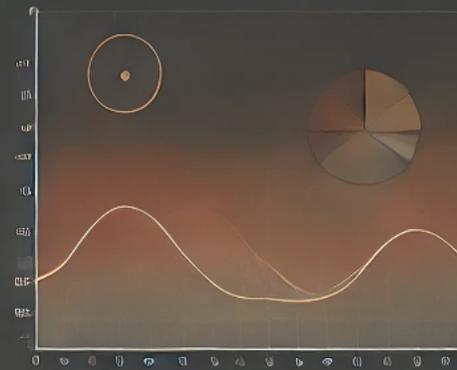
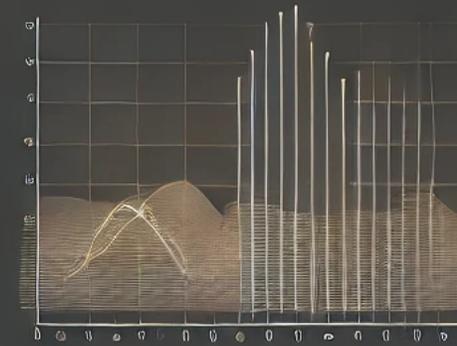
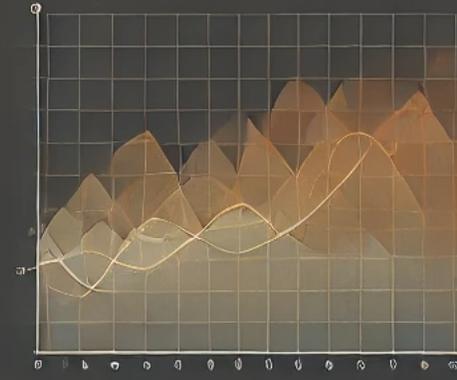
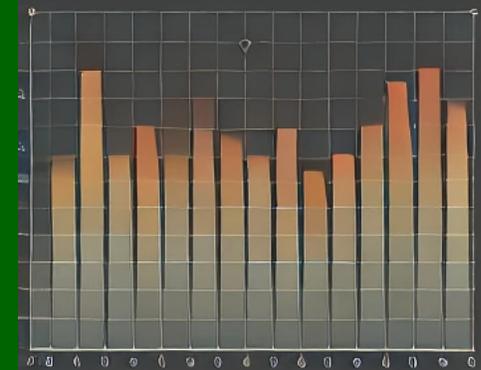
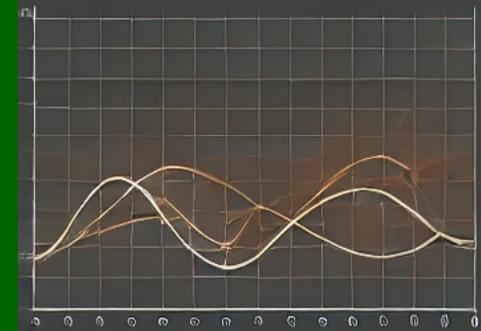
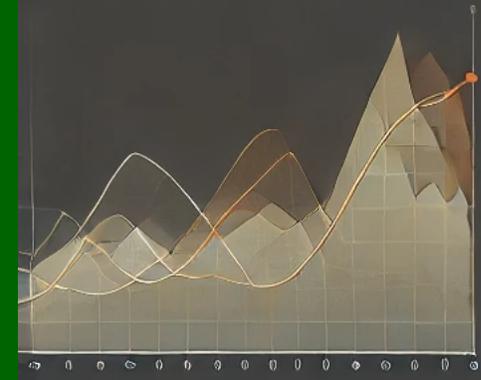


Programmieren in R



Bedingte Anweisungen

In manchen Situationen wollen wir bestimmte Werte, oder `NA`s durch vordefinierte, andere Werte ersetzen. Gegeben wir wollen dies hier tun und die Bevölkerung für Indien immer mit 0 ersetzen

```
data("gapminder")
gapminder_indien <- gapminder |>
  mutate( pop = ifelse(country == "India", 0, pop) )

filter(gapminder_indien, country == "India")
```

```
# A tibble: 12 × 6
  country continent  year lifeExp  pop gdpPercap
  <fct>    <fct>      <int>  <dbl> <dbl>   <dbl>
1 India   Asia      1952   37.4    0     547.
2 India   Asia      1957   40.2    0     590.
3 India   Asia      1962   43.6    0     658.
4 India   Asia      1967   47.2    0     701.
5 India   Asia      1972   50.7    0     724.
6 India   Asia      1977   54.2    0     813.
7 India   Asia      1982   56.6    0     856.
8 India   Asia      1987   58.6    0     977.
9 India   Asia      1992   60.2    0    1164.
10 India  Asia      1997   61.8    0    1459.
11 India  Asia      2002   62.9    0    1747.
12 India  Asia      2007   64.7    0    2452.
```

case_when Anweisung

Wenn wir mehrer Bedingungen auf einmal betrachten müssen, dann hilft uns `case_when` weiter.

Beispielsweise wollen wir für alle Länder mit einem weniger als 50 Jahren Lebenserwartung eine Variable Lebenserwartung generieren, welche den Wert `<50` annimmt, für weniger als 50 Jahre Lebenserwartung, `50-70` mit Lebenserwartung zwischen 50 und 70 Jahren und `>70` bei einer Lebenserwartung mehr als 70 Jahre. Wir betrachten hier alle Länder für das Jahr 2007.

```
gapminder <- gapminder |>
  filter( year == 2007 ) |>
  mutate(Lebenserwartung = case_when(
    lifeExp < 50 ~ "<50",
    lifeExp >= 50 & lifeExp <= 70 ~ "50-70",
    lifeExp > 70 ~ ">70"))

table(gapminder$Lebenserwartung)
```

```
<50    >70  50-70
  19     83    40
```

Funktionen in R

Für Aufgaben, welche wir in R öfter anwenden, wollen wir nicht jedes mal von neuem den gleichen Befehl eingeben.

- ✚ Hier lohnt es sich eine Funktion für den Befehl zu schreiben
- ✚ Durch die Funktion `function` weiß R das nun eine Funktion definiert wird
- ✚ Beispielsweise den Mittelwert berechnen durch `summe(x) / länge(x)`

```
durchschnitt <- function(x) {  
  s <- sum(x)  
  n <- length(x)  
  s/n  
}
```

Da es in R bereits die Funktion `mean` gibt können wir testen ob unsere Funktion zum gleichen Ergebnis kommt wie die in R vordefinierte Funktion:

```
x <- 1:100  
identical(mean(x), durchschnitt(x))
```

```
[1] TRUE
```

Scoping

Variablen welche in R innerhalb einer Funktion definiert werden, werden auch nur in dieser Funktionsumgebung verwendet.

```
s <- 5  
durchschnitt(51:100)
```

```
[1] 75.5
```

```
s
```

```
[1] 5
```

Wenn wir uns nun `s` anschauen, dann ist dies immer noch 5, auch nachdem wir die Funktion `durchschnitt` aufgerufen haben (wird lexikalisches Scoping genannt).

✚ Unterschiede zwischen lexikalischem und dynamischem Scoping finden Sie [hier](#)

For-Schleifen

Nach der Definition einer Funktion wollen wir diese auf mehrere Elemente anwenden. Dies können wir über eine Schleife erreichen.

- ✚ Über eine Schleife können bestimmte Aktionen n mal wiederholt werden.

```
n <- 5
for(i in 1:n) {
  print(i)
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

- ✚ In der For-Schleife wird die Eingabe evaluiert und entsprechende Aktion ausgeführt (hier die Ausgabe der Zahlen von $n = 1, \dots, 5$)
- ✚ Die gleiche Aktion wird 5 mal ausgeführt, es ändert sich nur i

Beispiel für Schleifen in R

Nehmen wir an, dass wir einem Datensatz eine neue Variable hinzufügen möchten (z.B. $\text{gdp}/1000$). Dies können wir innerhalb von `dplyr` über `mutate()` tun:

```
data("gapminder")
gapminder_copy <- gapminder

gapminder_copy <- gapminder_copy %>%
  mutate(gdpPercap_thousands = gdpPercap / 1000)
```

Beispiel für Schleifen in R

Alternativ könnten wir dafür eine for-Schleife verwenden:

```
# Spalte initialisieren
gapminder_copy$gdpPercap_thousands <- NA

# For-Schleife
for (i in 1:nrow(gapminder_copy)) {
  gapminder_copy$gdpPercap_thousands[i] <- gapminder_copy$gdpPercap[i] / 1000
}
```

sapply () und map () als Alternativen zu for-Schleifen

In R sind for-Schleifen recht langsam und es gibt gute Alternativen dazu.

Beispielsweise aus der apply-Familie die `sapply()` Funktion oder aus dem Paket `purrr` die Funktion `map()`.

Um beim obigen Beispiel zu bleiben hier die Funktion mit `sapply()`:

```
gapminder_copy$gdpPerCap_thousands <- sapply(gapminder_copy$gdpPerCap, function(x) x / 1000)
```

Oder mit der `map()` Funktion aus dem `purrr` Paket:

```
library(purrr)
gapminder_copy$gdpPerCap_thousands <- map_dbl(gapminder_copy$gdpPerCap, ~ .x / 1000)
```

Vergleich der einzelnen Methoden

Wir simulieren 1000-mal die Erstellung der zusätzlichen Variable `gdpPerCap`, um die effizienteste Methode zu identifizieren.

```
Unit: microseconds
  expr      min      lq      mean      median      uq      max  neval
For_Loop 29627.806 32806.7300 34131.5466 33890.3620 34839.9445 88010.909 1000
  Sapply   600.243    670.4765   774.2413   707.6480   806.7425   3550.324 1000
  Map     935.721   1074.5675  1190.4705  1125.7895  1172.4345  3785.429 1000
  Mutate  440.970    602.4120   743.0555   765.2495   844.6605   6625.666 1000
```

- + Die for-Schleife ist in R sehr ineffizient.
- + `sapply` ist eine deutlich effizientere Möglichkeit um mit Schleifen zu arbeiten
- + Doch am Effizientesten ist es die vektorisierten Befehle innerhalb von `dplyr` zu verwenden (hier `mutate()`)