3. Themes

Fertige Themes, eigene Theme-Elemente, Google Fonts und formatierter Text

Dr. Paul Schmidt

```
for (pkg in c("ggplot2", "showtext", "ggtext")) {
  if (!require(pkg, character.only = TRUE)) install.packages(pkg)
}
```

Alles was wir bisher getan haben - Achsen, Farben, Formen - betrifft die Darstellung der *Daten*. Die `theme()`-Funktion steuert dagegen alle *nicht-datenbezogenen* visuellen Elemente: Hintergruende, Gitterlinien, Achsenlinien, Textformatierung, Legenden, Abstaende und mehr. Diese Elemente tragen zwar keine Daten, haben aber einen enormen Einfluss darauf, wie professionell und lesbar ein Plot wirkt.

Themes zu beherrschen ist das, was einen schnellen explorativen Plot von einer publikationsreifen Grafik trennt. Die gute Nachricht: ggplot2 macht das bemerkenswert systematisch - es gibt nur vier Typen von Theme-Elementen zu lernen und eine Handvoll Complete Themes, die sinnvolle Ausgangspunkte bieten.

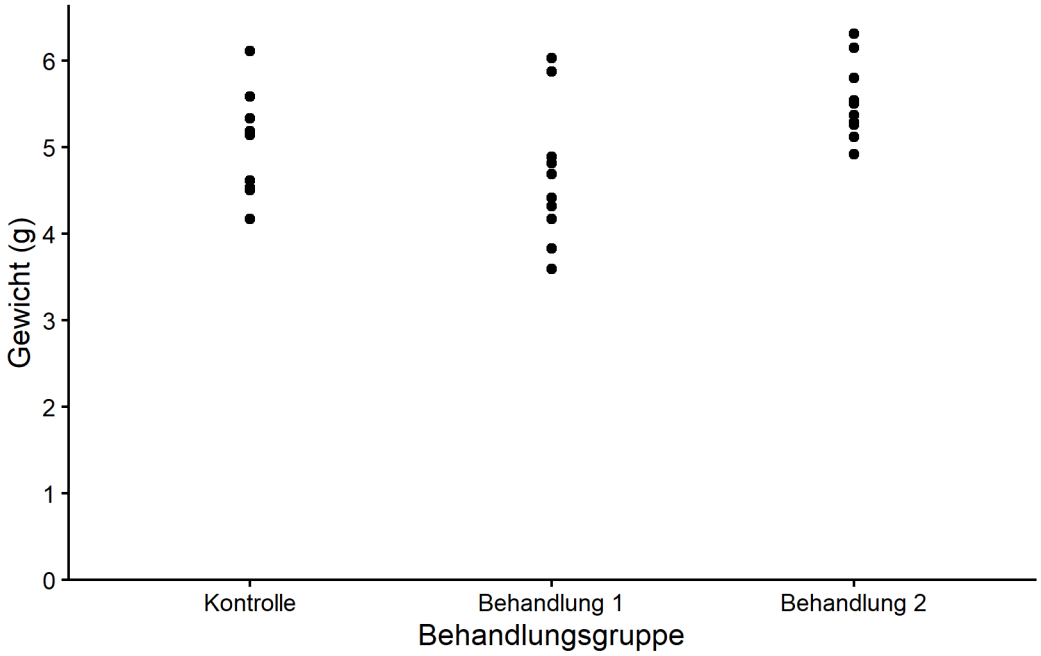
Wir arbeiten weiter mit dem `PlantGrowth`-Plot aus den vorherigen Kapiteln:

```
myplot <- ggplot(data = PlantGrowth) +
  aes(y = weight, x = group) +
  geom_point() +
  scale_y_continuous(
    name = "Gewicht (g)",
    limits = c(0, NA),
    breaks = seq(0, 6),
    expand = expansion(mult = c(0, 0.05))
  ) +
  scale_x_discrete(
    name = "Behandlungsgruppe",
    labels = c(
      ctrl = "Kontrolle",
      trt1 = "Behandlung 1",
      trt2 = "Behandlung 2"
    )
  )
```

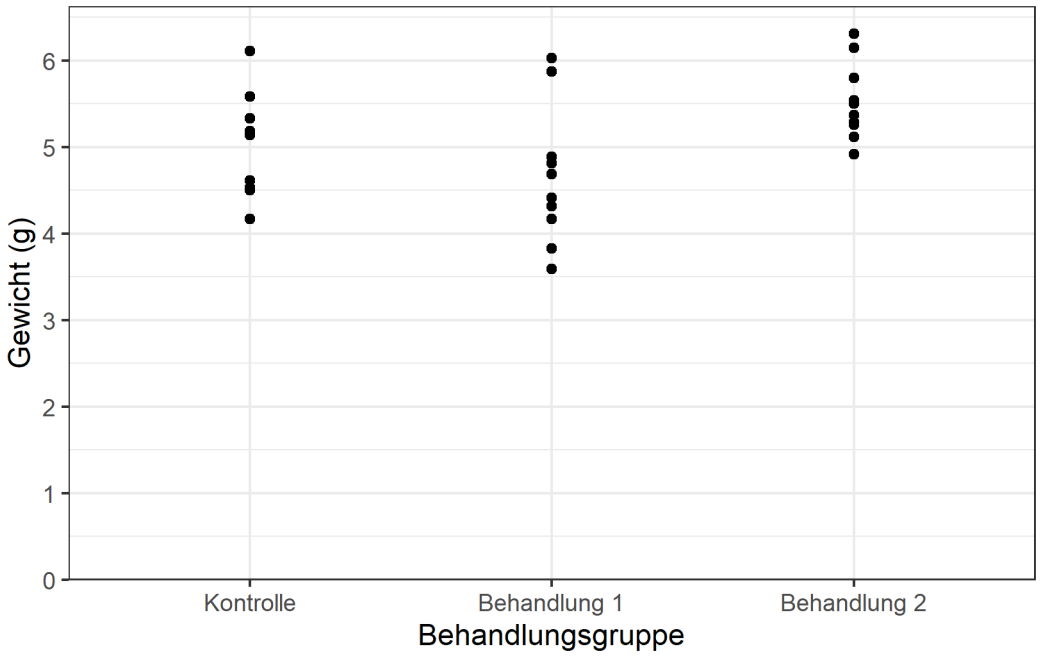
Fertige Themes

Anstatt jedes visuelle Detail von Grund auf zu konfigurieren, bringt ggplot2 mehrere vordefinierte “Complete Themes” mit, die das gesamte Erscheinungsbild in einem Schritt aendern. Der Standard ist `theme_grey()` - erkennbar an seinem grauen Hintergrund und weissen Gitterlinien. Funktional, aber selten das, was man in einem Bericht oder einer Publikation moechte. Hier sind einige beliebte Alternativen:

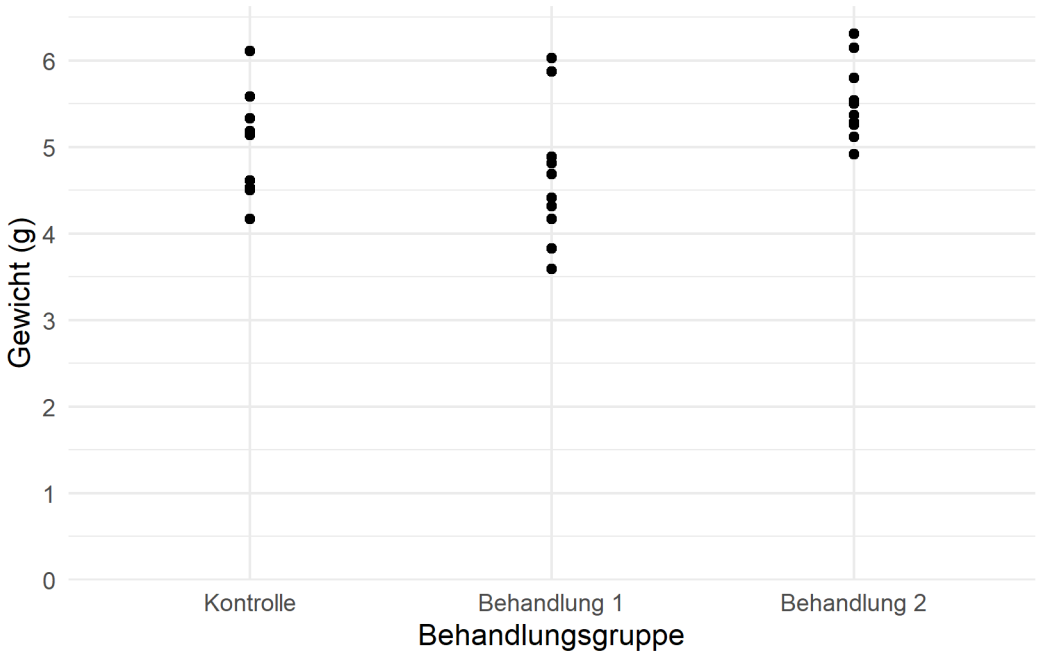
```
myplot +
  theme_classic()
```



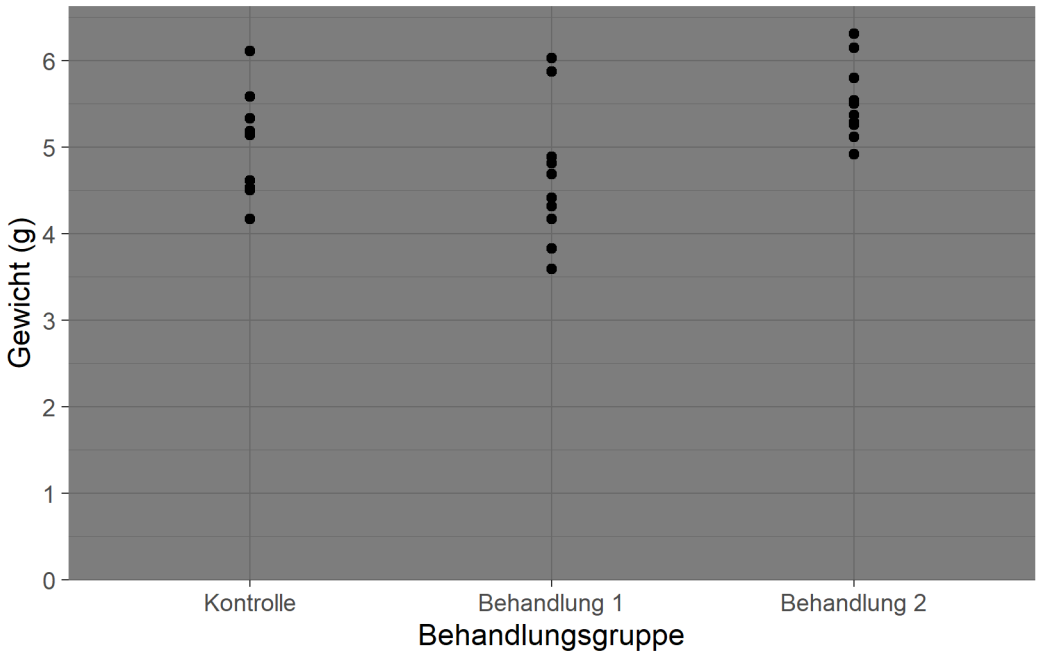

```
myplot +
  theme_bw()
```




```
myplot +
  theme_minimal()
```

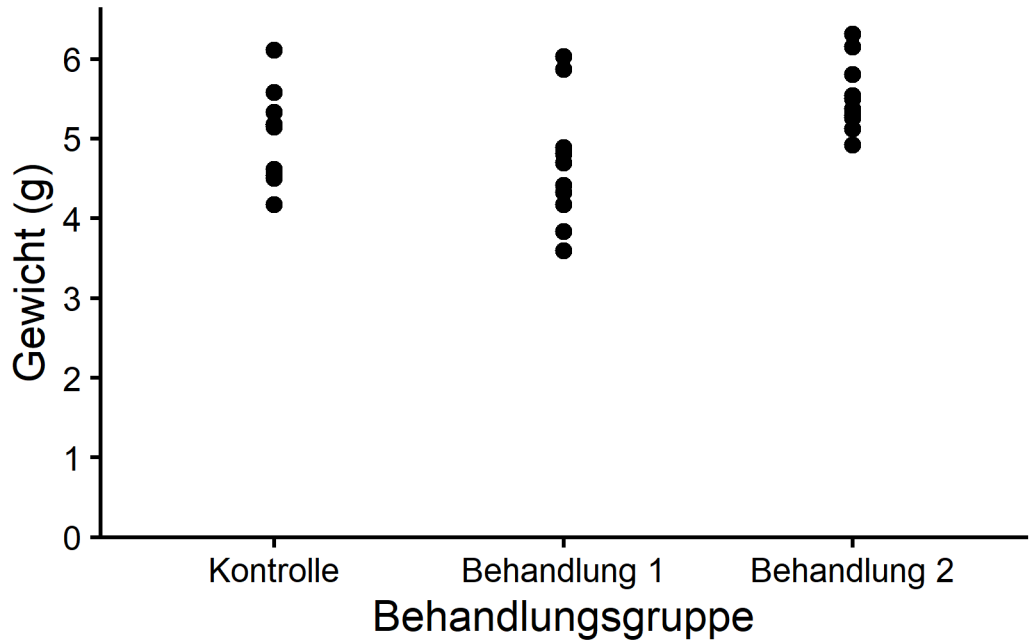



```
myplot +
  theme_dark()
```



Alle Complete Themes akzeptieren ein `base_size`-Argument, das alle Textelemente proportional skaliert. Das ist besonders nuetzlich, wenn man Plots fuer Praesentationen (groesser) vs. Publikationen (kleiner) vorbereitet:

```
myplot +
  theme_classic(base_size = 16)
```



Theme-Elemente anpassen

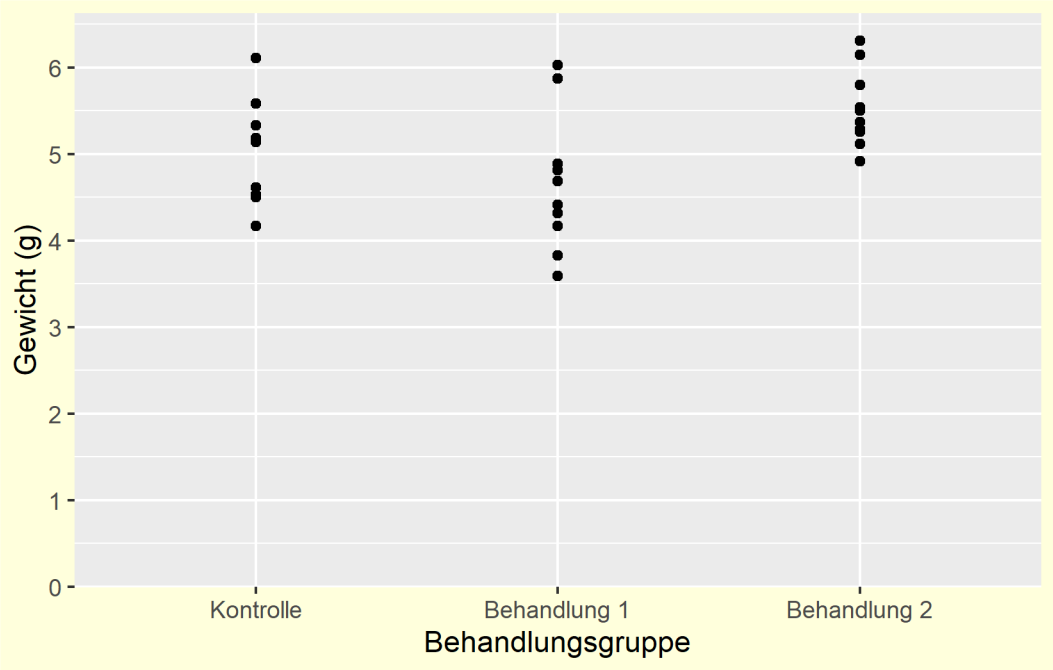
Ein Complete Theme bietet einen guten Ausgangspunkt, aber fast immer moechte man einzelne Details anpassen - vielleicht die Minor-Gitterlinien entfernen, die Achsenlinienfarbe aendern oder die Textgroesse anpassen. Dafuer ist `theme()` da. Entscheidend ist, zuerst ein Complete Theme anzuwenden und dann mit `theme()` spezifische Elemente zu ueberschreiben. Die Reihenfolge ist wichtig: `theme()`-Aufrufe *nach* einem Complete Theme ueberschreiben dessen Standards.

Es gibt genau vier Elementtypen, und jede Theme-Anpassung verwendet einen davon:

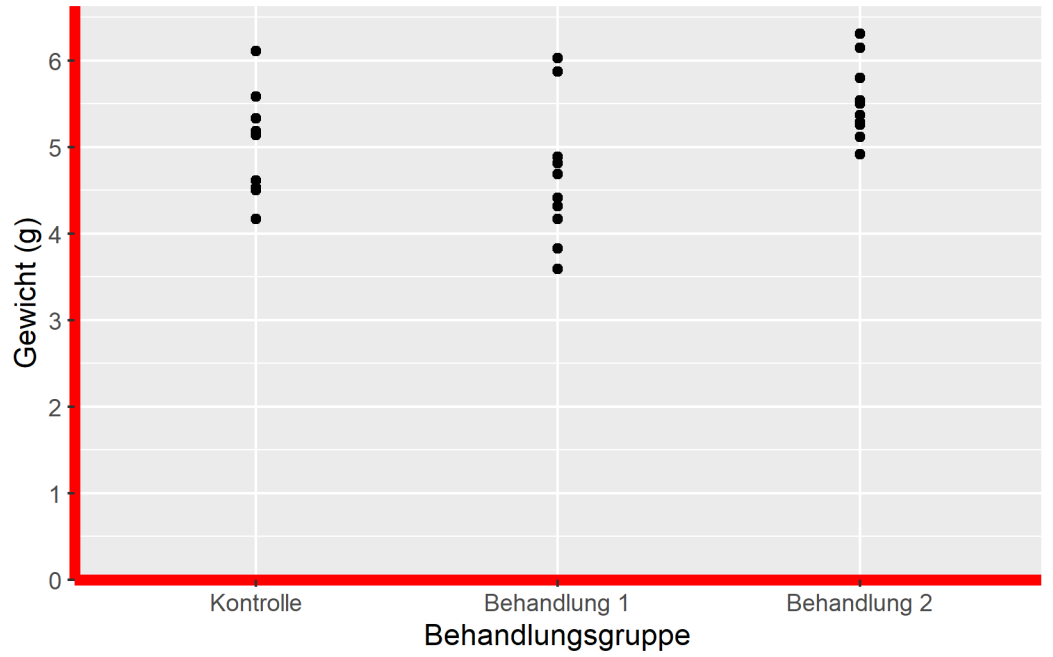
- `element_blank()` - entfernt das Element vollstaendig
- `element_rect()` - fuer Raender und Hintergruende
- `element_line()` - fuer Linien (Achsen, Gitter)
- `element_text()` - fuer alle Textelemente

Wenn man diese vier Bausteine kennt, ist jede Theme-Anpassung eine Frage des richtigen Elementnamens (wie `plot.background`, `axis.line`, `panel.grid.major.y`) und des passenden Elementtyps. Hier zwei bewusst uebertriebene Beispiele zur Veranschaulichung:

```
myplot +
  theme(plot.background = element_rect(fill = "lightyellow"))
```

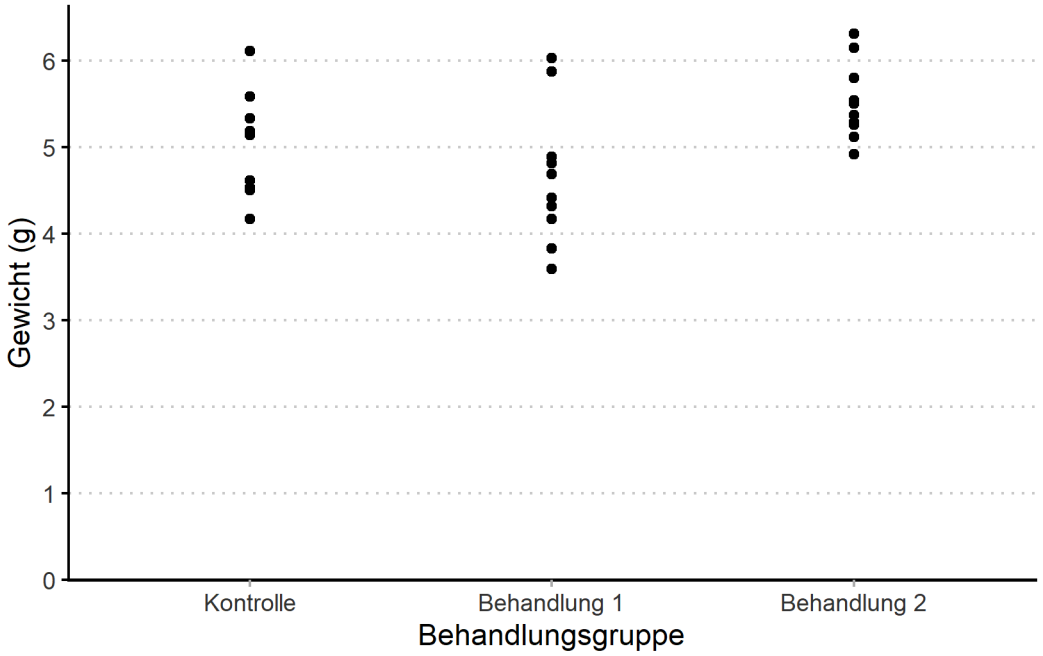



```
myplot +
  theme(axis.line = element_line(color = "red", linewidth = 2))
```



Ein praktischeres Beispiel: Ausgehend von `theme_classic()` wird die x-Achsenlinie beibehalten und dezente horizontale Gitterlinien hinzugefuegt:

```
myplot +
  theme_classic() +
  theme(
    panel.grid.major.y = element_line(linetype = "dotted", color = "#CCCCCC"),
    axis.line.x = element_line(color = "black", linewidth = 0.6),
    axis.ticks.x = element_line(color = "#AAAAAA"),
    axis.text = element_text(color = "#333333")
  )
```



Google Fonts mit showtext

Die Standardschriften in R-Plots sind auf wenige Systemschriften beschraenkt. Fuer publikationsreife Grafiken benoetigt man oft eine bestimmte Schriftart - sei es passend zum Corporate Design, zum Journal-Stil oder einfach fuer einen saubereren Look. Das {showtext}-Paket loest dieses Problem, indem es jede Google Font in ggplot2 verfuegbar macht.

Der Workflow hat drei Schritte:

1. `sysfonts::font_add_google()` laedt die Schriftart herunter
2. `showtext::showtext_auto()` aktiviert das Rendering benutzerdefinierter Schriften
3. `showtext::showtext_opts(dpi = 300)` stellt die korrekte Aufloesung sicher

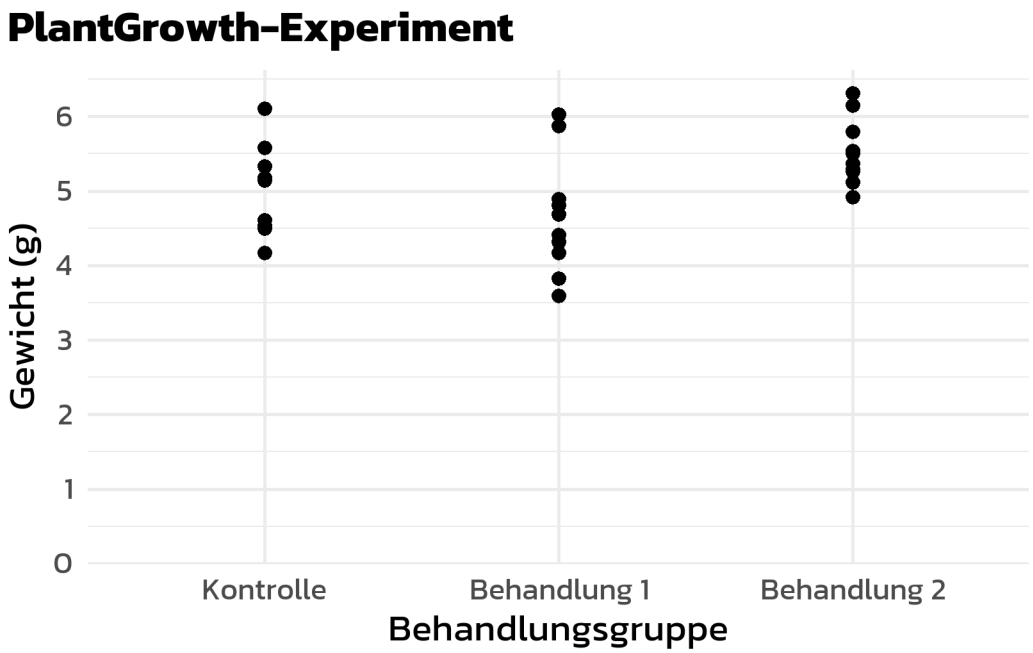
```
sysfonts::font_add_google("Kanit", "kanit")
showtext::showtext_auto()
showtext::showtext_opts(dpi = 300)
```

> **i Hinweis**
>
> Der `showtext_opts(dpi = 300)`-Aufruf ist wichtig: showtext nutzt standardmaessig 72 dpi, waehrend `ggsave()` standardmaessig 300 dpi verwendet. Ohne diese Abstimmung erscheinen Schriften in exportierten Plots in der falschen Groesse.

> ⚠ Warnung
>
> `font_add_google()` benoetigt eine Internetverbindung, um die Schriftart herunterzuladen. Wenn der Download fehlschlaegt (langsames Netzwerk, Firewall, Offline-Arbeit), steht die Schriftart nicht zur Verfuegung und Plots erzeugen einen Fehler oder fallen auf eine Standardschrift zurueck. Fuer Offline-Arbeit kann man die `.ttf`-Datei manuell von Google Fonts herunterladen und stattdessen `sysfonts::font_add()` mit einem lokalen Dateipfad verwenden.

Nach der Einrichtung kann die Schriftart in `theme()` ueber das `family`-Argument in `element_text()` verwendet werden:

```
myplot +
  labs(title = "PlantGrowth-Experiment") +
  theme_minimal(base_size = 14) +
  theme(
    text = element_text(family = "kanit"),
    plot.title = element_text(face = "bold", size = 16),
    plot.title.position = "plot"
  )
```



> ℹ Hinweis
>
> Beim Rendern in Quarto benoetigt man `fig-showtext: TRUE` in den knitr-Optionen (oder als Chunk-Option `#| fig-showtext: true`), damit showtext-Schriften korrekt gerendert werden.

Formatierter Text mit ggtext

Standard-ggplot2-Textelemente unterstuetzen nur einfachen Text - man kann keine fetten, kursiven oder farbigen Woerter innerhalb eines Titels verwenden. Das {ggtext}-Paket hebt

diese Einschraenkung auf, indem es HTML- und Markdown-Formatierung in Plot-Titeln, Untertiteln, Achsenbeschriftungen und Annotationen erlaubt.

Das Paket stellt zwei Ersatzfunktionen fuer `element_text()` bereit:

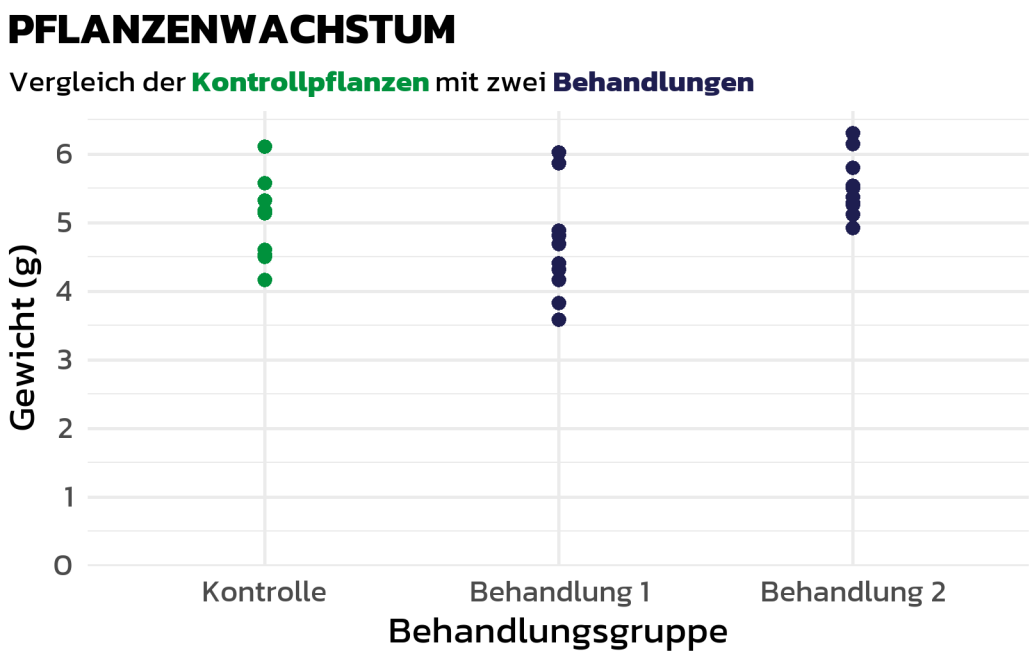
- `ggtext::element_markdown()` — fuer einzeiligen Text (Titel, Achsenbeschriftungen). Rendert HTML/Markdown inline.
- `ggtext::element_textbox_simple()` — fuer mehrzeiligen Text mit automatischem Zeilenumbruch (Untertitel, Bildunterschriften).

Ein normales `element_text()` rendert keine Formatierung — HTML-Tags wuerden als woertlicher Text erscheinen.

Die Formatierung selbst verwendet einfaches HTML: `<b>fett</b>` fuer Fettdruck, `<i>kursiv</i>` fuer Kursivschrift, und `<b style='color:#00923f;'>gruener Fettdruck</b>` fuer farbigen Text. Man muss kein HTML beherrschen — diese wenigen Tags decken die meisten Anwendungsfaelle ab.

Hier ein vollstaendiges Beispiel, das die Legende durch farbcodierten Text im Untertitel ersetzt:

```
myplot +
  aes(color = group) +
  scale_color_manual(
    values = c(ctrl = "#00923f", trt1 = "#201E50", trt2 = "#201E50"),
    guide = "none"
  ) +
  labs(
    title = "PFLANZENWACHSTUM",
    subtitle = "Vergleich der <b style='color:#00923f;'>Kontrollpflanzen</b> mit
zwei <b style='color:#201E50;'>Behandlungen</b>"
  ) +
  theme_minimal(base_size = 14) +
  theme(
    text = element_text(family = "kanit"),
    plot.title = element_text(face = "bold", size = 16),
    plot.title.position = "plot",
    plot.subtitle = ggtext::element_textbox_simple(
      size = 11,
      margin = margin(0, 0, 5, 0)
    )
  )
```



Diese Technik wird besonders maechtig, wenn man die Legende durch farbcodierten Text im Untertitel ersetzt - diesen Ansatz werden wir in den naechsten Kapiteln intensiv nutzen, um visuelle Unordnung zu reduzieren.

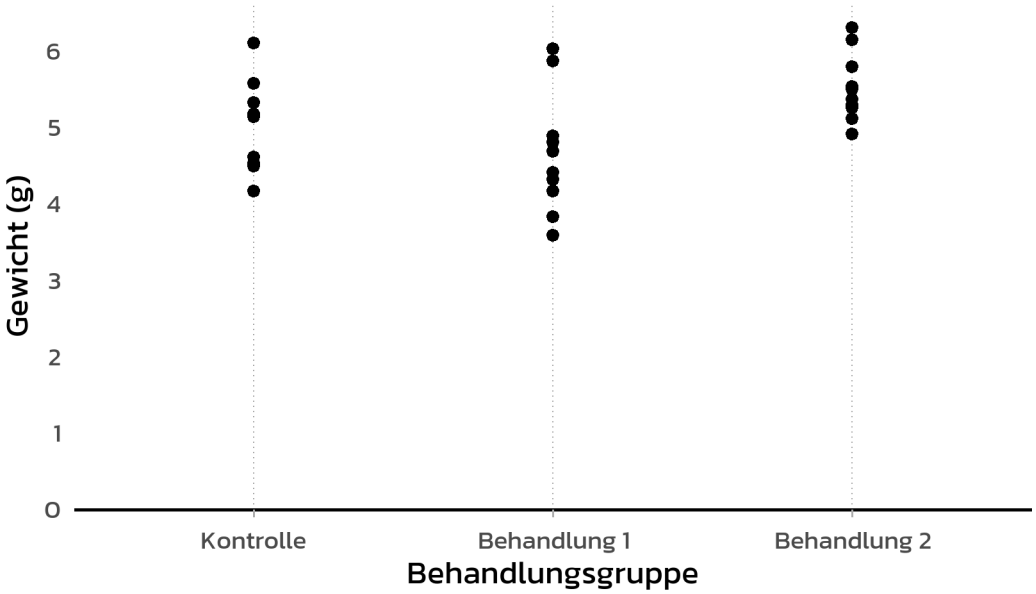
Eigene Theme-Funktion erstellen

Wenn man an einem Projekt mit mehreren Plots arbeitet, stellt man schnell fest, dass dieselben Theme-Einstellungen immer wieder kopiert werden. Das ist sowohl muehsam als auch fehleranfaellig - eine Detailaenderung bedeutet, jeden Plot zu aktualisieren. Ein besserer Ansatz ist, die Theme-Einstellungen in eine eigene Funktion zu verpacken, was den Code DRY (Don’t Repeat Yourself) haelt und visuelle Konsistenz ueber alle Plots sicherstellt:

```
theme_clean <- function(base_size = 12) {
  theme_minimal(base_size = base_size) +
    theme(
      text = element_text(family = "kanit"),
      plot.title.position = "plot",
      plot.title = element_text(face = "bold", size = base_size + 4),
      plot.subtitle = ggtext::element_textbox_simple(
        size = base_size,
        margin = margin(0, 0, 8, 0)
      ),
      panel.grid.minor = element_blank(),
      panel.grid.major.y = element_blank(),
      panel.grid.major.x = element_line(
        linetype = "dotted", color = "#AAAAAA", linewidth = 0.3
      ),
      axis.line.x = element_line(color = "black", linewidth = 0.6),
      axis.ticks.x = element_line(color = "#AAAAAA", linewidth = 0.4)
    )
}

myplot +
  labs(title = "Ein sauberer Look") +
  theme_clean()
```

Ein sauberer Look



Diese `theme_clean()`-Funktion wird in den folgenden Kapiteln wiederverwendet, wenn komplexere Plots erstellt werden.

Bibliography