()`

The `align_hclust()` function is designed to reorder observations and group them based on hierarchical clustering. Unlike `align_dendro()`, however, `align_hclust()` does not add a dendrogram tree to the plot. All the arguments intruduced here can also be used by `align_dendro()`.

```
ggheatmap(small_mat) +
    anno_top() +
    align_hclust()
#> > heatmap built with `geom_tile()`
```
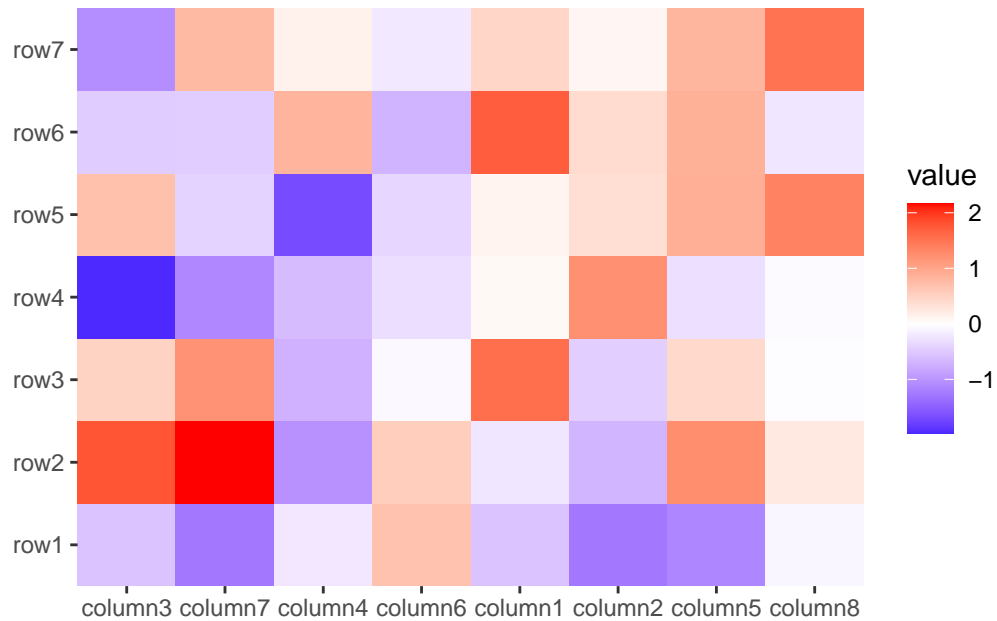
Hierarchical clustering is performed in two steps: calculate the distance matrix and apply clustering. You can use the `distance` and `method` argument to control the builind process.

There are two 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 coorelation coefficient `"pearson"`, `"spearman"` and `"kendall"`. The correlation distance is defined as `1 - cor(x, y, method = distance)`.
- a self-defined function which calculates distance from a matrix. The function should only contain one argument.

```
ggheatmap(small_mat) +
    anno_top() +
    align_hclust(distance = "pearson") +
    layout_title("pre-defined distance method (1 - pearson)")
#> > heatmap built with `geom_tile()`
```

## pre−defined distance method (1 − pearson)
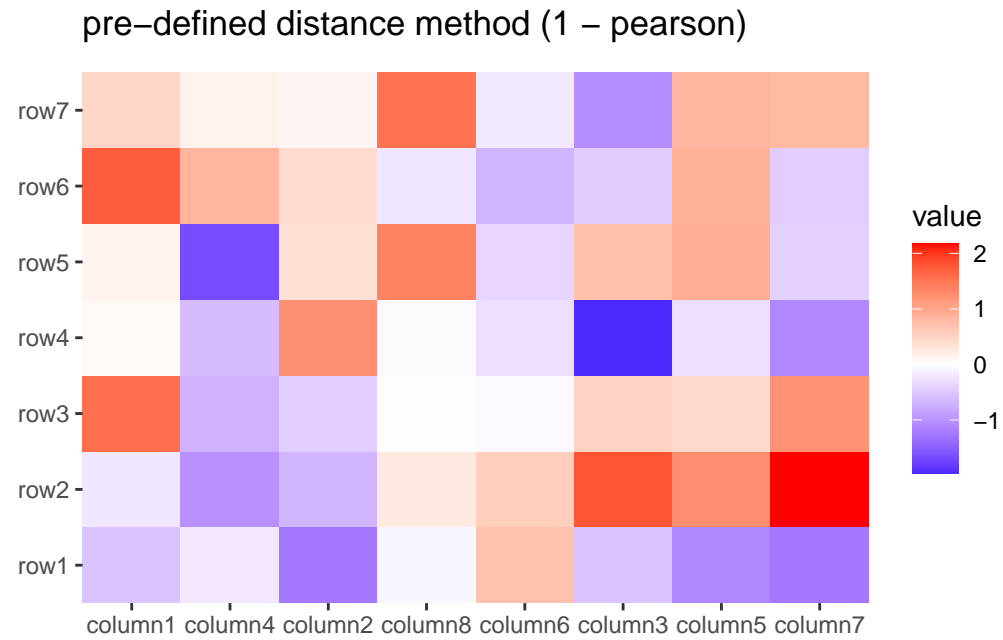

```
ggheatmap(small_mat) +
    anno_top() +
    align_hclust(distance = function(m) dist(m)) +
    layout_title("a function that calculates distance matrix")
#> > heatmap built with `geom_tile()`
```

a function that calculates distance matrix

Method to perform hierarchical clustering can be specified by `method`. Possible methods are those supported in `hclust()` function. And you can also provide a self-defined function, which accepts the distance object and return a `hclust` object.

```
ggheatmap(small_mat) +
    anno_top() +
    align_hclust(method = "ward.D2")
#> > heatmap built with `geom_tile()`
```
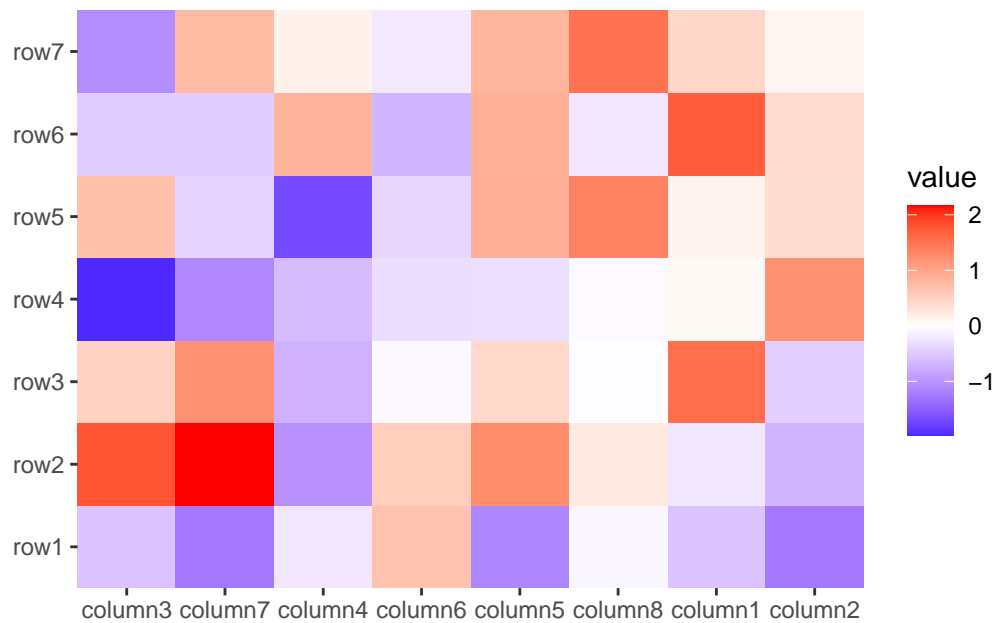
You can specify `k` or `h` to split observations into groups, which work similarly to `cutree()`:

```
ggheatmap(small_mat) +
    anno_top() +
    align_hclust(k = 3L)
#> > heatmap built with `geom_tile()`
```
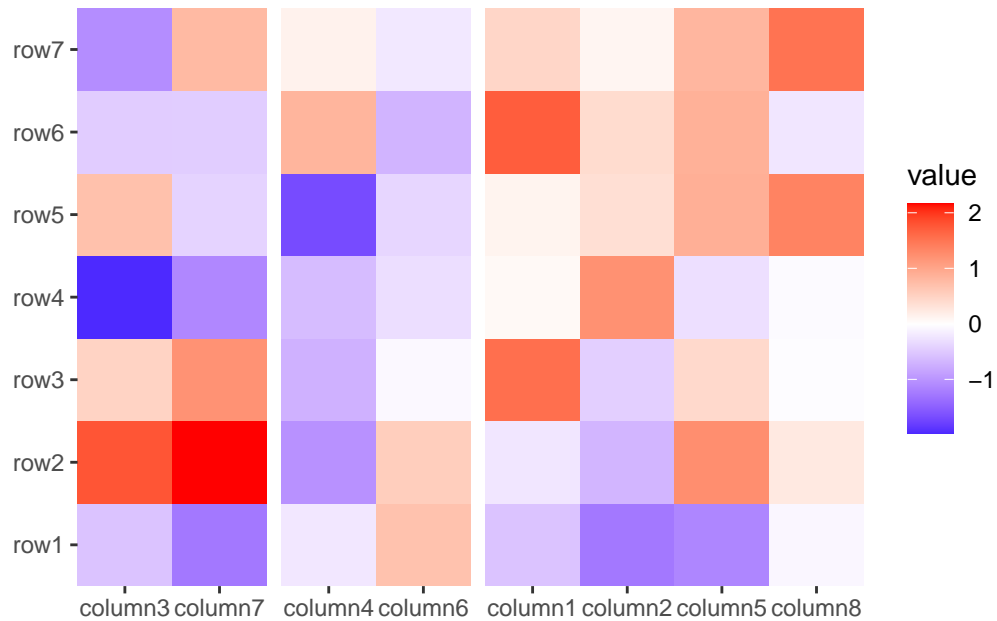
You can reorder the dendrogram based on the mean values of the observations by setting
`reorder_dendrogram = TRUE`.

```
h1 <- ggheatmap(small_mat) +
    anno_top() +
    align_hclust(k = 3, reorder_dendrogram = TRUE) +
    layout_title("reorder_dendrogram = TRUE")
h2 <- ggheatmap(small_mat) +
    anno_top() +
    align_hclust(k = 3) +
    layout_title("reorder_dendrogram = FALSE")
align_plots(h1, h2)
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```
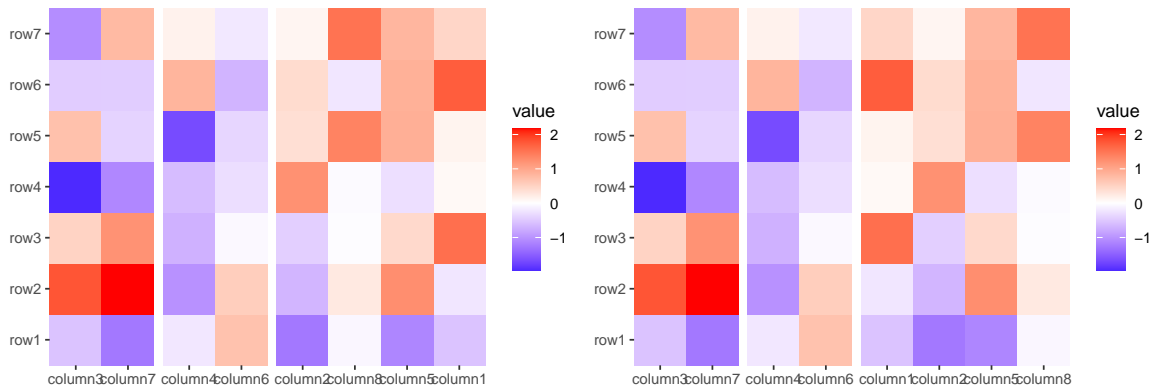
In this example, we use `align_plots()` to arrange the layouts, which functions similarly to `cowplot::align_plots()` and `patchwork::wrap_plots()`. However, unlike those, `align_plots()` can be directly used with `quad_layout()` and `stack_layout()`, ensuring proper alignment by plot panel. Additionally, `align_plots()` can align `pheatmap` and `ComplexHeatmap` objects, though they won't align by panel area in the same way as ggplot2 plots.

`align_hclust()` can also perform clustering between groups, meaning it can be used even if there are existing groups present in the layout, in this way, you cannot specify `k` or `h`:

```
set.seed(3L)
column_groups <- sample(letters[1:3], ncol(small_mat), replace = TRUE)
ggheatmap(small_mat) +
    anno_top() +
    align_group(column_groups) +
    align_hclust()
#> > heatmap built with `geom_tile()`
```
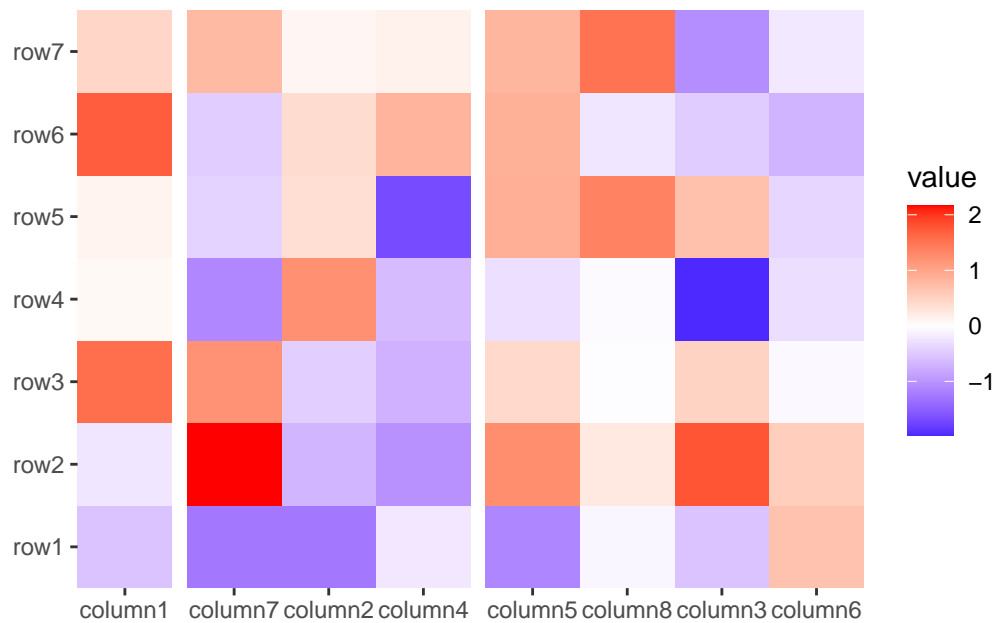
You can reorder the groups by setting `reorder_group = TRUE`, which reorders the hierarchical clusters based on the group tree:

```
ggheatmap(small_mat) +
    anno_top() +
    align_group(column_groups) +
    align_hclust(reorder_group = TRUE)
#> > heatmap built with `geom_tile()`
```
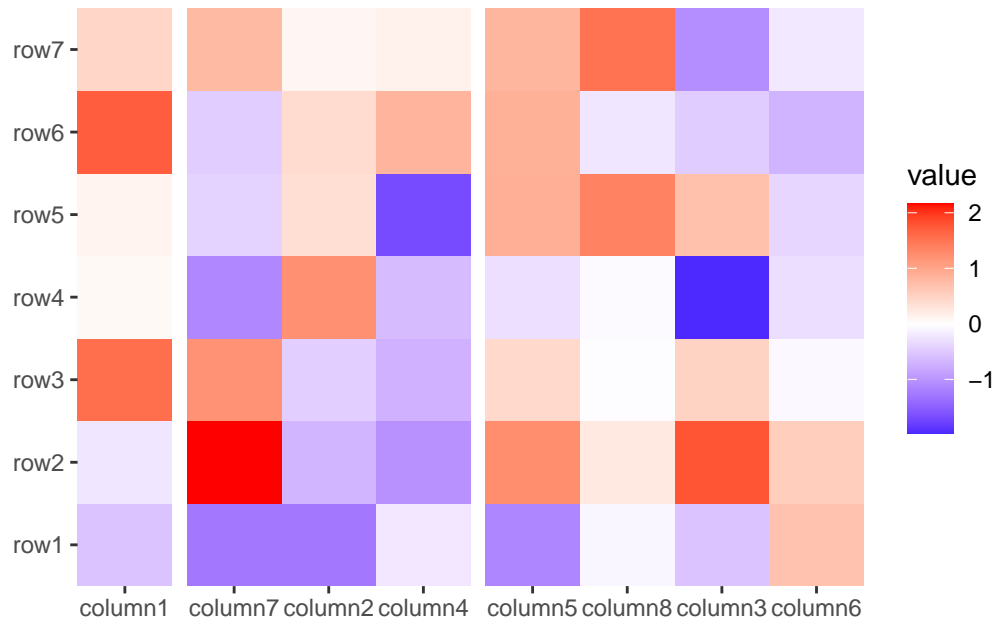
If you specify `k` or `h`, this will always turn off sub-clustering. The same principle applies to `align_hclust()`, where new groups must be nested within the previously established groups.

```
ggheatmap(small_mat) +
    anno_top() +
    align_group(column_groups) +
    align_hclust(k = 2L)
#> Error in `align()`:
#> ! align_hclust(k = 2L) disrupt the previously established panel groups
#>   of the top annotation `stack_align()`
```

4.5 `align_reorder()`

The `align_reorder()` function enables the reordering of observations based on a specified statistical function. This function accepts a data argument and computes a statistic that determines the new ordering of the observations. The resulting order is extracted using the `order2()` function.

Foe example, you can also use hierarchical clustering to reorder the observations like this:

```
ggheatmap(small_mat) +
    anno_left() +
    align_reorder(hclust2)
#> > heatmap built with `geom_tile()`
```
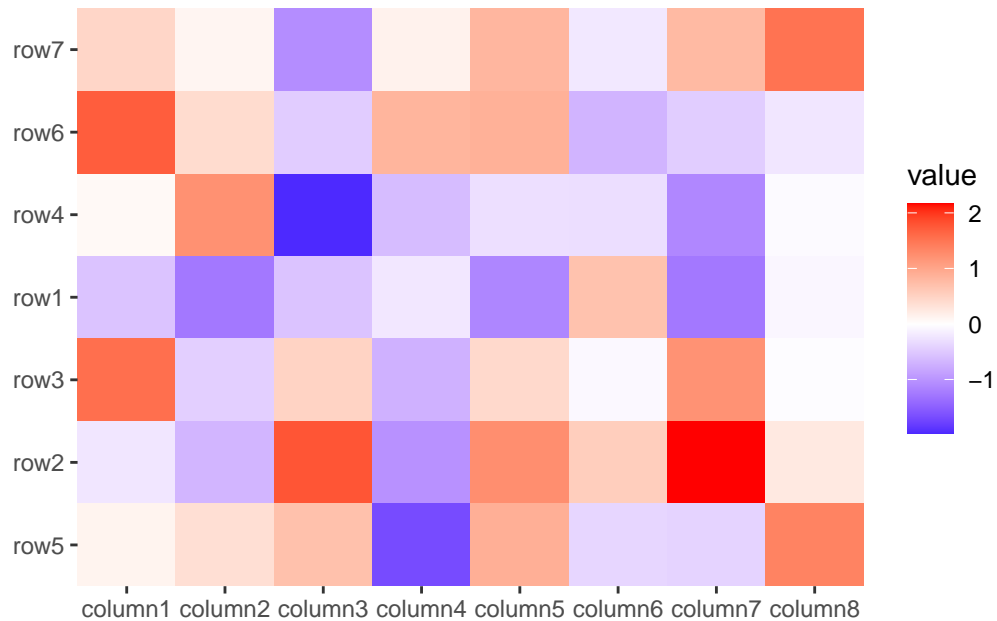

Here, `hclust2()` is a special function used by `align_hclust()` to calculate the distance matrix and perform hierarchical clustering.

In addition to hierarchical clustering, the `align_reorder()` function can also handle objects from the seriation package. For example, you can use the `seriate()` function with a method like `"BEA_TSP"` to reorder the observations based on a specific seriation algorithm:

```
ggheatmap(small_mat) +
    anno_left() +
    align_reorder(seriation::seriate, method = "BEA_TSP", data = abs)
#> > heatmap built with `geom_tile()`
```
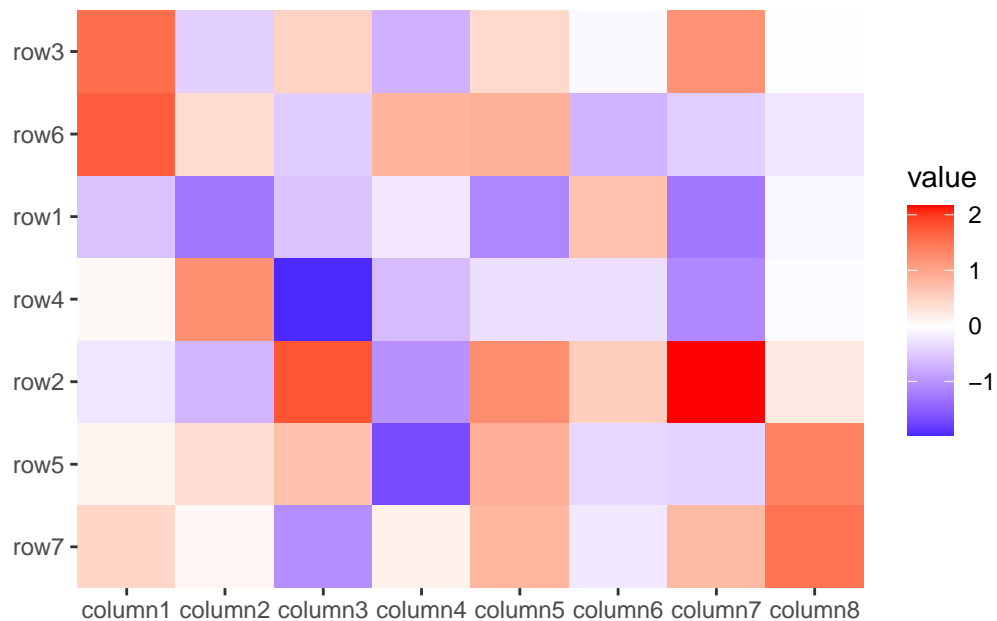
This approach gives you the flexibility to apply different statistical or algorithmic methods for reordering observations, which can be especially useful for exploring complex patterns in data.

With the `align_*` functions in place, we've covered how to reorder and group observations within a heatmap layout. Now, having set the stage for proper observation alignment, we can move on to the next essential step: `plot initialize`. In the following chapter, we'll dive deeper into how to set up your plots within the layout.

5 Plot initialize

There are three primary functions for adding plots:

- `align_gg()`/`ggalign()`: Create a ggplot object and aligns the observation.
- `free_gg()`/`ggfree()`: Create a ggplot object without aligning the observation.
- `align_dendro()`: Create a ggplot object of dendrogram tree, and align the observation. In addition, it can reorder and group the observations.

`ggalign()` and `ggfree()` are aliases for `align_gg()` and `free_gg()`, respectively. All three functions will set the active plot when added to the layout.

Both `ggalign()` and `align_dendro()` can only be used with a `stack_layout()` that aligns observations (`stack_align()`), whereas `ggfree()` can be added to a `stack_layout()` that does not align the observations (`stack_free()`).

```
library(ggalign)
#> Loading required package: ggplot2
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))
```

5.1 `align_gg()`/`ggalign()`

`ggalign()` is similar to `ggplot` in that it initializes a `ggplot` data and `mapping`. `ggalign()` allowing you to provide data in various formats, including matrices, data frames, or simple vectors. By default, it will inherit from the layout. If a function, it will apply with the layout matrix.

`ggalign()` always applies a default mapping for the axis of the data index in the layout. This mapping is `aes(y = .data$.y)` for horizontal stacking (including left and right quad layout annotation) and `aes(x = .data$.x)` for vertical stacking (including top and bottom quad layout annotation).

The data in the underlying `ggplot` object will contain following columns:

- `.panel`: the panel for the aligned axis. It means `x-axis` for vertical stack layout (including top and bottom annotation), `y-axis` for horizontal stack layout (including left and right annotation).

- `.x` or `.y`: the `x` or `y` coordinates.

- `.names` and `.index`: A factor of the names (only applicable when names exists) and an integer of index of the original data.

- `.row_names` and `.row_index`: the row names and an integer of row index of the original matrix (only applicable if `data` is a `matrix`).

- `.column_names` and `.column_index`: the column names and column index of the original matrix (only applicable if `data` is a `matrix`).

- `value`: the actual value (only applicable if `data` is a `matrix` or atomic vector).

  It is recommended to use `.x`/`.y`, or `.names` as the `x/y` mapping.

```
ggheatmap(small_mat) +
    anno_top() +
    ggalign(data = rowSums) +
    geom_point(aes(y = value))
#> > heatmap built with `geom_tile()`
```
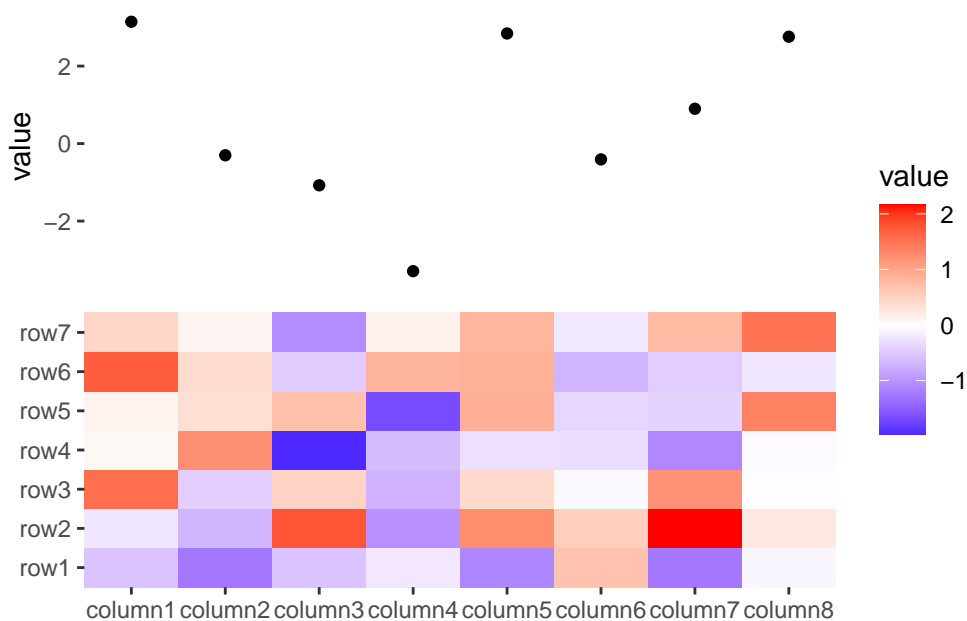
In the case where the input data is already a data frame, following columns (`.x`/`.y`, `.names`, `.index`, and `.panel`) will be added to the data frame if applicable. Note, `NROW(data)` must match the **number of observations** along the axis used for alignment.

```
my_df <- mtcars[seq_len(ncol(small_mat)), ]
ggheatmap(small_mat) +
    anno_top() +
    ggalign(data = my_df) +
    geom_point(aes(y = cyl))
#> > heatmap built with `geom_tile()`
```
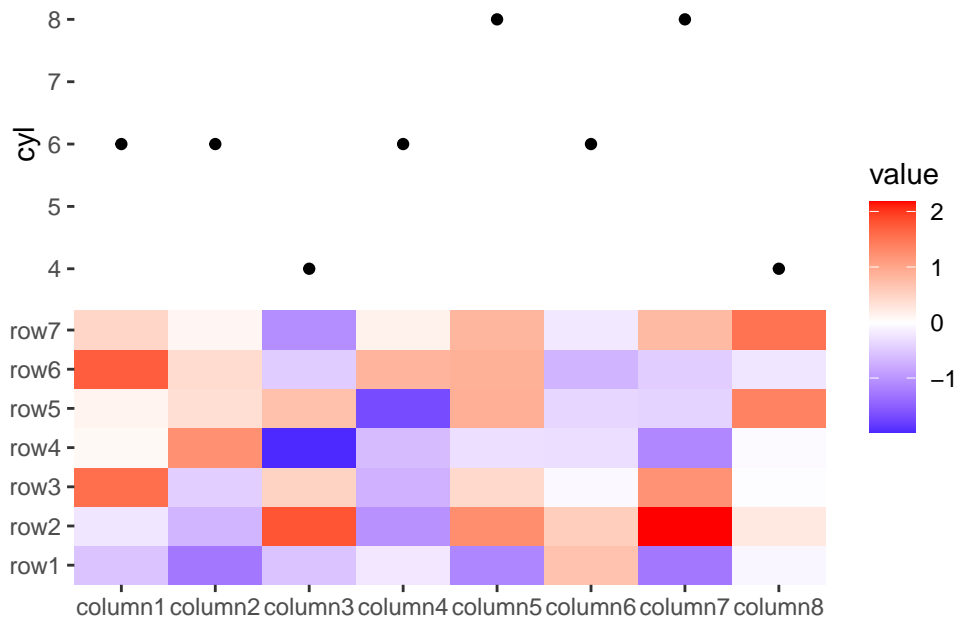

You may wonder if the strict data requirements will prevent us from preparing the data for plotting. Don't worry! In the later chapters, I'll introduce powerful way to prepare data for your plots.

If `data = NULL`, the data in the underlying `ggplot` object contains following columns:

- `.panel`: the panel for the axis used for alignment.
- `.index`: the index of the original layout data.
- `.x` or `.y`: the x or y coordinates

You can use it to integrate additional elements, such as block annotation or customized panel title, into your layout.

```
ggheatmap(small_mat) +
    anno_top(size = unit(1, "cm")) +
    align_kmeans(centers = 3L) +
    ggalign(data = NULL) +
    geom_tile(aes(y = 1L, fill = .panel, color = .panel))
#> > heatmap built with `geom_tile()`
```
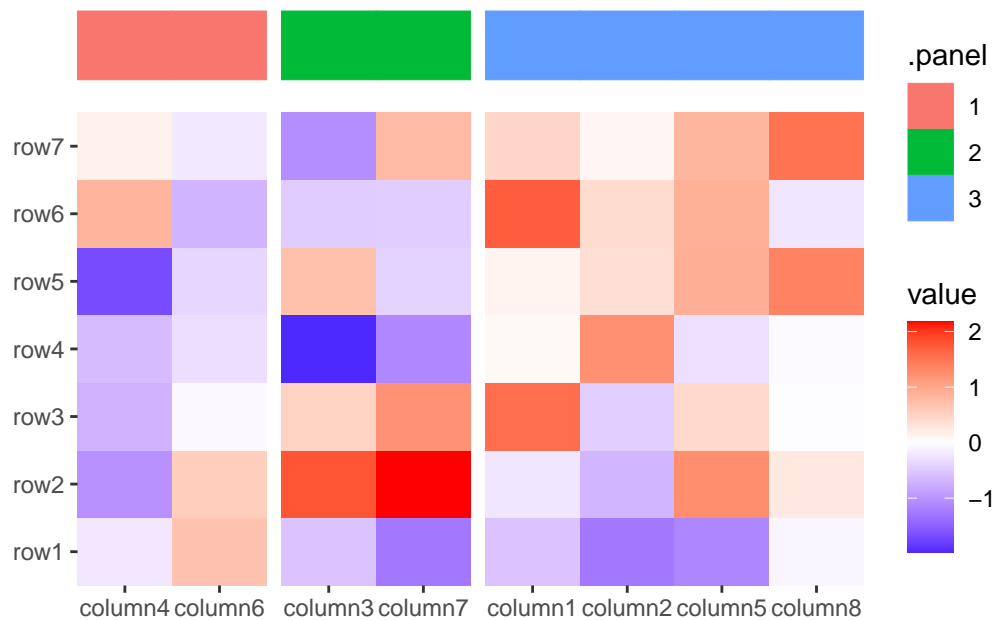

5.1.1 Cross panel sumamry

When used in a `quad_layout()`, and the data is inherited from the `quad_layout()`, a special column `.extra_panel` will be added if applicable, which is the panel information for column (for left or right annotation) or row (for top or bottom annotation). This is useful if you want to create summary plot using another axis panel groups. In such cases, it's often necessary to disable the automatic setting of limits (`limits = FALSE` in `ggalign()`).

```
set.seed(1234)
ggheatmap(small_mat) +
    anno_right() +
    align_kmeans(2) +
    anno_top(size = 0.5) +
    ggalign(limits = FALSE) +
```

```
    geom_boxplot(aes(.extra_panel, value, fill = .extra_panel)) +
    scale_fill_brewer(palette = "Dark2", name = "row groups")
#> > heatmap built with `geom_tile()`
```
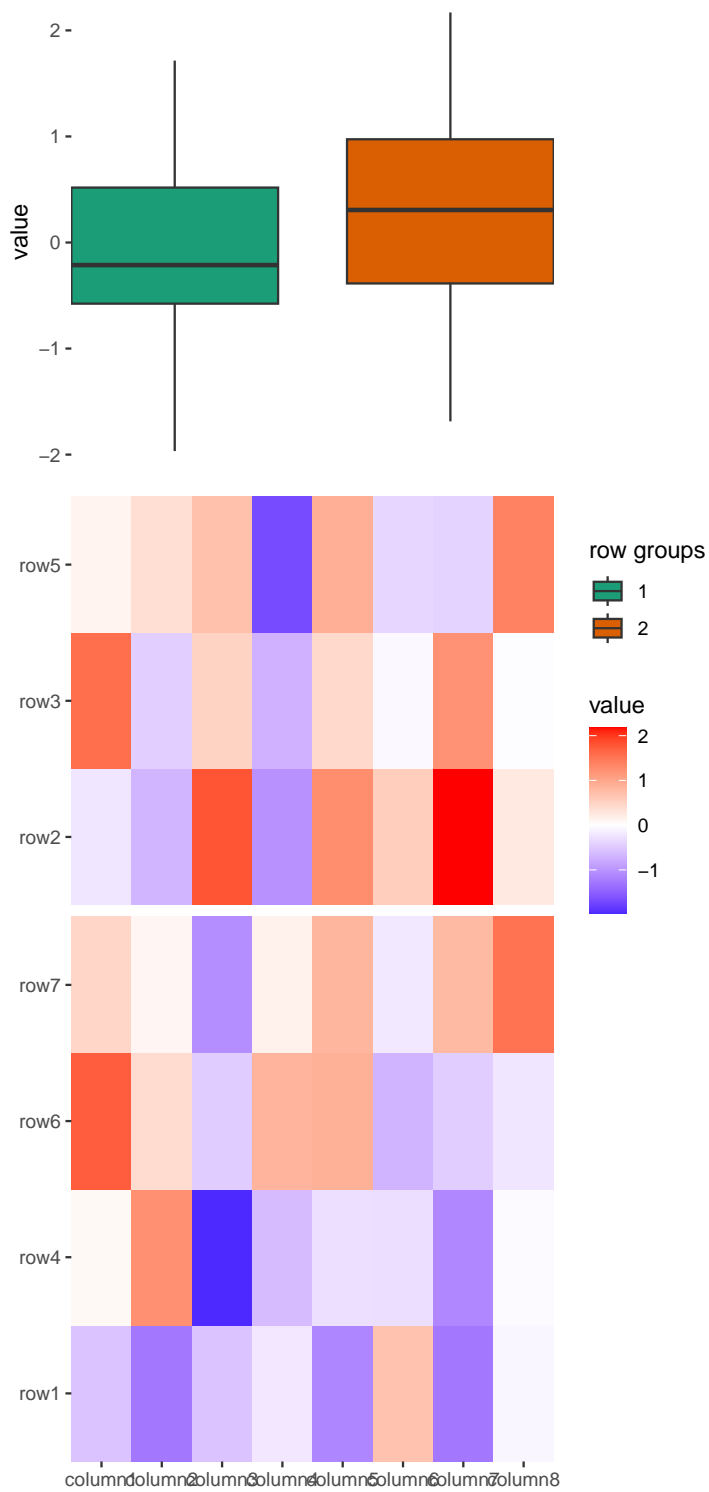
This approach replicates the functionality of ComplexHeatmap::anno_summary(), but is versatile enough to be used with any heatmap, not just single-column or single-row heatmaps.

5.2 `free_gg()`/`ggfree()`

The `ggfree()` function allows you to incorporate a ggplot object into your layout. Unlike `align_gg()`, which aligns every axis value precisely, `free_gg()` focuses on layout integration without enforcing strict axis alignment.

Internally, the function uses `fortify_data_frame()` to transform the input into a data frame. For matrices, it converts them into a long-formatted data frame. Note that `ggfree()` does not provide a default aesthetic mapping, which contrasts with `ggalign()`. You will need to manually provide the default mappings.

```
ggheatmap(small_mat) +
    anno_top() +
    ggfree(mtcars, aes(wt, mpg)) +
    geom_point()
#> > heatmap built with `geom_tile()`
```
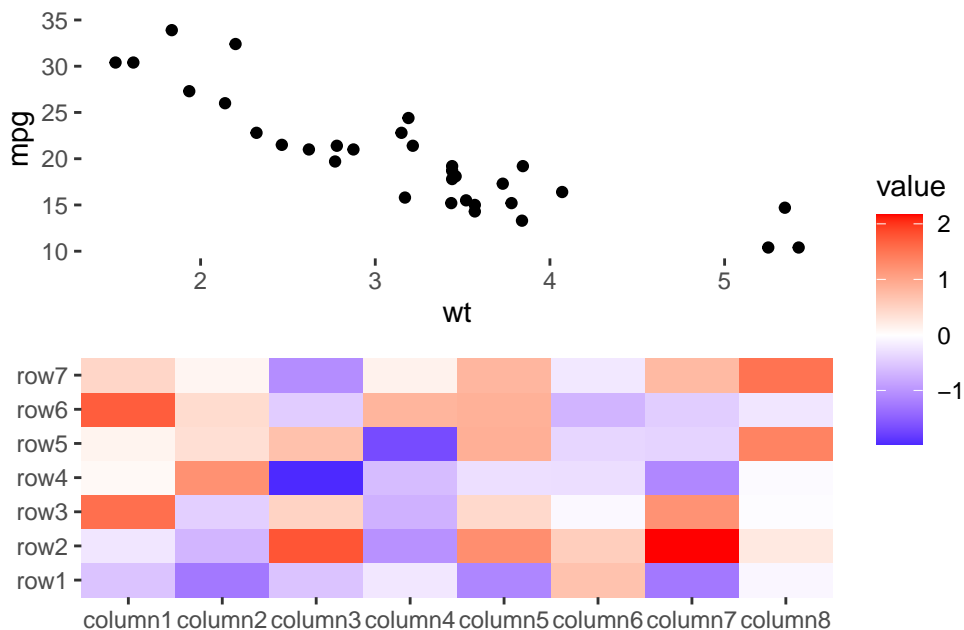

Alternatively, you can directly input the ggplot object.

```
ggheatmap(small_mat) +
    anno_top() +
    ggfree(ggplot(mtcars, aes(wt, mpg))) +
    geom_point()
#> > heatmap built with `geom_tile()`
```
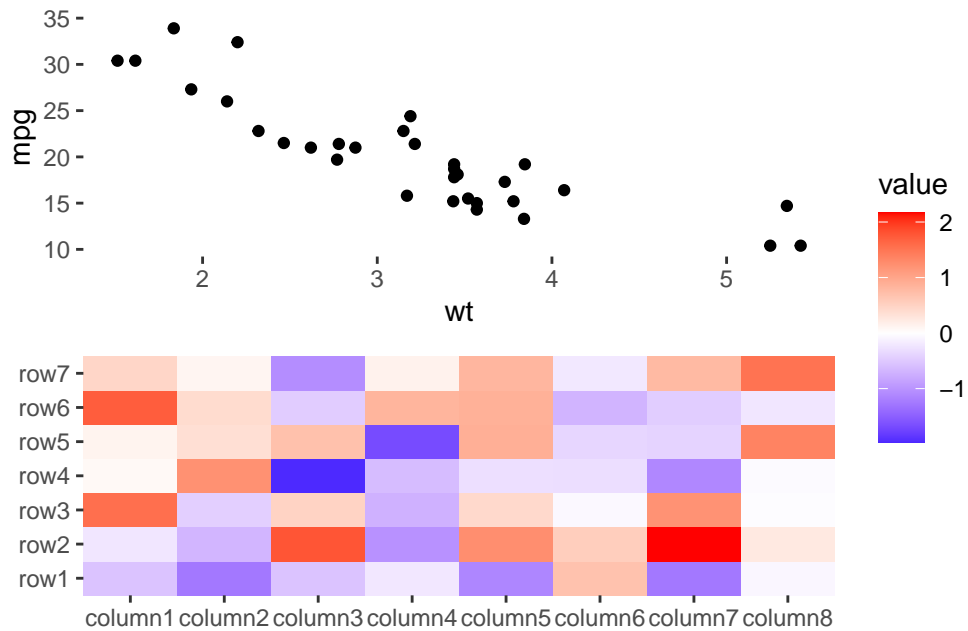

You can also add the `ggplot` object directly without using `ggfree()`. However, doing so will limit control over the plot (like plot area `size`, and `active` components):

```
ggheatmap(small_mat) +
    anno_top() +
    ggplot(mtcars, aes(wt, mpg)) +
    geom_point()
#> > heatmap built with `geom_tile()`
```
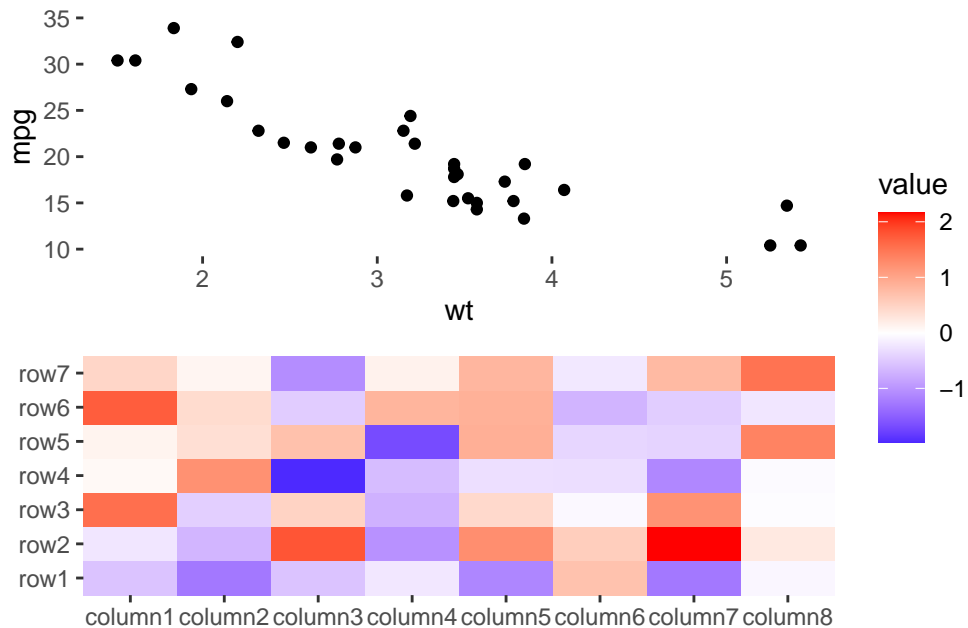
5.3 `ggwrap()` and `inset()`

The `ggwrap()` function allows you to wrap objects that can be converted into a grob, turning them into a `ggplot` for plotting. Further you can still add ggplot elements like title, subtitle, tag, caption, and geoms using the same approach as with normal ggplots (using `ggtitle()`, `labs()`, `geom_*()`) as well as styling using `theme()`. This enables you to pass these wrapped objects into `ggfree()`.

```
library(grid)
ggheatmap(small_mat) +
    anno_top() +
    # `ggwrap()` will create a `ggplot` object, we use `ggfree` to add it into the layout
    ggfree(data = ggwrap(rectGrob(gp = gpar(fill = "goldenrod")), align = "full"))
#> > heatmap built with `geom_tile()`
```
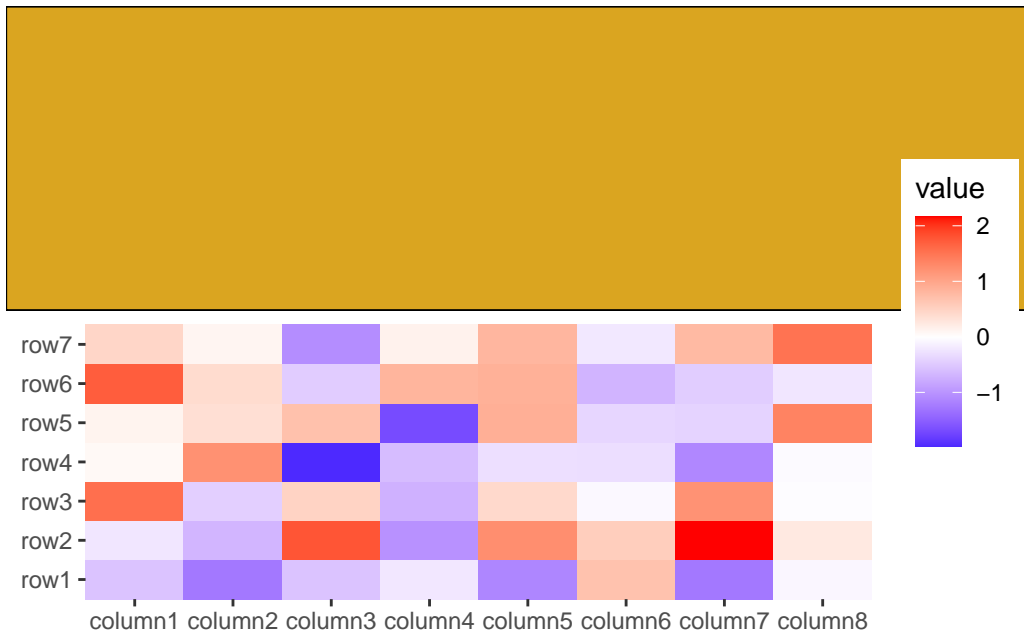
You can also integrate base plots, `pheatmap`, `ComplexHeatmap`, e.g.

Additionally, you can add any graphics as a inset to a ggplot using the `inset()` function. Like `ggwrap()`, `inset()` can accept any object that can be converted to a grob.

```
ggheatmap(small_mat) +
    anno_top() +
    ggfree(data = ggwrap(rectGrob(gp = gpar(fill = "goldenrod")), align = "full")) +
    # we can then add any inset grobs (the same as ggwrap, it can take any objects
    # which can be converted to a `grob`)
    inset(rectGrob(gp = gpar(fill = "steelblue")), align = "panel") +
    inset(textGrob("Here are some text", gp = gpar(color = "black")),
        align = "panel"
    )
#> > heatmap built with `geom_tile()`
```

86
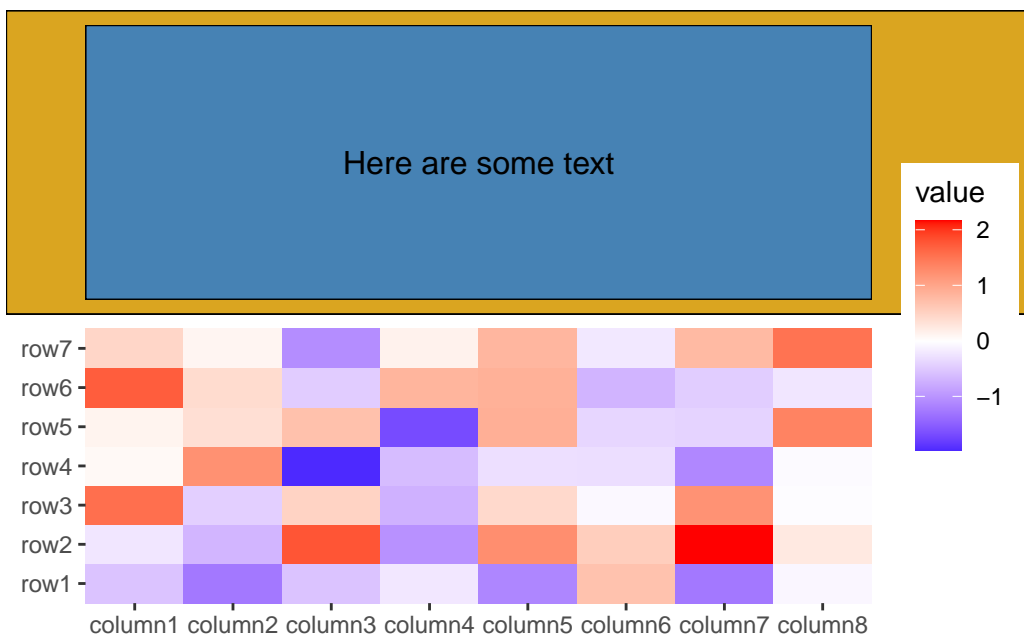
5.4 `align_dendro()`

`align_dendro()` is an extension of `align_hclust()` that adds a dendrogram to the layout. All functions of `align_hclust()` introduced in Section 4.4 can be used with `align_dendro()`. Here, we focus on the plot-related function.

`align_dendro()` will initialize a ggplot object, the data underlying the ggplot object contains the dendrogram `node` data with dendrogram `edge` data attached in a special attribute.

dendrogram `node` and `edge` contains following columns: - `index`: the original index in the tree for the current node - `label`: node label text - `x` and `y`: x-axis and y-axis coordinates for current node or the start node of the current edge. - `xend` and `yend`: the x-axis and y-axis coordinates of the terminal node for current edge. - `branch`: which branch current node or edge is. You can use this column to color different groups. - `leaf`: A logical value indicates whether current node is a leaf.

By default, `plot_dendrogram` is set to `TRUE`, meaning a `geom_segment()` layer will be added using the dendrogram edge data when drawing. Note that this layer is always added to the first.

```
ggheatmap(small_mat) +
    anno_top() +
```
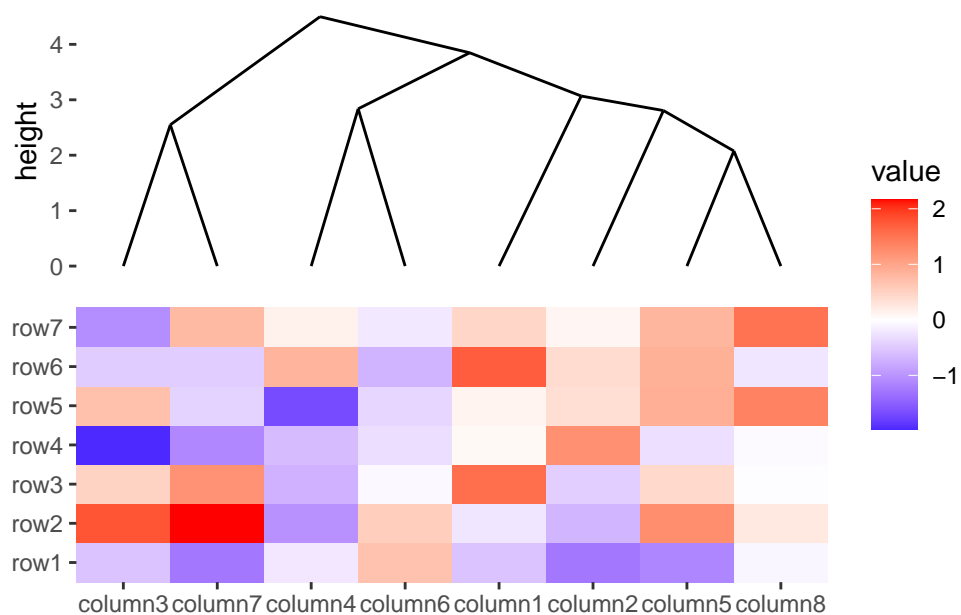
```
    align_dendro()
#> > heatmap built with `geom_tile()`
```
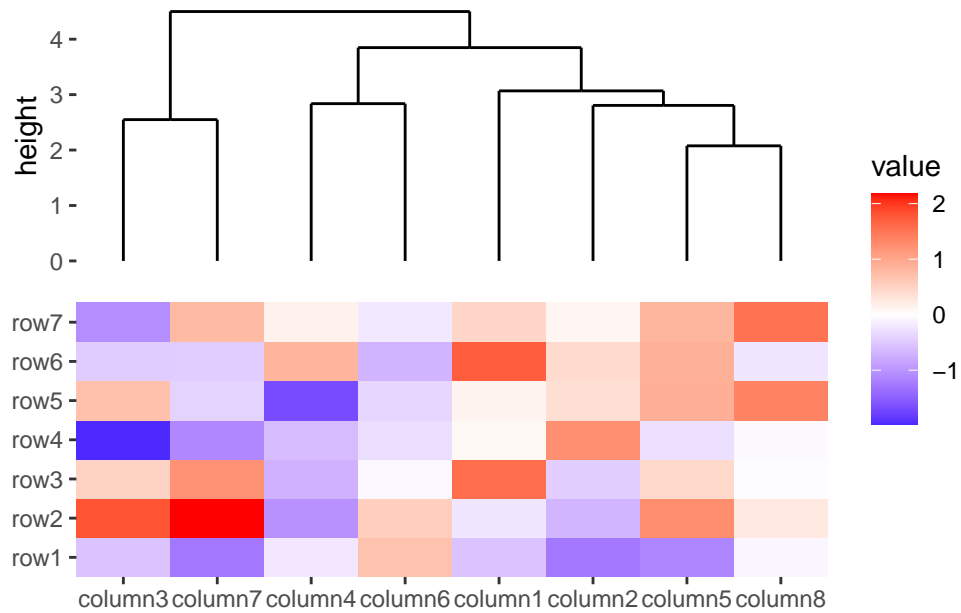

You can choose between two dendrogram types: `"rectangle"` (default) and `"triangle"`. However, if there are any groups in the stack, `"rectangle"` will be used.

```
ggheatmap(small_mat) +
    anno_top() +
    align_dendro(type = "triangle")
#> > heatmap built with `geom_tile()`
```

You can also manually add the dendrogram tree using the edge data by setting `plot_dendrogram = FALSE`. In this case, you can access the dendrogram `edge` data with `ggalign_attr()`. The `edge` data is stored in the edge field of `ggalign_attr()`:

```
ggheatmap(small_mat) +
    anno_top() +
    align_dendro() +
    geom_segment(
        aes(x = .data$x, y = .data$y, xend = .data$xend, yend = .data$yend),
        data = function(x) ggalign_attr(x, "edge")
    )
#> > heatmap built with `geom_tile()`
```
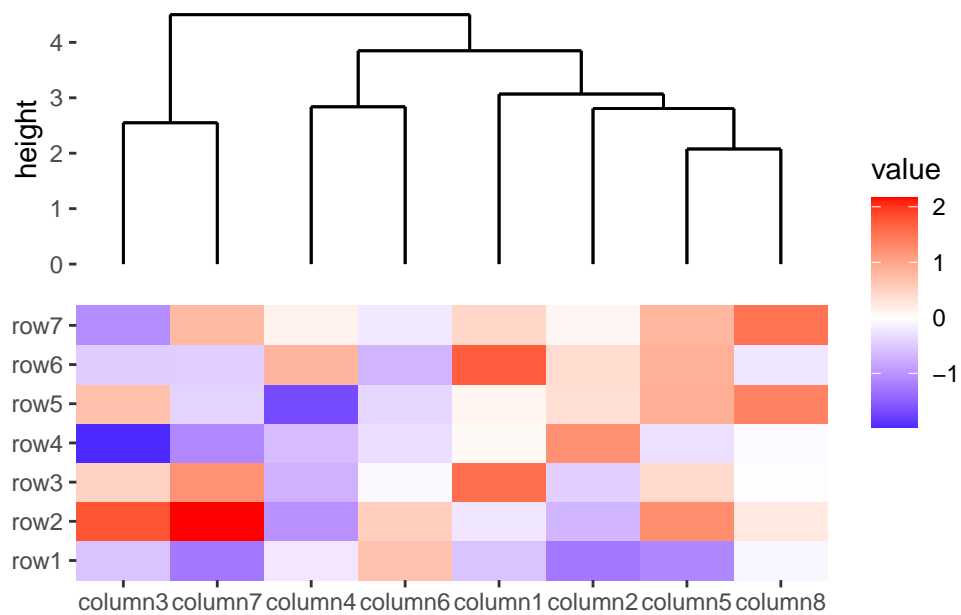
When there are multiple groups, a `branch` column will be available. This can be used to color the nodes or edges based on the group source.

```
ggheatmap(small_mat) +
    anno_top() +
    align_dendro(aes(color = branch), k = 3) +
    geom_point(aes(color = branch))
#> > heatmap built with `geom_tile()`
```
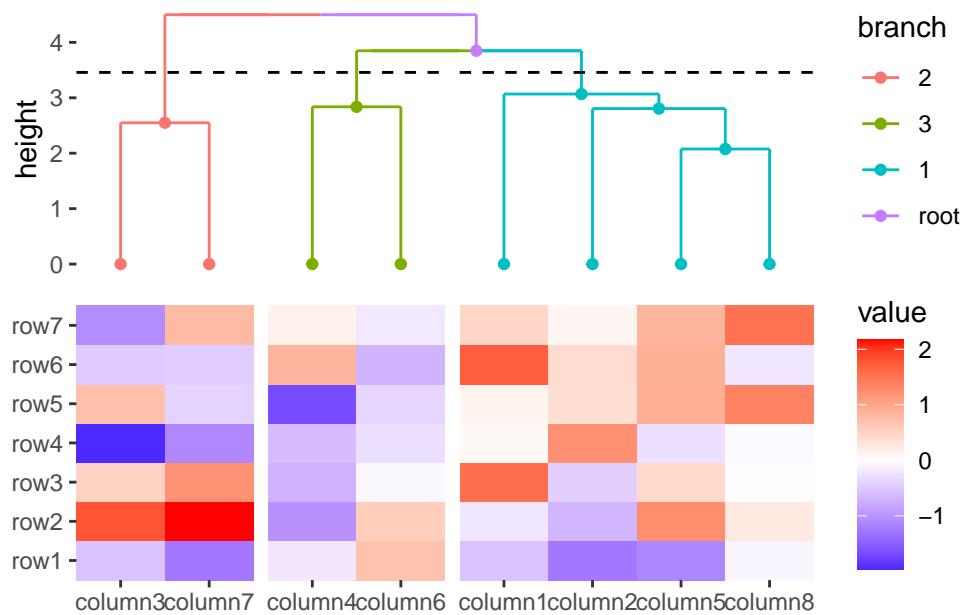
`align_dendro()` will draw dendrogram tree in each group when previous group exists.

```
set.seed(1234)
ggheatmap(small_mat) +
    anno_top() +
    align_kmeans(2) +
    align_dendro(aes(color = branch))
#> > heatmap built with `geom_tile()`
```

You can merge the sub-tree by settting `merge_dendrogram = TRUE`.

```
ggheatmap(small_mat) +
    anno_top() +
    align_kmeans(2) +
    align_dendro(aes(color = branch), merge_dendrogram = TRUE)
#> > heatmap built with `geom_tile()`
```
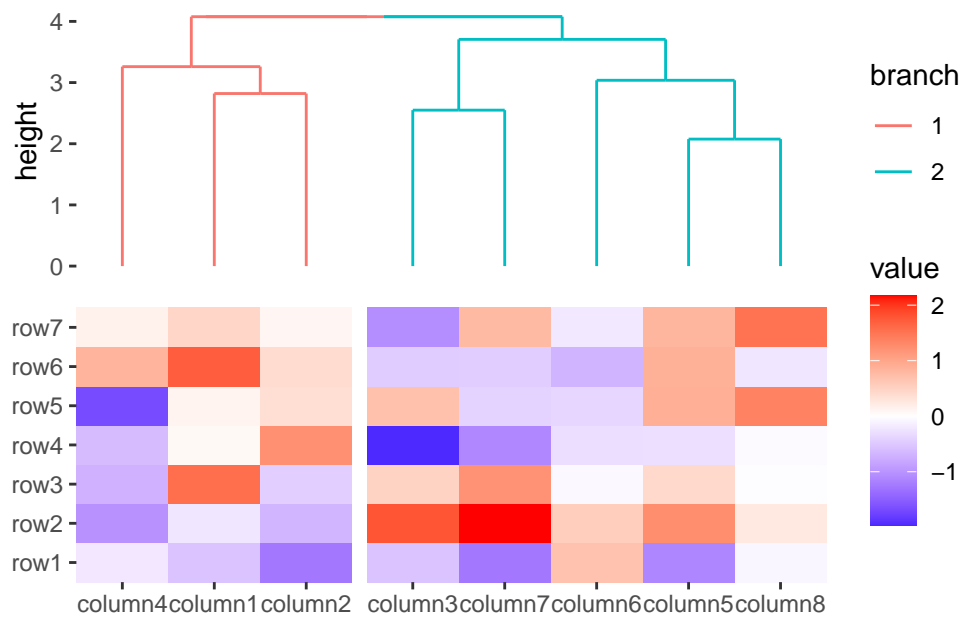
5.5 Plot titles

`ggplot2` only allow add titles in the top or add caption in the bottom. we extends this capability, allowing you to place titles around any border of the plot using the `patch_titles()` function.

```
ggheatmap(small_mat) +
    anno_top() +
    align_dendro(aes(color = branch), k = 3) +
    patch_titles(
        top = "top patch title",
        left = "left patch title",
        bottom = "bottom patch title",
        right = "right patch title"
    )
#> > heatmap built with `geom_tile()`
```
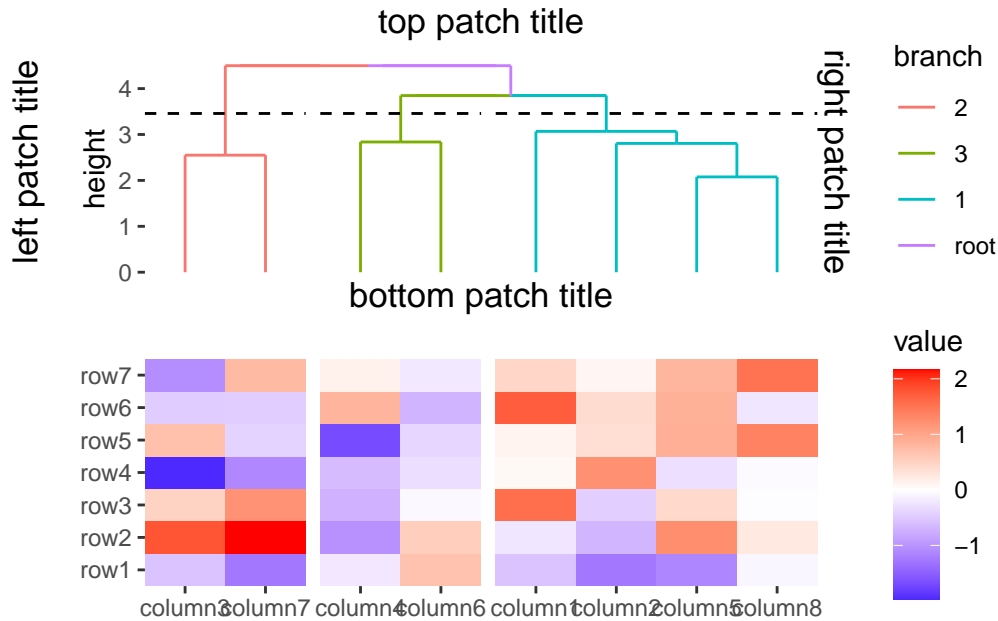
The appearance and alignment of these patch titles can be customized using `ggplot2::theme()`:
- `plot.patch_title`/`plot.patch_title.*`: Controls the text appearance of patch titles. By default, `plot.patch_title` inherit from `plot.title`, and settings for each border will inherit from `plot.patch_title`, with the exception of the `angle` property, which is not inherited. - `plot.patch_title.position`/`plot.patch_title.position.*`: Determines the alignment of the patch titles. By default, `plot.patch_title.position` inherit from `plot.title.position`, and settings for each border will inherit from `plot.patch_title`. The value `"panel"` aligns the patch titles with the plot panels. Setting this to `"plot"` aligns the patch title with the entire plot (excluding margins and plot tags).

```
ggheatmap(small_mat) +
    anno_top() +
    align_dendro(aes(color = branch), k = 3) +
    patch_titles(
        top = "top patch title",
        left = "left patch title",
        bottom = "bottom patch title",
        right = "right patch title"
    ) +
    theme(
        plot.patch_title.position = "plot",
        plot.patch_title = element_text(hjust = 0)
    )
```

```
#> > heatmap built with `geom_tile()`
```
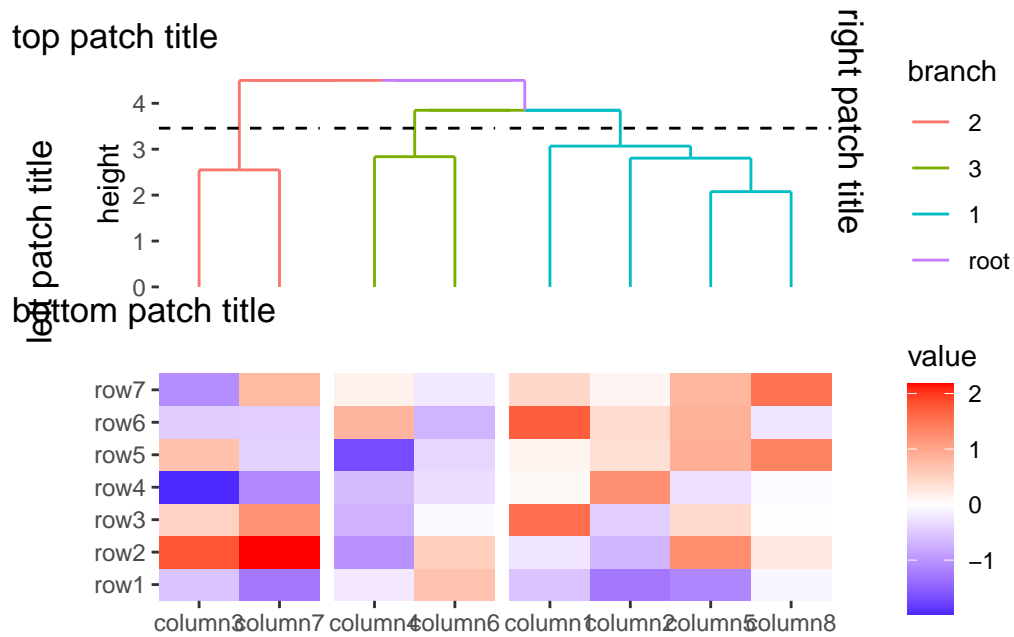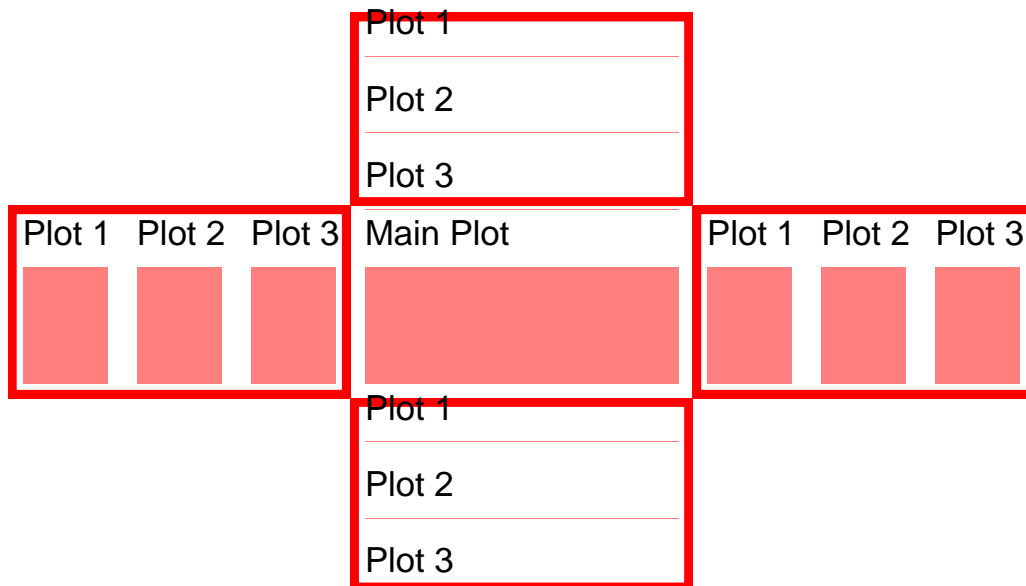

Now, let's move on to the next chapter, where we will introduce `quad_layout()` in full. While we've already introduced `ggheatmap()`—a specialized version of `quad_layout()`—most of the operations discussed in Chapter 3 can also be applied to `quad_layout()`. In the next section, we'll delve into `quad_layout()` and explore its full functionality.

6 quad-layout

`quad_layout()` arranges plots in the Quad-Side of a main plot.

QuadLayout

```
                          Plot 1
                          Plot 2
                          Plot 3
Plot 1  Plot 2  Plot 3    Main Plot          Plot 1  Plot 2  Plot 3




                          Plot 1
                          Plot 2
                          Plot 3
```

```r
library(ggalign)
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))
```

6.1 introduction

Depending on whether you want to align observations in the `horizontal` or `vertical` direction, there are four main types of `quad_layout()`:

Alignment of Observations	horizontal	vertical	Data Format
quad_free()/ggside()			data frame
quad_alignh()			matrix
quad_alignv()			matrix
quad_alignb()/ggheatmap()			matrix

When the layout requires alignment of observations, we typically use a matrix, regardless of whether alignment is needed in one or two directions.

- For data frame, the input will be automatically converted using `fortify_data_frame()` if needed.
- For matrix, the input will be automatically converted using `fortify_matrix()` if required.

6.2 Annotations

Annotation is typically handled using a `stack_layout()`. Depending on whether you want to align observations in the specified direction, different `stack_layout()` are compatible (Section 3.5). Below is a table outlining the compatibility of various layout types for annotations:

Annotations	left and right	top and bottom
quad_free()/ggside()	stack_freeh()	stack_freev()
quad_alignh()	stack_freeh()/stack_alignh()	stack_freev()
quad_alignv()	stack_freeh()	stack_freev()/stack_alignv()
quad_alignb()/ggheatmap()	stack_freeh()/stack_alignh()	stack_freev()/stack_alignv()

6.3 `quad_alignb()`

`quad_alignb()` aligns observations in both horizontal and vertical directions. It serves as the base version of `ggheatmap()`/`heatmap_layout()` and does not automatically add default layers or mappings.

The underlying `ggplot` data of the main plot is the same with `ggheatmap()`/`heatmap_layout()`, it is recommended to use `.y`, or `.row_names` as the y mapping and use `.x` or `.column_names` as the x mapping in the main plot.

```
quad_alignb(small_mat, aes(.x, .y)) +
    geom_tile(aes(fill = value)) +
    scale_fill_viridis_c()
```
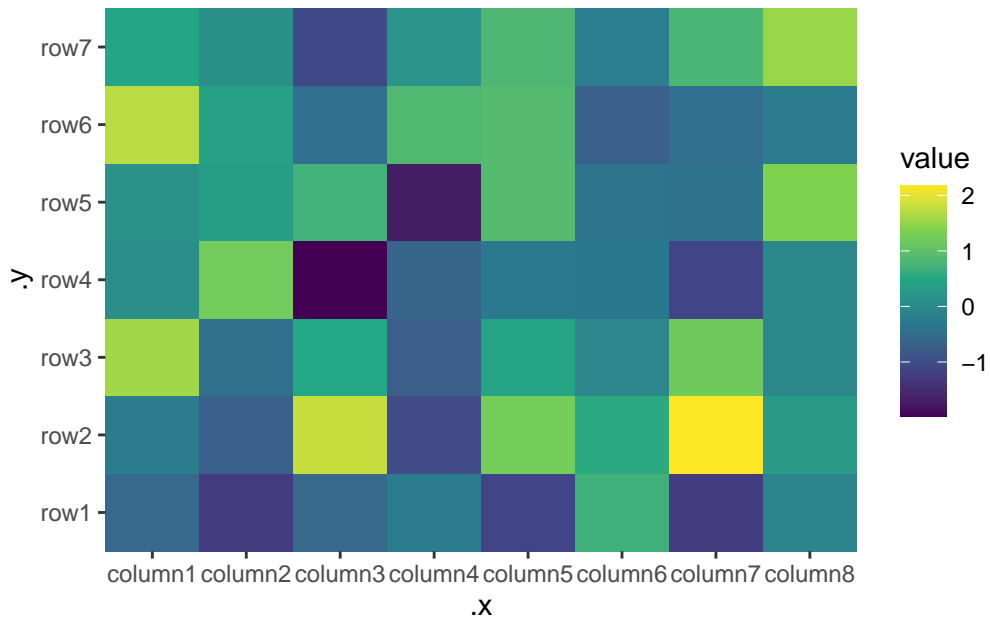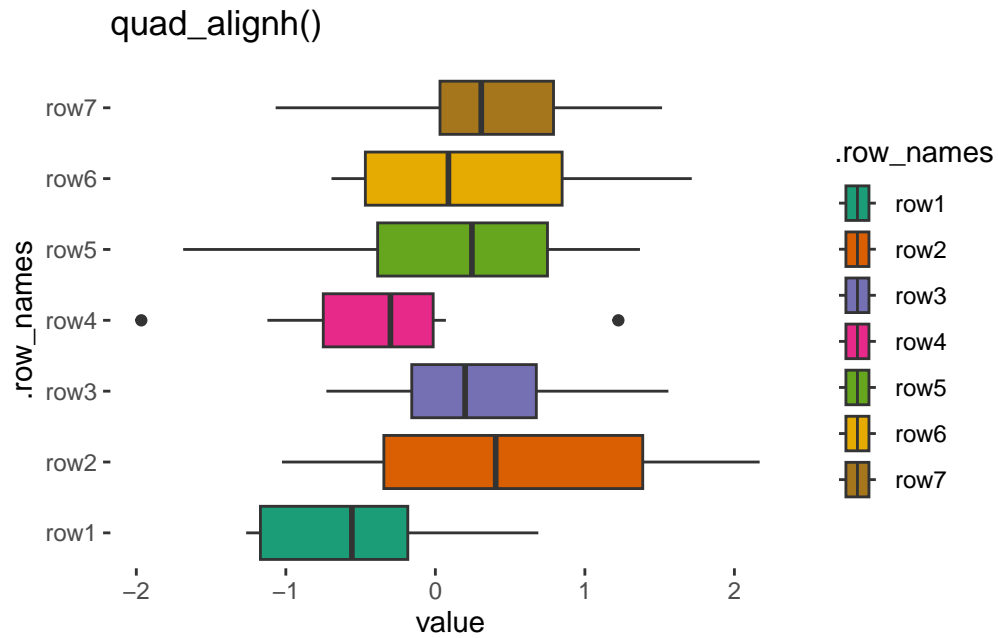
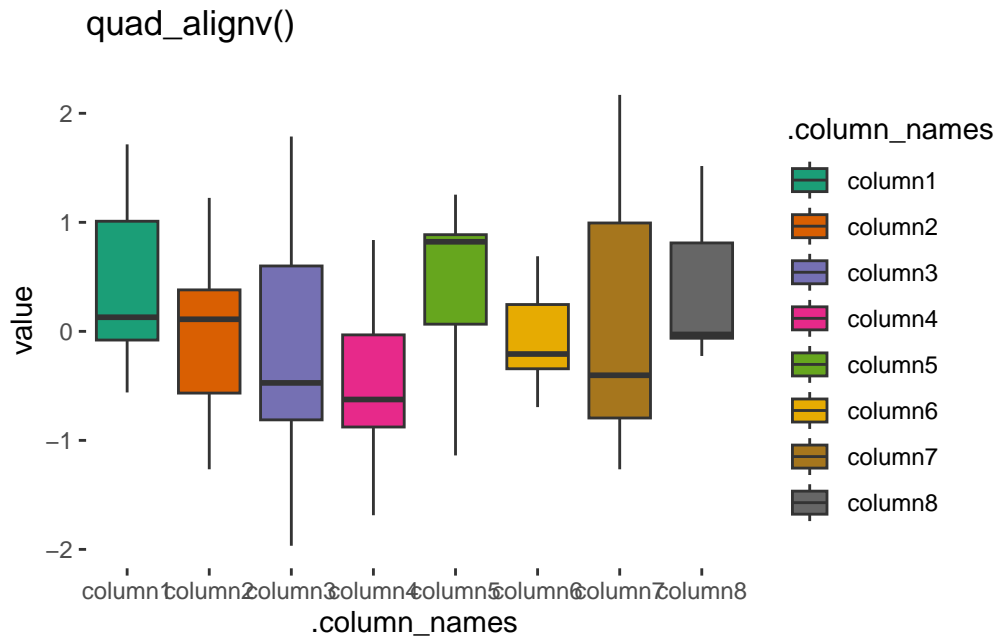## 6.4 `quad_alignh()`/`quad_alignv()`

`quad_alignh()` aligns observations horizontally, while `quad_alignv()` aligns observations vertically.

The underlying `ggplot` data is the same with `ggheatmap()`/`heatmap_layout()`, it is recommended to use `.y`, or `.row_names` as the y mapping in the main plot for `quad_alignh()` but use `.x` or `.column_names` as the x mapping in the main plot for `quad_alignv()`.

```
quad_alignh(small_mat) +
    geom_boxplot(aes(value, .row_names, fill = .row_names)) +
    scale_fill_brewer(palette = "Dark2") +
    layout_title("quad_alignh()")
```

```
quad_alignv(small_mat) +
    geom_boxplot(aes(.column_names, value, fill = .column_names)) +
    scale_fill_brewer(palette = "Dark2") +
    layout_title("quad_alignv()")
```

quad_alignv()

value

2 -
1 -
0 -
-1 -
-2 -

column1 column2 column3 column4 column5 column6 column7 column8

.column_names

.column_names

- column1
- column2
- column3
- column4
- column5
- column6
- column7
- column8

As discussed in Section 3.4, `quad_anno()` will always attempt to initialize a `stack_layout()` with the same alignment as the current direction. For top and bottom annotations in `quad_alignh()`, and left and right annotations in `quad_alignv()`, `quad_anno()` will not initialize the annotation due to inconsistent data types.

```
quadh <- quad_alignh(small_mat) +
    anno_top()
#> Warning: `data` in `quad_alignh()` is a double matrix, but the top annotation stack need
#> a <data.frame>, won't initialize the top annotation stack
quadv <- quad_alignv(small_mat) +
    anno_left()
#> Warning: `data` in `quad_alignv()` is a double matrix, but the left annotation stack
#> need a <data.frame>, won't initialize the left annotation stack
```

Manual adding of a `stack_layout()` is required in such cases, you can set `initialize = FALSE` to prevent the warning message.

```
quadh <- quad_alignh(small_mat) +
    anno_top(initialize = FALSE)
quadv <- quad_alignv(small_mat) +
    anno_left(initialize = FALSE)
```

```
quadh +
    stack_freev(mpg) +
    # add a plot in the top annotation
    ggfree(mapping = aes(displ, hwy, colour = class)) +
    geom_point(aes(displ, hwy, colour = class)) +
    quad_active() +
    geom_boxplot(aes(value, .row_names, fill = .row_names)) +
    scale_fill_brewer(palette = "Dark2")+
    layout_title("quad_alignh()")
```
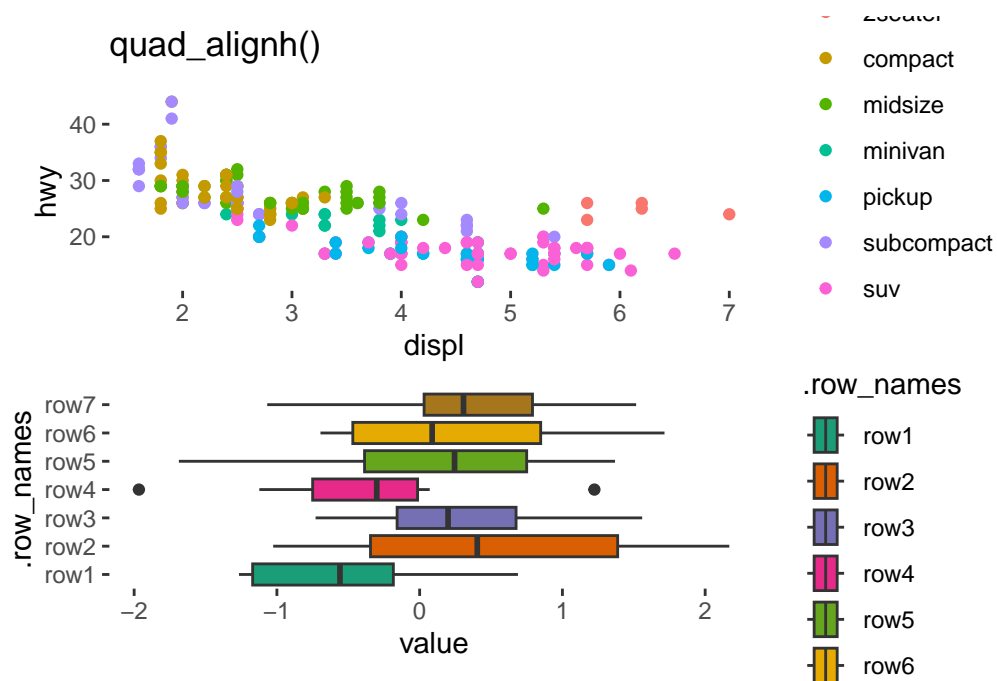


```
quadv +
    stack_freeh(data = mpg) +
    # add a plot in the top annotation
    ggfree(mapping = aes(displ, hwy, colour = class)) +
    geom_point(aes(displ, hwy, colour = class)) +
    quad_active() +
    geom_boxplot(aes(.column_names, value, fill = .column_names)) +
    scale_fill_brewer(palette = "Dark2") +
    layout_title("quad_alignv()")
```
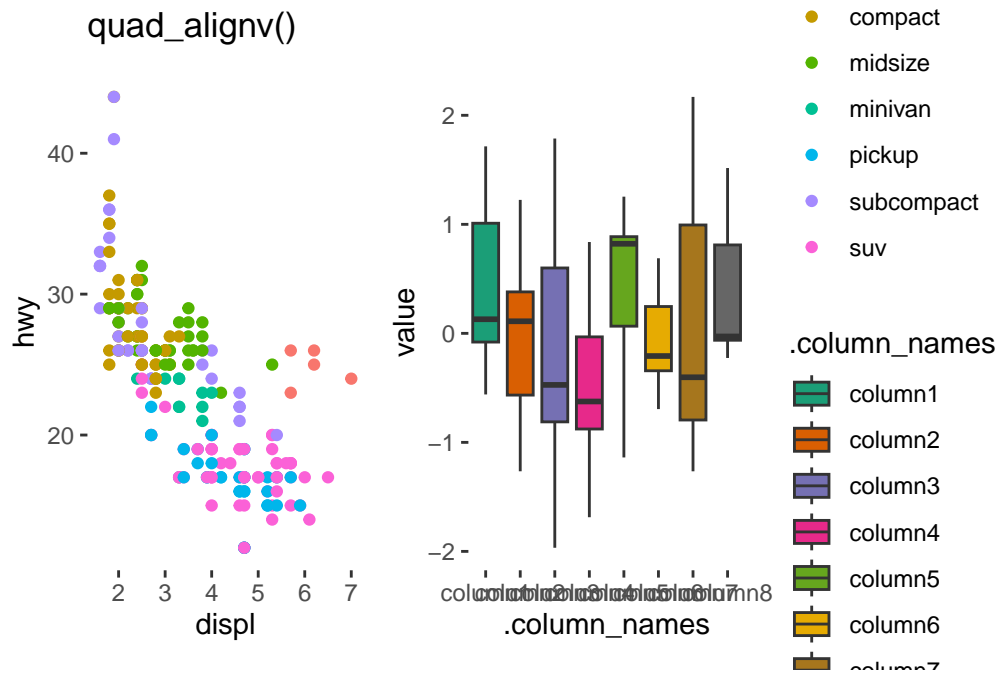
Alternatively, you can set `initialize = TRUE`, which will initialize the annotation stack layout with no data. In this case, you must provide `data` in each plot within the annotation.

```
quad_alignh(small_mat) +
    geom_boxplot(aes(value, .row_names, fill = .row_names)) +
    scale_fill_brewer(palette = "Dark2") +
    anno_top(initialize = TRUE) +
    ggfree(data = mpg, aes(displ, hwy, colour = class)) +
    geom_point(aes(displ, hwy, colour = class))+
    layout_title("quad_alignh()")
```
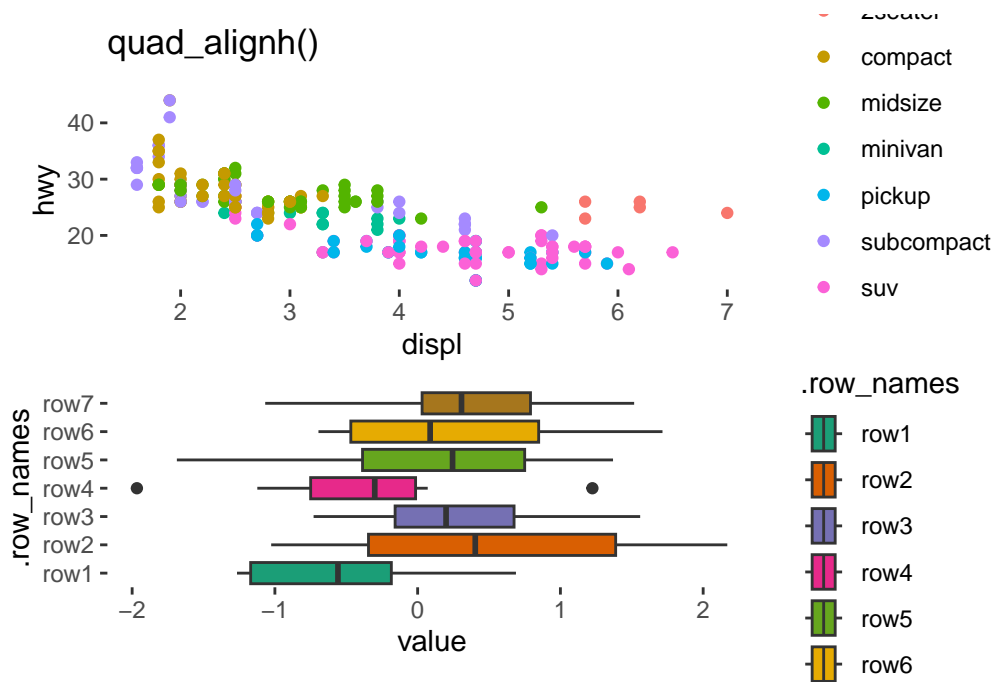
```
quad_alignv(small_mat) +
    geom_boxplot(aes(.column_names, value, fill = .column_names)) +
    scale_fill_brewer(palette = "Dark2") +
    anno_left(initialize = TRUE) +
    ggfree(data = mpg, aes(displ, hwy, colour = class)) +
    geom_point(aes(displ, hwy, colour = class))+
    layout_title("quad_alignv()")
```
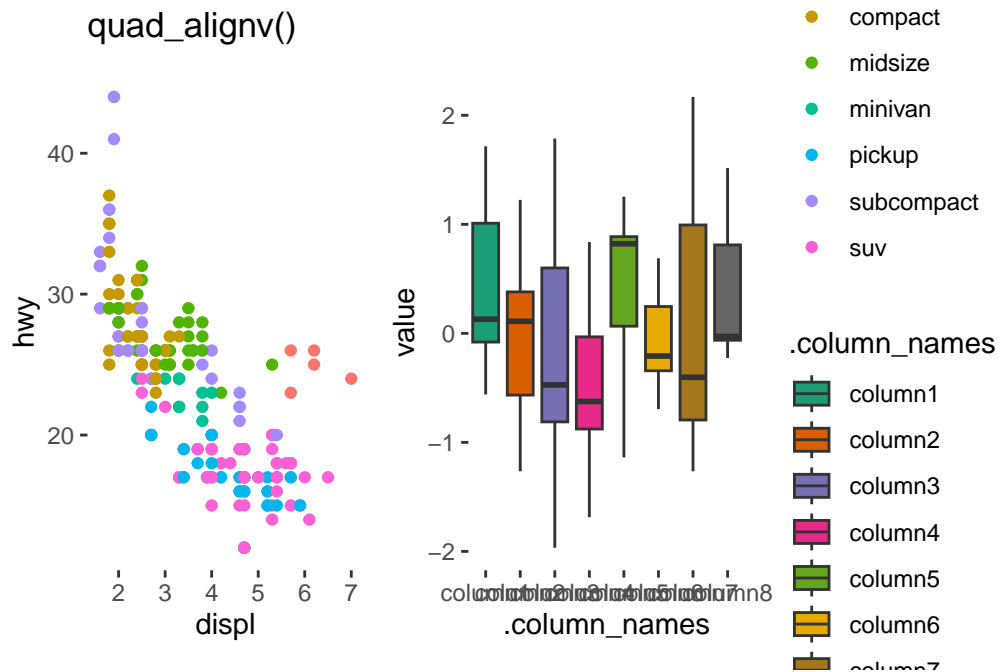
6.5 `quad_free()`

`quad_free()` does not align observations and is functionally equivalent to the ggside package. For convenience, `ggside()` is provided as an alias for quad_free(). This layout is particularly useful for adding metadata or summary graphics along a continuous axis.

```r
ggside(mpg, aes(displ, hwy, colour = class)) +
    geom_point(size = 2) +
    # initialize top annotation
    anno_top(size = 0.3) +
    # add a plot in the top annotation
    ggfree() +
    geom_density(aes(displ, y = after_stat(density), colour = class), position = "stack") +
    # initialize right annotation
    anno_right(size = 0.3) +
    # add a plot in the right annotation
    ggfree() +
    geom_density(aes(x = after_stat(density), hwy, colour = class),
        position = "stack"
    ) &
    theme_bw()
```
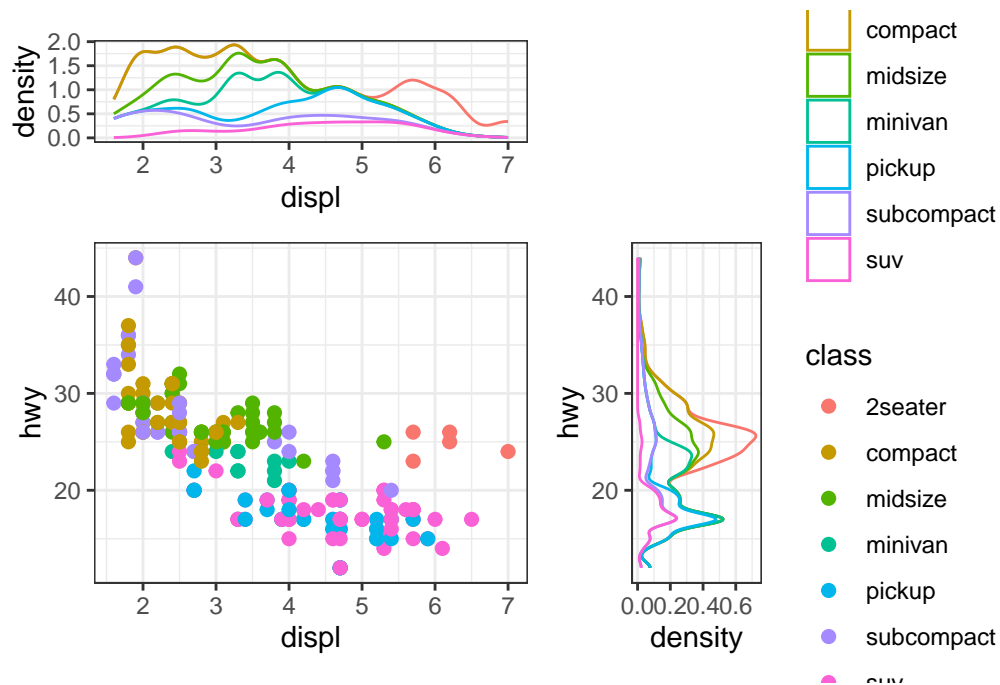
`ggside()` allows facetting for the main plot, which should also be applied to the annotations for proper alignment.

```
i2 <- iris
i2$Species2 <- rep(c("A", "B"), 75)
ggside(i2, aes(Sepal.Width, Sepal.Length, color = Species)) +
    geom_point(size = 2) +
    facet_grid(Species ~ Species2) +
    anno_top(size = 0.3) +
    ggfree() +
    geom_density(aes(Sepal.Width, y = after_stat(density), colour = Species),
        position = "stack"
    ) +
    facet_grid(cols = vars(Species2)) +
    anno_right(size = 0.3) +
    ggfree() +
    geom_density(aes(x = after_stat(density), Sepal.Length, colour = Species),
        position = "stack"
    ) +
    facet_grid(rows = vars(Species)) &
    theme_bw()
```
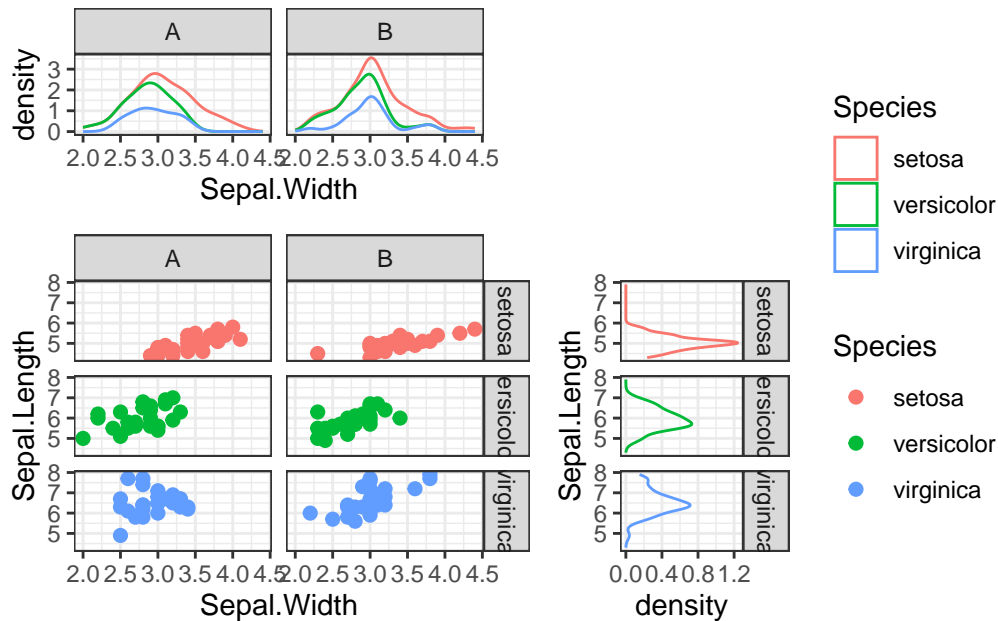
If an annotation contains multiple plots, it can be tedious to add the same element to each one individually. One way to simplify this is by creating an external `stack_layout()` and adding the desired elements using the `&` operator. Then, you can add this `stack_layout()` to the `quad_layout()`. In a later chapter, I will introduce another operator that seamlessly combines with the `+` operator, allowing you to add elements to multiple plots at once.

In the next chapter, we will explore even more advanced techniques for combining multiple `quad_layout()`s. These methods will provide you with the tools to manage more complex plot arrangements and make your visualizations even more flexible and powerful.

7 A list of `quad_layout()`

Similarly, `stack_layout()` can be added to a `quad_layout()`, and a `quad_layout()` can also be added to a `stack_layout()`.

```
library(ggalign)
#> Loading required package: ggplot2
set.seed(123)
small_mat <- matrix(rnorm(56), nrow = 7)
rownames(small_mat) <- paste0("row", seq_len(nrow(small_mat)))
colnames(small_mat) <- paste0("column", seq_len(ncol(small_mat)))
```

> note: when `stack_layout()` contains a nested `quad_layout()`, it cannot be used within the annotation of another `quad_layout()`.
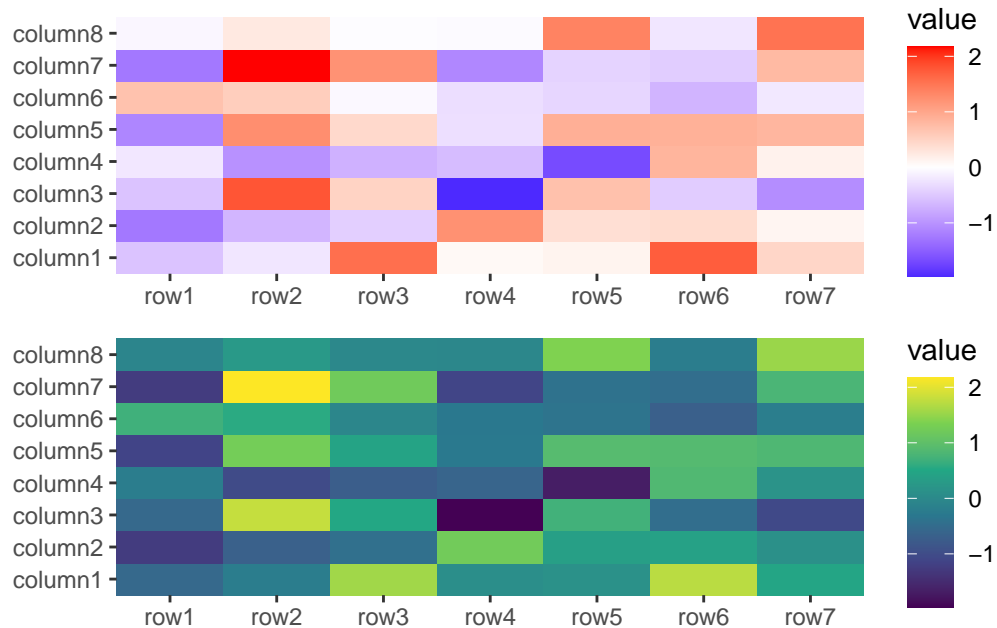
7.1 `stack_align()`

As discussed in Section 2.3, `stack_align()` can accept plots regardless of whether alignment of observations is required.

```
stack_alignh(small_mat) +
    ggheatmap() +
    ggheatmap()
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```
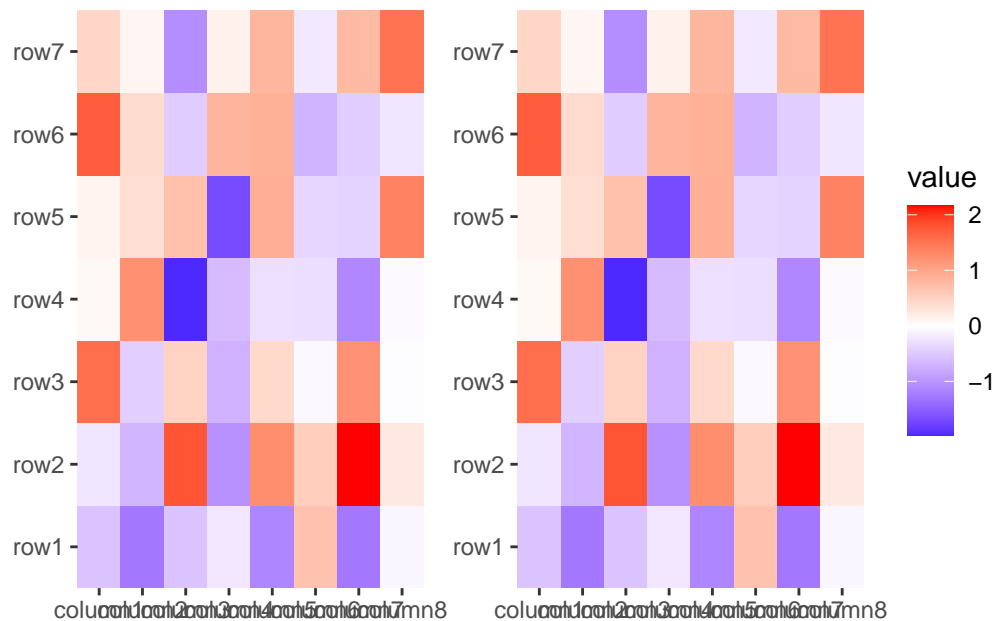
When `ggheatmap()`/`quad_layout()` is added to a `stack_layout()`, it will also set the active context to itself, which means subsequent addition will be directed to `ggheatmap()`/`quad_layout()`. One exception is the `ggheatmap()`/`quad_layout()` itself, which cannot be added to another `quad_layout()`. In this case, they will be added directly to the `stack_layout()`.

```
stack_alignv(small_mat) +
    ggheatmap() +
    ggheatmap() +
    scale_fill_viridis_c()
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```

The data of `ggheatmap()`/`quad_layout()` can inherit from the parent `stack_layout()`, but the data format must match. Otherwise, you will need to manually provide the data. For `stack_align()`, the data is always expected to be a matrix, if you try to add a `ggside()` (which uses a data frame), it will result in an error.

```
stack_alignh(small_mat) +
    ggside()
#> Error in `stack_layout_add()`:
#> ! Cannot add `quad_free()` to `stack_align()`
#> i `data` in `stack_align()` is a double matrix, but `quad_free()` need a
#>    <data.frame>.
#> i Try provide `data` in `quad_free()`
```

When `ggheatmap()`/`quad_layout()` is added to a vertical `stack_align()`, the inherited matrix is transposed before use. This is because the columns of `ggheatmap()`/`quad_layout()` must match the number of observations in `stack_align()`.

```
stack_alignv(small_mat) +
    ggheatmap() +
    ggheatmap()
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```
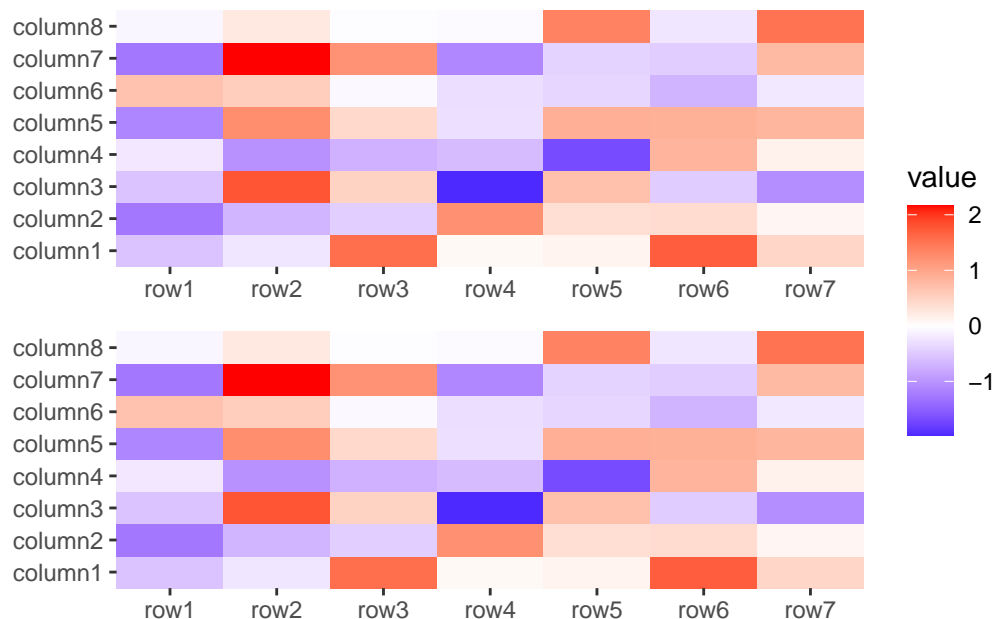
`stack_align()` ensures that all plots aligned along the stack have the same ordering index or groups for the observations. Here's a table summarizing which `quad_layout()` will be aligned in `stack_align()`.

|	stack_alignh()	stack_alignv()
quad_free()/ggside()		
quad_alignh()		
quad_alignv()		
quad_alignb()/ggheatmap()		

We can customize the layout in the `stack_align()` directly, or in `quad_layout()`. As introduced in Section 2.5, you can easily switch from the `ggheatmap()`/`quad_layout()` to the `stack_layout()` using `stack_active()`.

```
stack_alignv(small_mat) +
    ggheatmap() +
    ggheatmap() +
    anno_bottom(size = 0.2) +
    align_dendro(aes(color = branch), k = 3)+
    layout_title("dendrogram in ggheatmap()")
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```

dendrogram in ggheatmap()
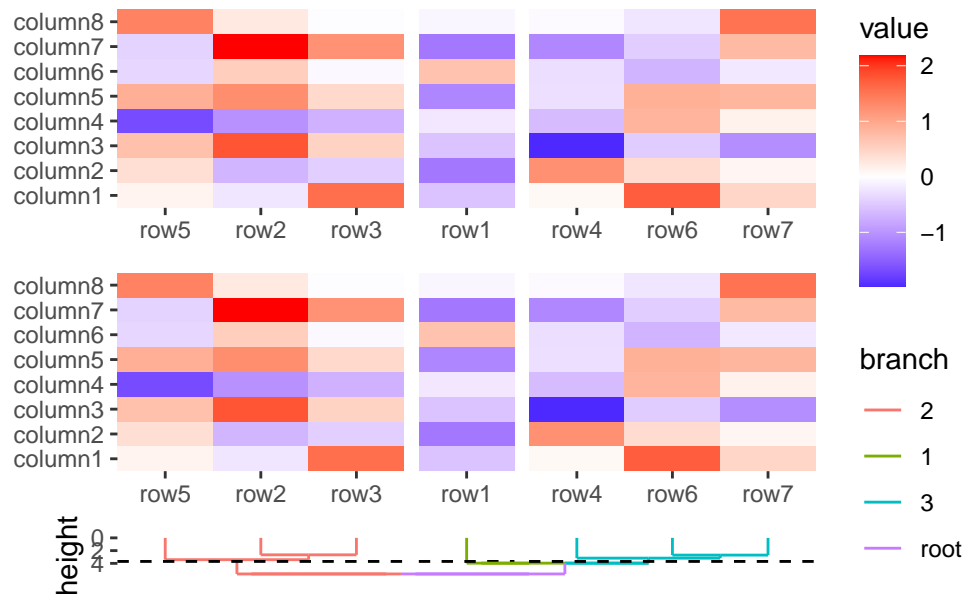
```
stack_alignv(small_mat) +
    ggheatmap() +
    ggheatmap() +
    stack_active() +
    align_dendro(aes(color = branch), k = 3, size = 0.2) +
    scale_y_reverse() +
    layout_title("dendrogram in stack_layout()")
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```

dendrogram in stack_layout()
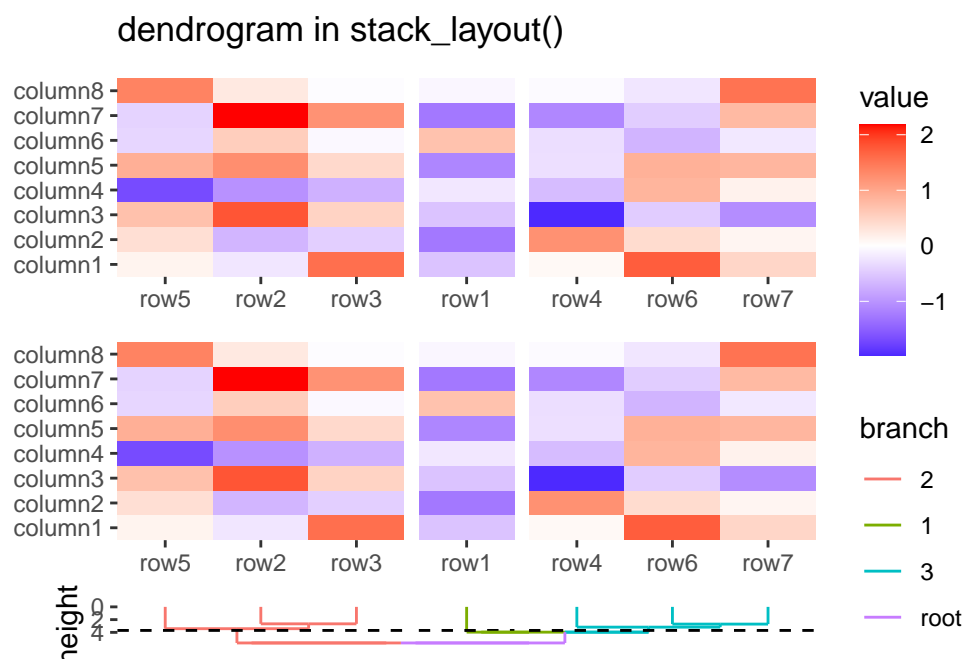
7.2 `stack_free()`

However, `stack_free()` is designed to only accept plots that are not aligned along the axis (Section 2.3). Below is a table summarizing which `quad_layout()` are compatible with `stack_free()`:

	stack_freeh()	stack_freev()
quad_free()/ggside()		
quad_alignh()		
quad_alignv()		
quad_alignb()/ggheatmap()		

```
alignh_quad_plot <- quad_alignh(small_mat) +
    geom_boxplot(aes(value, .row_names, fill = .row_names)) +
    scale_fill_brewer(palette = "Dark2")
stack_freev() +
    alignh_quad_plot +
    alignh_quad_plot
```
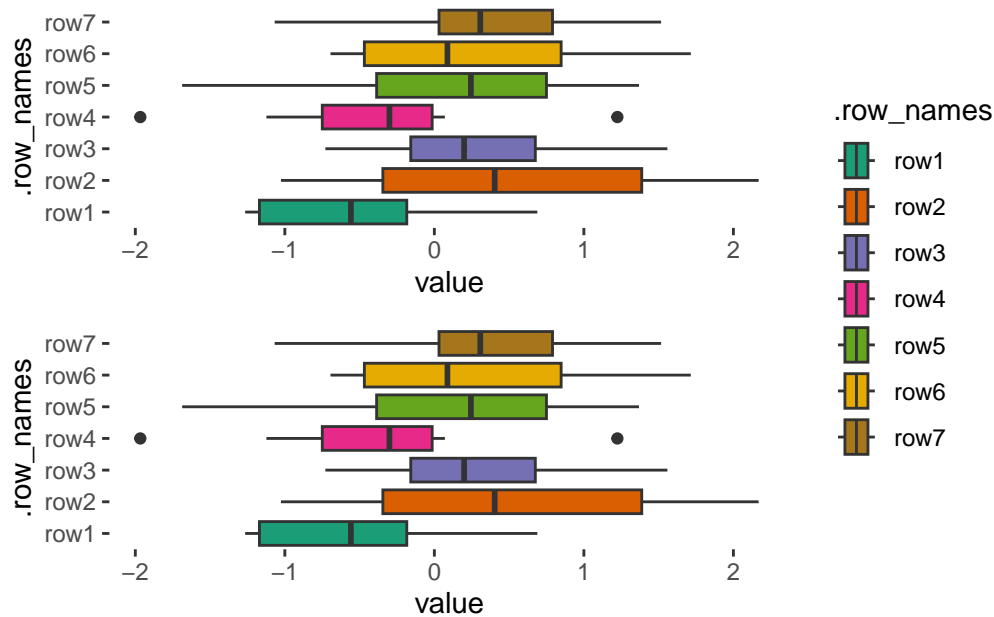
7.3 Control sizes

A numeric or a unit object of length 3 should be provided in `stack_align()`/`stack_free()` when placing a `quad_layout()`. For vertical `stack_layout()`, this means `quad_layout()` with left or right annotations; for horizontal `stack_layout()`, this means `quad_layout()` with top or bottom annotations. The first size controls the relative `width`/`height` of the left or top annotation, the second controls the relative `width`/`height` of the main plot, and the third controls the relative `width`/`height` of the right or bottom annotation.

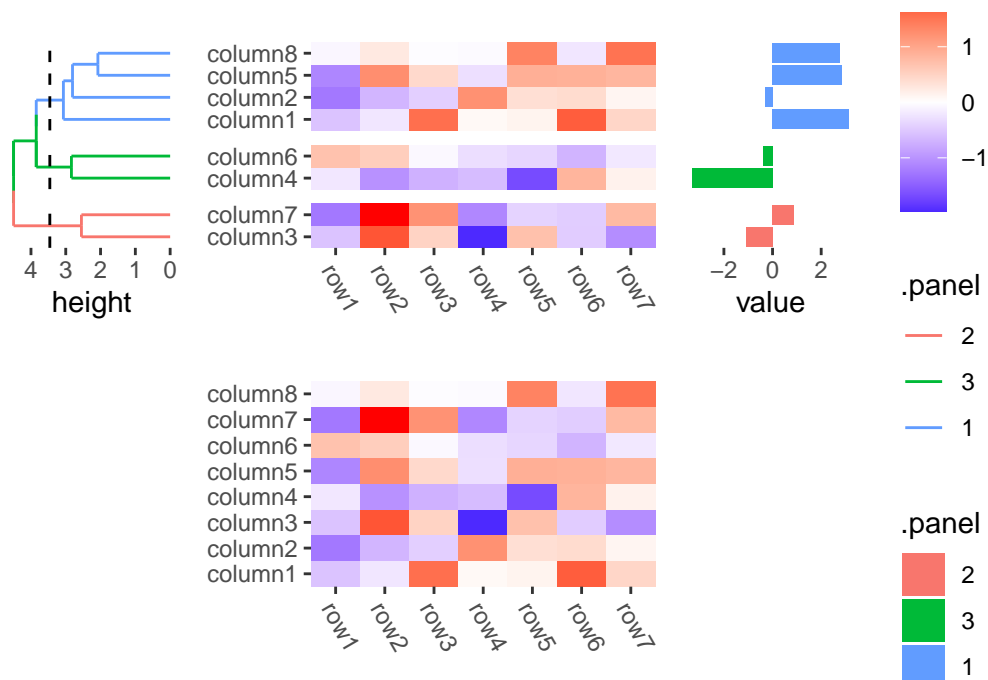By default the three rows/columns will have equal sizes.

```
stack_alignv(small_mat) +
    ggheatmap() +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_left() +
    align_dendro(aes(color = .panel), k = 3L) +
    anno_right() +
    ggalign(data = rowSums) +
    geom_bar(aes(value, fill = .panel), orientation = "y", stat = "identity") +
    ggheatmap() +
    theme(axis.text.x = element_text(angle = -60, hjust = 0))
```

```
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```
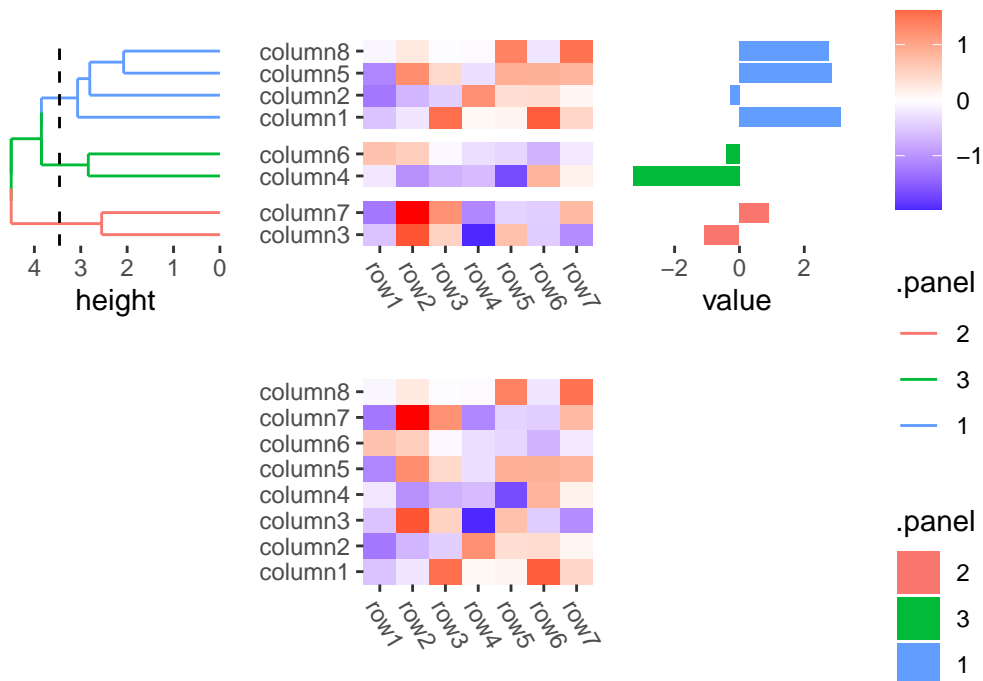


```
heat1 <- ggheatmap(t(small_mat)) +
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_left() +
    align_dendro(aes(color = .panel), k = 3L) +
    anno_right() +
    ggalign(data = rowSums) +
    geom_bar(aes(value, fill = .panel), orientation = "y", stat = "identity")

stack_alignv(small_mat, sizes = c(1, 2, 1)) +
    heat1 +
    ggheatmap() +
    theme(axis.text.x = element_text(angle = -60, hjust = 0))
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```

In this way, the width/height of main plot specified in `quad_active()` or `quad_layout()`/`ggheatmap()` won't work.

```
stack_alignv(small_mat) +
    ggheatmap(width = unit(2, "null")) + # not work
    theme(axis.text.x = element_text(angle = -60, hjust = 0)) +
    anno_left() +
    align_dendro(aes(color = .panel), k = 3L) +
    anno_right() +
    ggalign(data = rowSums) +
    geom_bar(aes(value, fill = .panel), orientation = "y", stat = "identity") +
    ggheatmap(width = unit(2, "null")) + # not work
    theme(axis.text.x = element_text(angle = -60, hjust = 0))
#> > heatmap built with `geom_tile()`
#> > heatmap built with `geom_tile()`
```
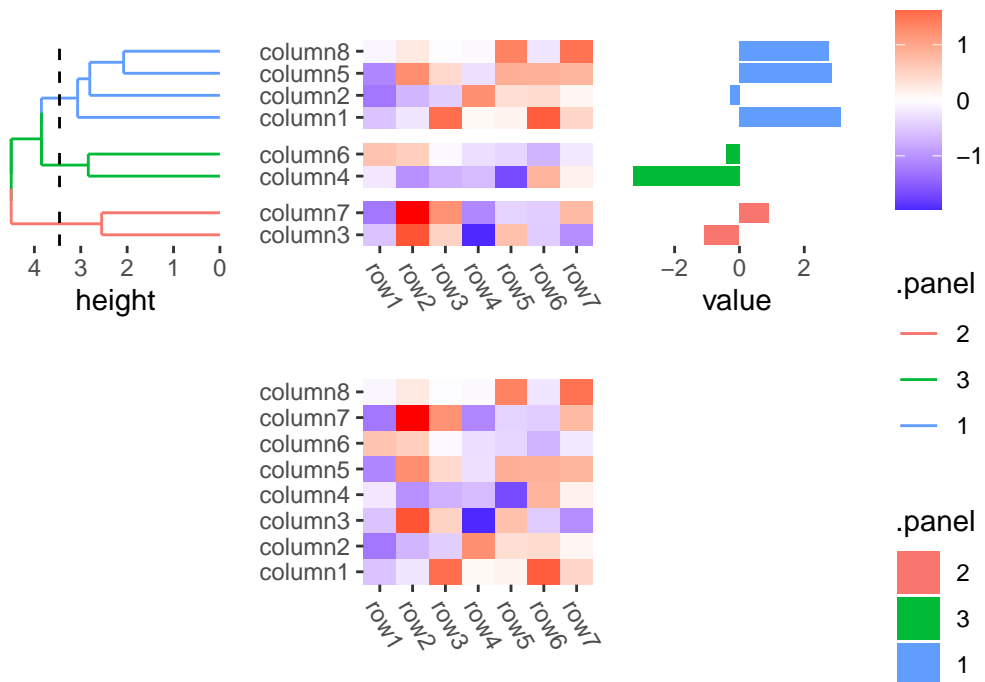
Next, we will build on these concepts and explore even more advanced strategies for integrating elements across multiple plots or annotations in a layout.