

Einführung in R

Fortbildung im Institut der deutschen Wirtschaft

7 Funktionen & Iteration

Inhalte und Ziele der Sitzung

- Funktionen im Tidyverse (mit dplyr)
- Moderne Iteration im Tidyverse
- Leseempfehlung:
 - [Programming with dplyr Vignette](#)

Funktionen: Basics

Wann und warum eigene Funktionen?

- Eine Daumenregel: Mehr als dreimal sollte derselbe Code nicht von Hand (»hardgecodet«) geschrieben, beziehungsweise kopiert werden.
- Funktionen haben drei große Vorteile gegenüber Copy-and-paste:
 1. Funktionen können (und sollten) einen ausdrucksstarken Namen haben, das macht den Code besser lesbar.
 2. Wenn die Anforderungen an den Code sich verändern, müssen wir nur an einer Stelle etwas ändern, anstatt an vielen Stellen.
 3. Wir verhindern Folgefehler, die entstehen wenn Code kopiert wird.

Aufbau einer Funktion

```
mein_funktionsname <- function( Argument(e) ) {  
    Ergebnis <- Mach dies, mach jenes  
    ...  
    return(Ergebnis)  
}
```

- `mein_funktionsname`: (optional) Name, mit dem die Funktion als Objekt im Environment gespeichert wird.
- `function() {}`: Grundgerüst, um eine Funktion zu definieren.
- `Argument(e)`: Argumente, die mit Inputs gefüllt werden können.
- `Mach dies, mach jenes`: Dinge, die die Funktion macht.
- `Ergebnis <-`: Dinge, die die Funktion gemacht hat werden gespeichert.
- `return(Ergebnis)`: Dinge, die die Funktion gemacht hat werden ausgegeben.

Ein simples Beispiel für eine Funktion

- Wir möchten eine Funktion erstellen, die zu jedem Element eines numerischen Vektors Eins addiert.
- Übersetzung in Funktion:
 - Klingender Funktionsname z. B.: `plus_eins`
 - Argument, das den Platzhalter für den Input darstellt, z. B.: `numerischer_vektor`
 - Was die Funktion macht: `numerischer_vektor + 1`
 - Name für Ergebnis, z.B.: `numerischer_vektor_plus_eins`
 - Ausgabe des Ergebnisses: `return(numerischer_vektor_plus_eins)`

- Um eine Funktion anwenden zu können brauchen wir immer drei Dinge:
 1. Daten
 - Auf was wollen wir die Funktion anwenden?
 2. Definieren der Funktion
 - Was soll die Funktion mit unserem Input / unseren Daten machen?
 3. Call der Funktion
 - Schließlich wenden wir die Funktion auf unseren Input an.

Drei Schritte zum Anwenden einer Funktion

1. Daten:

```
(daten <- 1:10)
```

```
#> [1] 1 2 3 4 5 6 7 8 9 10
```

2. Definieren der Funktion:

```
plus_eins <- function(numerischer_vektor) {  
  numerischer_vektor_plus_eins <- numerischer_vektor + 1  
  return(numerischer_vektor_plus_eins)  
}
```

3. Call der Funktion

```
plus_eins(daten) # oder x |> plus_eins()
```

```
#> [1] 2 3 4 5 6 7 8 9 10 11
```

- Bei der explorativen Datenanalyse, zum Beispiel beim Erstellen von deskriptiven Statistiken, wenden wir häufig die gleichen Funktionen nacheinander an.
- Wenn wir den Anteil der Ausprägung einer kategorialen Variable an allen Ausprägungen der Variable bestimmen möchten, dann sind die Berechnungsschritte immer dieselben, zuerst zählen wir mit `count()`, dann berechnen wir den Anteil als Anzahl der Ausprägung in Relation zur Summe der Anzahlen der Ausprägungen: `mutate(share = n / sum(n))`.
- Wir könnten eine eigene Funktion schreiben, die diese beiden Schritte für uns durchführt.

Beispieldatensatz

```
(gapminder <- gapminder::gapminder)
```

```
#> # A tibble: 1,704 x 6
#>   country      continent  year lifeExp      pop gdpPercap
#>   <fct>      <fct>      <int>  <dbl>    <int>    <dbl>
#> 1 Afghanistan Asia      1952   28.8  8425333    779.
#> 2 Afghanistan Asia      1957   30.3  9240934    821.
#> 3 Afghanistan Asia      1962   32.0 10267083    853.
#> 4 Afghanistan Asia      1967   34.0 11537966    836.
#> 5 Afghanistan Asia      1972   36.1 13079460    740.
#> 6 Afghanistan Asia      1977   38.4 14880372    786.
#> 7 Afghanistan Asia      1982   39.9 12881816    978.
#> 8 Afghanistan Asia      1987   40.8 13867957    852.
#> 9 Afghanistan Asia      1992   41.7 16317921    649.
#> 10 Afghanistan Asia      1997   41.8 22227415    635.
#> # ... with 1,694 more rows
```

Wiederholung der Analyseschritte

```
ergebnis <- gapminder |> #Datensatz  
  count(continent) |> #was die Funktion...  
  mutate(share = n/sum(n)) #...macht
```

```
ergebnis #das Ergebnis der angewendeten Funktion anschauen
```

```
#> # A tibble: 5 x 3  
#>   continent     n share  
#>   <fct>      <int> <dbl>  
#> 1 Africa        624 0.366  
#> 2 Americas      300 0.176  
#> 3 Asia          396 0.232  
#> 4 Europe        360 0.211  
#> 5 Oceania        24 0.0141
```

- Was aber, wenn wir den Anteil der Länder ermitteln wollen?
- Wir könnten den oberen Code Copy-pasten und `continent` durch `country` ersetzen.
- Oder wir schreiben eine Funktion, die als Argument verschiedene Spalten annimmt.

Erster Versuch

- Funktion:

```
my_share_function <- function(cat_var) {  
  ergebnis <- gapminder |>  
    count(cat_var) |>  
    mutate(share = n/sum(n))  
  return(ergebnis)  
}
```

- Funktion anwenden:

```
my_share_function(country)
```

- Funktioniert nicht!

- Die vorangegangene Funktion funktioniert nicht, weil die Funktion `count()` versucht eine Variable zu evaluieren, die tatsächlich `cat_var` heißt.
- Dieses Problem tritt auf, weil wir mit den dplyr-Funktionen arbeiten.
- Diese nutzen **Data masking**.
- Wir können dieses Problem umgehen, indem wir die Variable mit zwei geschweiften Klammern `{{ }}` umschließen.

Zweiter Versuch

- Funktion:

```
my_share_function <- function(cat_var) {  
  ergebnis <- gapminder |>  
    count({{ cat_var }}) |>  
    mutate(share = n/sum(n))  
  return(ergebnis)  
}
```

Zweiter Versuch

- Die Funktion funktioniert jetzt auch mit der Variable `country` ...

```
my_share_function(country)
```

```
#> # A tibble: 142 x 3
#>   country      n  share
#>   <fct>    <int> <dbl>
#> 1 Afghanistan    12 0.00704
#> 2 Albania         12 0.00704
#> 3 Algeria         12 0.00704
#> 4 Angola          12 0.00704
#> 5 Argentina       12 0.00704
#> 6 Australia       12 0.00704
#> 7 Austria         12 0.00704
#> 8 Bahrain         12 0.00704
#> 9 Bangladesh      12 0.00704
#> 10 Belgium        12 0.00704
#> # ... with 132 more rows
```


Zweiter Versuch

- ...genauso wie mit anderen Variablen:

```
my_share_function(year)
```

```
#> # A tibble: 12 x 3
#>   year      n share
#>   <int> <int> <dbl>
#> 1  1952   142 0.0833
#> 2  1957   142 0.0833
#> 3  1962   142 0.0833
#> 4  1967   142 0.0833
#> 5  1972   142 0.0833
#> 6  1977   142 0.0833
#> 7  1982   142 0.0833
#> 8  1987   142 0.0833
#> 9  1992   142 0.0833
#> 10 1997   142 0.0833
#> 11 2002   142 0.0833
#> 12 2007   142 0.0833
```

Noch mehr Schreibarbeit sparen

- Unsere `my_share_function` ist zwar schön und gut, allerdings spezifisch für das Objekt, das wir `gapminder` genannt haben geschrieben.
- Was, wenn wir die Funktion allgemeiner für Datensätze unserer Wahl schreiben möchten?
- Wir verändern die Funktion schlichtweg so, dass wir ein zweites Argument einfügen:

```
my_share_function_general <- function(my_data, cat_var) {  
  ergebnis <- my_data |> #Das Data-Argument benutzt kein Data-masking.  
    count({{ cat_var }}) |>  
    mutate(share = n/sum(n))  
  return(ergebnis)  
}
```

Auf den gapminder-Datensatz:

```
my_share_function_general(gapminder, continent)
```

```
#> # A tibble: 5 x 3  
#>   continent      n share  
#>   <fct>      <int> <dbl>  
#> 1 Africa        624 0.366  
#> 2 Americas      300 0.176  
#> 3 Asia          396 0.232  
#> 4 Europe        360 0.211  
#> 5 Oceania        24 0.0141
```

Einfach übertragbar auf andere Datensätze:

```
my_share_function_general(palmerpenguins::penguins, species)
```

```
#> # A tibble: 3 x 3  
#>   species      n share  
#>   <fct>    <int> <dbl>  
#> 1 Adelie    152  0.442  
#> 2 Chinstrap  68  0.198  
#> 3 Gentoo   124  0.360
```

Funktion für Anteile nach Gruppen (Kreuztabelle)

- Zur Erinnerung, die Funktion für einen **bestimmten** Datensatz und zwei **bestimmte** Variablen:

```
palmerpenguins::penguins |>
  count(year, species) |>
  group_by(year) |>
  mutate(share = n/sum(n)) |>
  ungroup() |>
  select(-n) |> #Entfernen der absoluten Werte
  pivot_wider(names_from = "species", id_cols = "year", values_from = "share")
```

```
#> # A tibble: 3 x 4
#>   year Adelie Chinstrap Gentoo
#>   <int>   <dbl>     <dbl>   <dbl>
#> 1  2007  0.455     0.236   0.309
#> 2  2008  0.439     0.158   0.404
#> 3  2009  0.433     0.2     0.367
```

- Übersetzen in eine Funktion:

```
my_share_cross_tabulation_function <- function(data, var1, var2) {  
  result <- data |>  
    count({{ var1 }}, {{ var2 }}) |>  
    group_by({{ var1 }} ) |>  
    mutate(share = n/sum(n)) |>  
    ungroup() |>  
    select(-n) |>  
    pivot_wider(id_cols = {{ var1 }}, names_from = {{ var2 }}, values_from = "share")  
  return(result)  
}
```

- *Hinweis: Funktionen sollten eigentlich einen Kommentar enthalten, in dem die Funktionsweise und die verschiedenen Argumente beschrieben werden.*

Anwenden der Funktion für Anteile in Kreuztabellen

- Egal ob Pinguin...

```
my_share_cross_tabulation_function(palmerpenguins::penguins, year, species)
```

```
#> # A tibble: 3 x 4  
#>   year Adelie Chinstrap Gentoo  
#>   <int> <dbl>    <dbl> <dbl>  
#> 1  2007  0.455    0.236  0.309  
#> 2  2008  0.439    0.158  0.404  
#> 3  2009  0.433    0.2    0.367
```

Anwenden der Funktion für Anteile in Kreuztabellen

- ...oder Kontinent:

```
my_share_cross_tabulation_function(gapminder::gapminder, year, continent)
```

```
#> # A tibble: 12 x 6
#>   year Africa Americas Asia Europe Oceania
#>   <int> <dbl>    <dbl> <dbl> <dbl>    <dbl>
#> 1  1952  0.366    0.176 0.232  0.211  0.0141
#> 2  1957  0.366    0.176 0.232  0.211  0.0141
#> 3  1962  0.366    0.176 0.232  0.211  0.0141
#> 4  1967  0.366    0.176 0.232  0.211  0.0141
#> 5  1972  0.366    0.176 0.232  0.211  0.0141
#> 6  1977  0.366    0.176 0.232  0.211  0.0141
#> 7  1982  0.366    0.176 0.232  0.211  0.0141
#> 8  1987  0.366    0.176 0.232  0.211  0.0141
#> 9  1992  0.366    0.176 0.232  0.211  0.0141
#> 10 1997  0.366    0.176 0.232  0.211  0.0141
#> 11 2002  0.366    0.176 0.232  0.211  0.0141
#> 12 2007  0.366    0.176 0.232  0.211  0.0141
```


Iteration

Funktionen + Iteration =

- Funktionen entfalten ihr wahres Potenzial wenn sie iterativ angewendet werden.
- Die moderne Syntax, um Funktionen iterativ anzuwenden sind die `map`-Funktionen aus dem Paket `purrr`, das Teil des Tidyverse ist.
- Leseempfehlungen:
 - [Learn to purrr](#)
 - [purrr cheat sheet](#)
 - [R4DS, Kapitel 21](#)

- Eine `map`-Funktion wendet dieselbe Funktion auf jedes Element eines Objekts an, z. B. auf jeden Eintrag einer Liste, eines Vektors oder jede Spalte eines Tibbles.
- Die verschiedenen `map`-Funktionen sind nach dem Typ des **Outputs** benannt, z. B.:

Funktion	Beschreibung
<code>map()</code>	Hauptfunktion; Output ist eine Liste
<code>map_df()</code>	Output ist ein Tibble (Dataframe)
<code>map_dbl()</code>	Output ist ein numerischer (double) Vektor
<code>map_chr()</code>	Output ist ein character-Vektor
<code>map_lgl()</code>	Output ist ein logischer Vektor

- *Hinweis: Es gibt noch weitere `map`-Funktionen.*

map () - simples Anwendungsbeispiel: Tibble

- map -Funktion iterativ über alle Spalten eines Datensatzes:

```
gapminder::gapminder |>  
  map_df(mean)
```

```
#> # A tibble: 1 x 6  
#>   country continent  year lifeExp      pop gdpPercap  
#>   <dbl>      <dbl> <dbl>   <dbl>    <dbl>    <dbl>  
#> 1      NA          NA 1980.    59.5 29601212.    7215.
```

Funktion: Vektor

- `map`-Funktion iteriert über alle Elemente eines Vektors:
- Eigene Funktion:

```
eigene_funktion <- function(input_vektor) {  
  set.seed(12345)  
  
  rnorm(n = 1, mean = 12, sd = 3) * input_vektor |>  
    log()  
}
```

map () - simples Anwendungsbeispiel mit eigener




Funktion: Vektor

- Anwenden der Funktion:

```
1:10 |>  
    map_dbl(.f = eigene_funktion)
```

```
#> [1] 0.000000 9.535339 15.113155 19.070678 22.140372 24.648494 26.769081 28.606017 30.226310 31.675
```

Genug der Theorie. Ab nach  Studio®.