IQB · METHODEN-TEAM

# eatPrepTBA

## Antwortdaten, Logdaten und Codebücher

Karoline Sachse · 26.10.2026

1 Paket & Kodierung · 2 Studie & Skalierung · 3 Logs & Codebücher



# 0 · Einstieg

| Abschnitt | Inhalt |
|---|---|
| **0 · Einstieg** | Materialien, Installation und Ablauf |
| **1 · Paket und erste Kodierung** | Datenabruf, R-Objekte und Beispiel 1: Rechenaufgabe mit Teilpunkten |
| **2 · Studie und Skalierung** | Beispiel 2: manuelle Codes, Missings, Kennwerte und Skalierungsdatensatz |
| **3 · Logs und Codebücher** | Ereignisse, Zeiten, Fokusverluste und Codebuchexporte |

Nach jedem Arbeitsblock: **ca. 10 Minuten zum Ausprobieren**, anschließend Fragen.

# Materialien öffnen und herunterladen

sachseka.github.io/eatPrepTBAworkshop

- Folien im Browser und Handout zum Nachlesen.
- Materialpaket herunterladen: nur `Syntaxen/`, `Daten/` und `WSeatPrepTBA.Rproj`.
- ZIP vollständig entpacken und `WSeatPrepTBA.Rproj` in RStudio öffnen.

00_installation.R öffnen: Pakete bei Bedarf installieren, Versionen abfragen und Pakete laden.

Folien separat als PDF herunterladen

# Materialien und Skripte

**Synthetische Beispieldaten** ermöglichen alle lokalen Schritte ohne eigenen Zugang. Das Handout enthält Code, Ausgaben und Erläuterungen.

| Datei in `Syntaxen/` | Verwendung |
|---|---|
| 00_installation.R | Pakete installieren, Versionen abfragen und Pakete laden |
| 01_datenabruf.R | Daten aus IQB-Studio und Testcenter beziehen |
| 02_daten_verstehen.R | JSON, vorhandene Codes und Rechenaufgabe |
| 03_datenaufbereitung.R | Vollständige Aufbereitung der kleinen Studie |
| 04_logs.R | Ereignisse, Zeiten und Fokusverluste |
| 05_kodiermanuale.R | Lokales Codebuch als Excel und CSV |
| 06_kodiermanuale_studio.R | Codebuch und DOCX aus IQB-Studio |

Lokale Beispiele: 02–05. Die Vorlagen 01 und 06 benötigen eigene Zugänge.

# 1 · Paket und erste Kodierung

**Vom Datenabruf zur ersten selbst kodierten Antwort.**

Paket und Schnittstellen kennenlernen, Antworten und Kodierschema in R ansehen und das Rechenbeispiel mit Teilpunkten kodieren.

Skripte: 01_datenabruf.R · 02_daten_verstehen.R

# eatPrepTBA und eatAutoCode




**eatPrepTBA** bezieht Antworten und Metadaten, verbindet sie mit dem Testdesign und bereitet Auswertungen und Exporte vor.




**eatAutoCode** führt den IQB-Autocoder lokal in R aus. Die Regeln und Ableitungen stammen aus dem **Kodierschema**.

`eatPrepTBA::code_responses()` übergibt Antworten und Kodierschemata an eatAutoCode. Zurück kommen Werte, **Codes, Scores und Kodierstatus**.

Im Workshop `library(eatPrepTBA)` laden; eatAutoCode wird intern mitgeladen.

# Rückmeldungen sind ausdrücklich willkommen

**Auch kleine Unklarheiten und umständliche Schritte bitte melden.**

Unverständliche Funktionsnamen, irritierende Ausgaben, fehlende Beispiele, Fehlermeldungen oder unnötige Umwege: Jedes Feedback hilft, beide Pakete verständlicher und leichter nutzbar zu machen.

- Datenabruf und Aufbereitung: eatPrepTBA-Issues.
- Automatische Kodierung in R: eatAutoCode-Issues.
- Zu Folien und Skripten: Workshop-Issues.

Eine kurze Beschreibung reicht als Einstieg. Bei Fehlern helfen Paketversion, Aufruf und Fehlermeldung; vertrauliche Daten bitte weglassen.

# Wie kommuniziert R mit den Webanwendungen?

Die **API** ist die Schnittstelle, über die R Daten aus IQB-Studio oder Testcenter anfordert.




- **eatPrepTBA** wählt den passenden Endpunkt, etwa die Units eines Workspaces. Die Anmeldung liefert einen Sitzungstoken für weitere Anfragen.
- httr2 sendet die HTTP-Anfragen und empfängt die Antworten.
- **eatPrepTBA** bereitet die Antwort für R auf: JSON wird zunächst zu Listen, daraus entstehen Tabellen; einige Spalten bleiben verschachtelt.

API-Einführung: Kommunikation mit IQB-Studio und Testcenter

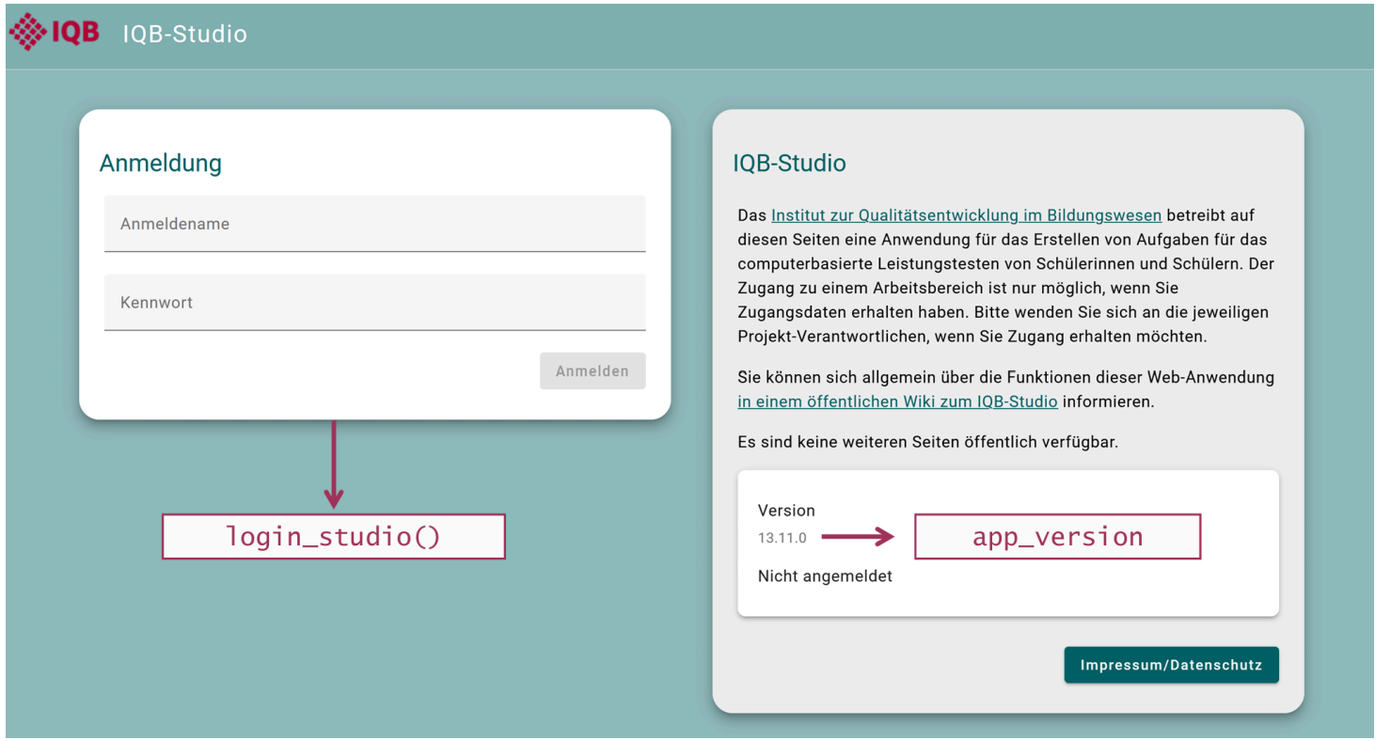
# Datenquellen und Einlesefunktionen

| Quelle | Benötigte Inhalte | Funktionen |
|---|---|---|
| **IQB-Studio** | Variablen, Kodierschema, Item-Verknüpfungen, Aufgabenpositionen | `get_units()` |
| **Testcenter: Antworten** | gespeicherte Eingaben | `get_responses()` oder `read_responses()` |
| **Testcenter: Testdesign** | vorgesehene Personen, Booklets und Reihenfolge | `get_design()` |
| **Testcenter: Logs** | Ereignisse während der Durchführung | `get_logs()` oder `read_logs()` |

`get_*()` ruft Daten über die **API** ab, die Schnittstelle zur Anwendung. `read_*()` liest vorhandene Exportdateien. Danach liegen R-Objekte vor.

Antwort- und Logexporte ersetzen weder die Aufgabeninformationen noch das vollständige Testdesign.

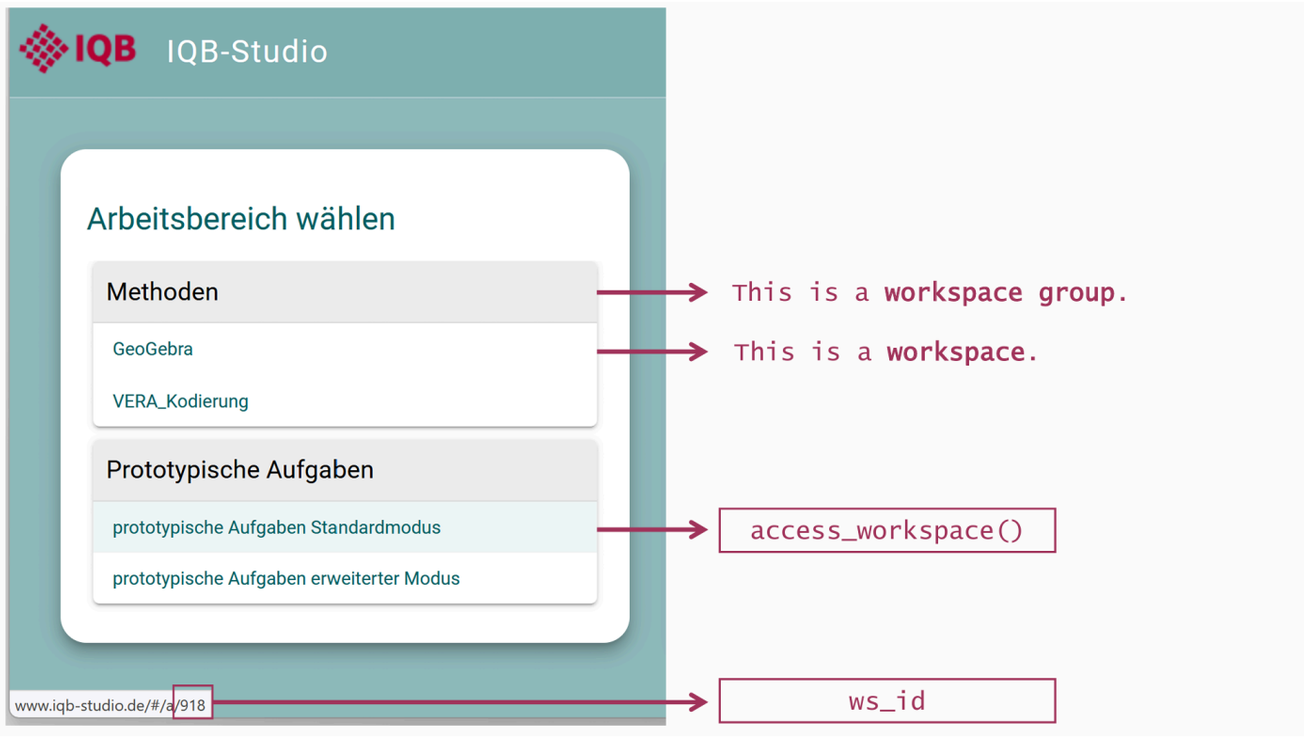
# IQB-Studio: die Version ablesen




Die Versionsangabe der **eigenen Anmeldeseite** vollständig übernehmen, einschließlich eines möglichen Beta-Zusatzes.

Screenshot einer älteren Oberfläche; die abgebildete Version ist ein Beispiel.

# IQB-Studio: die Workspace-ID finden




Workspace auswählen und die Zahl in der Browseradresse ablesen: Im Beispiel bedeutet …/#/a/918 die **Workspace-ID 918**.

Vignette: Workspace auswählen und ID ermitteln

# Eigene Zugänge: Einstellungen zuerst anpassen

Die Version steht auf der IQB-Studio-Anmeldeseite. Die Workspace-IDs 1300 und 35 sind Beispiele und werden durch die eigenen IDs ersetzt.

```r
library(eatPrepTBA)

studio_version <- "21.0.0" # genaue Version der Anmeldeseite übernehmen
studio_workspace_id <- 1300 # Beispiel; eigenen Workspace eintragen
testcenter_url <- "https://iqb-testcenter2.de/" # Beispiel
testcenter_workspace_id <- 35 # Beispiel; eigenen Workspace eintragen
```

Skript: 01_datenabruf.R

# IQB-Studio: Aufgabeninformationen laden

Ohne `base_url` verwendet `login_studio()` die Installation **www.iqb-studio.de**. Danach den Workspace mit den Aufgaben auswählen.

```r
dir.create("Daten/EigeneDaten", showWarnings = FALSE)
```

```r
studio_login <- login_studio(app_version = studio_version, keyring = TRUE)
studio <- access_workspace(studio_login, ws_id = studio_workspace_id)

units <- get_units(studio, metadata = TRUE, unit_definition = TRUE)
saveRDS(units, "Daten/EigeneDaten/units.rds")
```

`metadata = TRUE` bereitet die Metadaten auf. `unit_definition = TRUE` lädt zusätzlich die technische Aufgabendefinition.

Vorlage für eigene Zugänge: 01_datenabruf.R

# Anmeldung: Passwortspeicher und Fehlermeldungen

`keyring = TRUE` speichert die Zugangsdaten lokal im Passwortspeicher des Betriebssystems und verwendet sie bei weiteren Anmeldungen erneut.

| Beobachtung | Nächster Schritt |
|---|---|
| **HTTP 401 Unauthorized** | Installation und Zugangsdaten prüfen. Mit `keyring = FALSE` einmal frisch eingeben. |
| Gespeicherte Zugangsdaten sind veraltet | Einmal `keyring = TRUE, change_key = TRUE` verwenden, um sie zu ersetzen. |
| **HTTP 521** | `app_version` auf die genaue Version der Anmeldeseite setzen, einschließlich eines möglichen Beta-Zusatzes. |

Nach einem IQB-Studio-Update erneut anmelden und das Workspace-Objekt neu erstellen.

Die auskommentierten Aufrufe stehen in Syntaxen/01_datenabruf.R.

# Kodierschema vorhanden – wozu `add_coding_scheme()`?

`get_units()` lädt das Kodierschema bereits in der Spalte `coding_scheme`.

`add_coding_scheme()` macht daraus zusätzlich die verschachtelte Tabelle `unit_codes`. Darin lassen sich Variablen und Kodierregeln gezielt ansehen.

```
1 units <- add_coding_scheme(units) # coding_scheme wird zu unit_codes aufbereitet.
2 View(units)
```

Dieser Schritt ist zum Nachsehen nützlich. `code_responses()` mit `prepare = TRUE` bereitet das Kodierschema bei Bedarf selbst auf.

`View(units)` öffnet die vollständige Tabelle in RStudio. `units$unit_codes[[1]]` zeigt die Kodierinformationen der ersten Unit.

# Kodierschemata prüfen: `get_coding_report()`

Der Prüfbericht aus IQB-Studio nennt Probleme in den **Kodierschemata**.

```r
coding_report <- get_coding_report(studio)
head(coding_report)
coding_report$validation_problems[[1]]
saveRDS(coding_report, "Daten/EigeneDaten/coding_report.rds")
```

Synthetische Vorschau aus Skript 02; unten sind die Probleme der zweiten Zeile zu sehen.

```
      unit_key variable_id validation coding_type
    PC_RECHNEN          01         OK automatisch
DEMO_PRUEFFALL       SUMME     Fehler automatisch
```

```
          type breaking code
SOURCE_MISSING     TRUE <NA>
```

`SOURCE_MISSING`: fehlende Quellvariable; `breaking = TRUE`: blockierendes Problem.

# Metadaten aufbereiten: `add_metadata()`

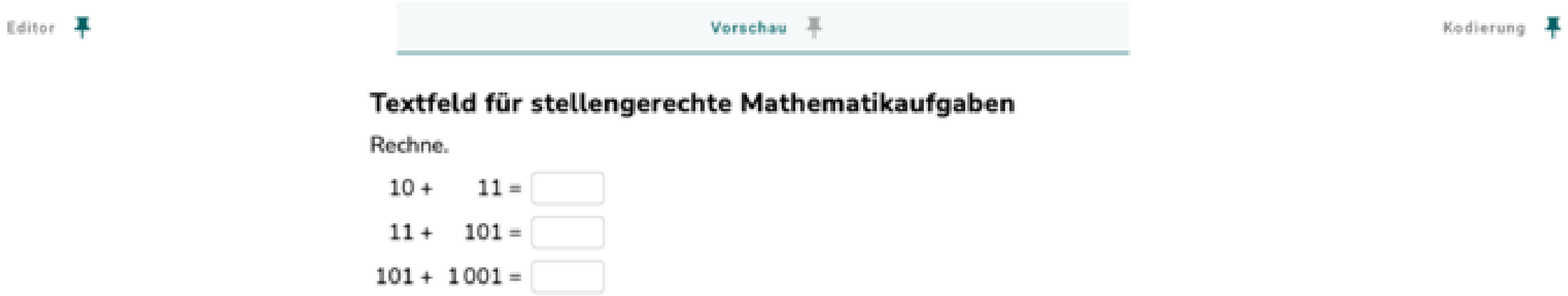
Metadaten beschreiben Aufgaben und Items, beispielsweise **Fach, Klassenstufe und Kompetenz**. `get_units(metadata = TRUE)` liefert die dafür benötigten Angaben.

```r
units_meta <- add_metadata(units)
head(units_meta)
units_meta$unit_metadata[[1]]
units_meta$item_metadata[[1]]
saveRDS(units_meta, "Daten/EigeneDaten/units_metadata.rds")
```

```
# A tibble: 1 × 3
  unit_key   unit_metadata    item_metadata
  <chr>      <list>           <list>
1 PC_RECHNEN <tibble [1 × 2]> <tibble [1 × 5]>
```

Synthetische Ausgabe. Die Felder richten sich nach dem Metadatenprofil; Profildefinitionen und Vokabulare werden bei Bedarf aus dem Internet geladen.

# Eine Unit in IQB-Studio: drei Eingabefelder




Die gesamte Aufgabe ist eine **Unit**. Die drei Eingabefelder heißen `01a`, `01b` und `01c`.

Das synthetische Beispiel verwendet dieselben Variablennamen und die zentralen Kodierregeln.

# Dasselbe Beispiel lokal in R laden

Die Daten wurden passend zum Screenshot synthetisch aufgebaut. Es sind keine aufgezeichneten Serverantworten.

```r
library(eatPrepTBA) # eatAutoCode nicht separat laden.
library(dplyr)
library(tidyr)

pc_units <- readRDS("Daten/demo-pc-units.rds")

select(pc_units, unit_key, unit_label)
```

Zunächst werden Aufgabeninformationen, Metadaten und Kodierschema betrachtet. Die Antworten folgen im nächsten Teil.

Skript: 02_daten_verstehen.R

# Was steckt in der Unit-Tabelle?

Eine Zeile steht für eine Unit. Die Spalten enthalten Werte, JSON oder Tabellen.

```
pc_units |>
  select(unit_key, unit_label, coding_scheme, unit_variables, items_list) |>
  print(width = 90)
names(pc_units)
```

```
# A tibble: 1 × 5
  unit_key   unit_label
  <chr>      <chr>
1 PC_RECHNEN Rechenpaeckchen
  coding_scheme                                          unit_variables items_list
  <chr>                                                  <list>         <list>
1 "{\"variableCodings\":[{\"id\":\"01a\",\"alias\":\"01… <tibble>       <tibble>
```

```
[1] "ws_id"          "ws_label"       "unit_id"        "unit_key"
[5] "unit_label"     "coding_scheme"  "unit_variables" "items_list"
[9] "unit_codes"
```

`coding_scheme`: Kodierschema als JSON. `unit_variables`: Antwortvariablen. `items_list`: Verknüpfung mit Auswertungsitems.

# Die Tabellen innerhalb einer Unit öffnen

`[[1]]` öffnet die Tabelle für die erste Unit innerhalb einer Listenspalte.

```
1 print(pc_units$unit_variables[[1]] |> select(variable_id, variable_type))
2 print(pc_units$items_list[[1]] |> select(variable_id, item_id))
```

```
# A tibble: 3 × 2
  variable_id variable_type
  <chr>       <chr>
1 01a         string
2 01b         string
3 01c         string
```

```
# A tibble: 1 × 2
  variable_id item_id
  <chr>       <chr>
1 01          RECHNEN_01
```

In diesem Beispiel wird **01** als Item ausgewertet. Die drei Ausgangsvariablen sind keine zusätzlichen Items.

# Metadaten ohne Serverabruf ansehen

Für den Workshop liegt ein **synthetischer Metadatenauszug** als Datei vor.

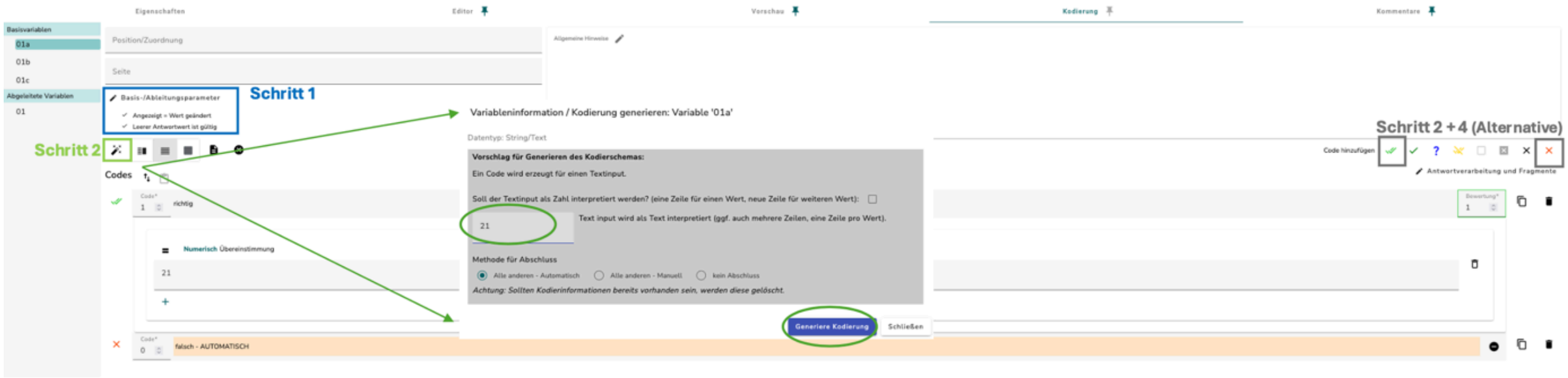
```r
metadata_demo <- readRDS("Daten/demo-metadata-prepared.rds")
select(metadata_demo, unit_key, unit_metadata, item_metadata)
metadata_demo$unit_metadata[[1]]
metadata_demo$item_metadata[[1]]
```

```
# A tibble: 1 × 2
  Fach       Klassenstufe
  <chr>      <chr>
1 Mathematik 4
```

```
# A tibble: 1 × 5
  item_no variable_id variable_ref item_id    Kompetenz
    <int> <chr>       <chr>        <chr>      <chr>
1       1 01          01           RECHNEN_01 Zahlen und Operationen
```

Skript 02 lädt ebenso den synthetischen Kodierbericht zum Nachsehen. Die echten Abrufe stehen in Skript 01.

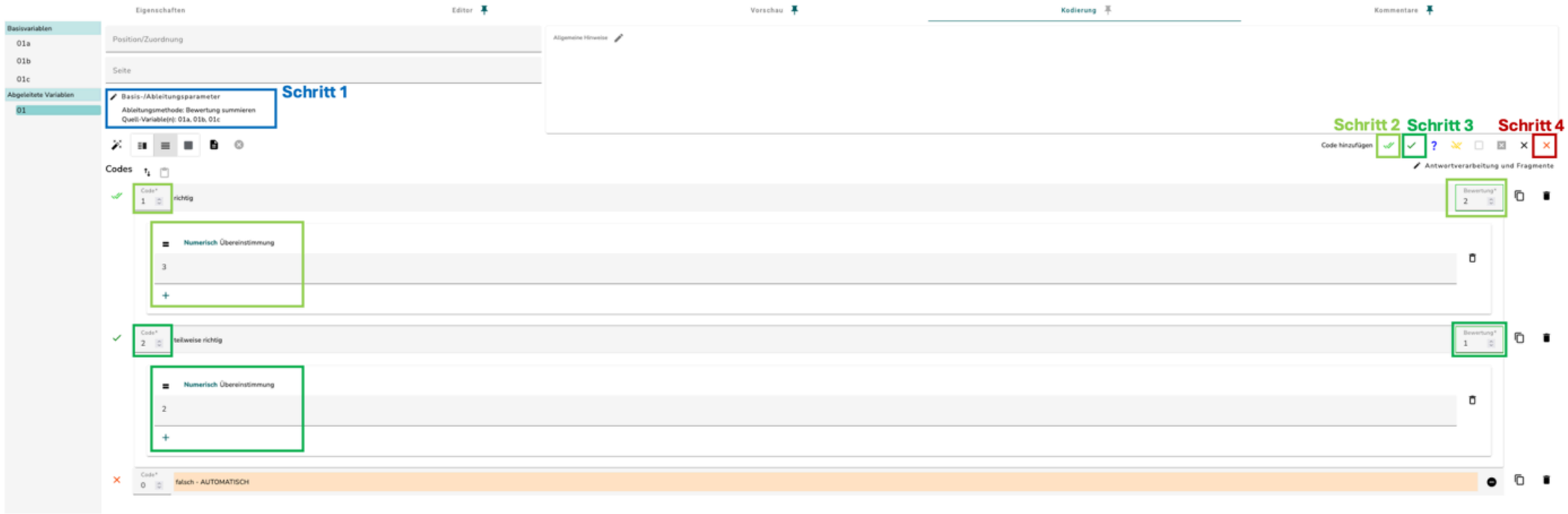
# Basisvariablen und abgeleitete Variable




`01a`, `01b`, `01c` speichern einzelne Antworten. `01` wird später aus deren Bewertungen abgeleitet.

IQB-Studio-Tutorial: Partial Credit am Rechenbeispiel

# Partial Credit: die abgeleitete Variable 01




Die Ableitung summiert die drei Basisbewertungen. Aus der Summe entsteht die Punktzahl für **ein Item**.

# Das Kodierschema in R nachvollziehen

```r
pc_rules <- pc_units |>
  select(unit_key, unit_codes) |>
  unnest(unit_codes) |>
  unnest(variable_codes) |>
  select(variable_id, variable_source_type, code_id, code_score)
pc_rules
```

```
# A tibble: 5 × 4
  variable_id variable_source_type code_id code_score
  <chr>       <chr>                  <int>      <dbl>
1 01a         BASE                       1          1
2 01a         BASE                       0          0
3 01          SUM_SCORE                  1          2
4 01          SUM_SCORE                  2          1
5 01          SUM_SCORE                  0          0
```

01a wird direkt bewertet; 01 erhält aus der Summe 0, 1 oder 2 Punkte. Code und Score sind unterschiedliche Angaben.

# Testcenter: die Workspace-ID finden

In der Administration den Workspace der Durchführung öffnen. Die Zahl nach `/admin/` in der Browseradresse ist dessen ID.

Beispieladresse:

```
https://iqb-testcenter2.de/#/admin/35/files
```

Hier wird `testcenter_workspace_id <- 35` eingetragen. Die tatsächliche ID gehört zur eigenen Durchführung und ist unabhängig von der IQB-Studio-ID.

API-Einführung · Testcenter: Workspace-Ansichten

# Woher kommt der Testmodus?

Der Modus wird in der **Testpersonen-XML** am Element `Login` festgelegt. Beim Einlesen heißt die Spalte **`login_mode`**.

| Abruf | Filter nach Testmodus |
|---|---|
| `get_design()` | Standardmäßig `mode = "run-hot-return"`; bei Bedarf ändern |
| `get_responses()`, `get_logs()` | Kein Modusfilter; `groups` kann die Auswahl auf Gruppen begrenzen |

`run-hot-return` bezeichnet eine reguläre Durchführung: Antworten und Logs speichern, Zeit- und Navigationsbeschränkungen anwenden und dieselbe Testsitzung bei erneuter Anmeldung fortsetzen.

Antworten und Logs können daher zusätzliche Fälle enthalten. Der spätere Abgleich mit dem Design prüft die Zuordnung.

# Testcenter: Antworten abrufen

Den Workspace der Durchführung auswählen und die Antworten im Rohformat sichern.

```r
testcenter_login <- login_testcenter(base_url = testcenter_url)
testcenter <- access_workspace(testcenter_login,
                               ws_id = testcenter_workspace_id)
```

```r
responses <- get_responses(testcenter)
saveRDS(responses, "Daten/EigeneDaten/responses.rds")

# Alternative, wenn bereits ein Antwortexport vorliegt:
# responses <- read_responses("Daten/EigeneDaten/Responses.csv")
# saveRDS(responses, "Daten/EigeneDaten/responses.rds")
```

Skript: 01_datenabruf.R

# Das vollständige Testdesign abrufen

Das vollständige Testdesign beschreibt die vorgesehenen Personen, Booklets und Aufgabenreihenfolgen.

```
design <- get_design(testcenter, units = units)
saveRDS(design, "Daten/EigeneDaten/design.rds")
```

Dazu müssen Testpersonen-XML und zugehörige Booklet-XMLs im Workspace vorliegen. Ein Antwort-CSV allein liefert das vollständige Design nicht.

Skript: 01_datenabruf.R

# Im Antwortexport steckt JSON

Die folgenden Schritte verwenden vier synthetische Antwortmuster für das zuvor betrachtete Kodierschema.

```r
pc_responses <- read_responses("Daten/demo-pc-responses.csv")
select(pc_responses, login_name, unit_key, response_progress)
```

Die Datei ist eine Tabelle. Eine ihrer Zellen enthält die Antworten einer Person auf die Unit als JSON.

```
# A tibble: 2 × 3
  login_name unit_key   responses
  <chr>      <chr>      <chr>
1 D1         PC_RECHNEN "[{\"id\":\"01a\",\"value\":\"21\",\"status\":\"VALUE_CHA…
2 D2         PC_RECHNEN "[{\"id\":\"01a\",\"value\":\"21\",\"status\":\"VALUE_CHA…
```

Im JSON stehen beispielsweise `id`, `status` und `value`. Der Wert 21 ist zunächst eine Eingabe, noch kein Punktwert.

# Eine gespeicherte Antwort genauer ansehen

`{ ... }` beschreibt einen Eintrag mit benannten Feldern; `[ ... ]` eine Folge von Einträgen.

```
1 cat(jsonlite::prettify(pc_responses$responses[[1]]))
```


```
[
    {
        "id": "01a",
        "value": "21",
        "status": "VALUE_CHANGED"
    },
    {
        "id": "01b",
        "value": "112",
        "status": "VALUE_CHANGED"
    },
    {
        "id": "01c",
        "value": "1102",
        "status": "VALUE_CHANGED"
    }
]
```

# Eingaben entpacken oder vorhandene Kodierungen lesen?

Ein Export kann gespeicherte **Eingaben** und bereits vorhandene **Kodierungen** enthalten. Entscheidend ist, welche Inhalte vorliegen – auch bei StarS.

| Inhalt im Export | Funktion | Ergebnis |
|---|---|---|
| `responses`: Eingaben und Bearbeitungsstatus | `prepare_responses()` | Antwortwerte je Variable |
| `coded`: bereits gespeicherte Kodierungen | `prepare_coded()` | vorhandene Codes, Scores und Kodierstatus |

**Beide Funktionen entpacken vorhandene Inhalte.** Neue Codes und Scores entstehen mit `code_responses()` aus Antworten und Kodierschema.

# Rohantworten mit `prepare_responses()` entpacken

Aus dem verschachtelten JSON wird eine Zeile je gespeicherter Variable.

```r
raw_values <- prepare_responses(pc_responses)
select(raw_values, login_name, variable_id, value, variable_status)
```

```
# A tibble: 3 × 4
  login_name variable_id value      variable_status
  <chr>      <chr>       <list>     <chr>
1 D1         01a         <list [1]> VALUE_CHANGED
2 D1         01b         <list [1]> VALUE_CHANGED
3 D1         01c         <list [1]> VALUE_CHANGED
```

`value` ist zunächst eine Listenspalte: Manche Antwortformate enthalten mehrere Werte.

Skript: 02_daten_verstehen.R

# Mit eatAutoCode neu kodieren

`code_responses()` verwendet das Kodierschema und berechnet auch die abgeleitete Variable.

```r
pc_coded <- code_responses(pc_responses, pc_units, prepare = TRUE)
pc_coded |>
  filter(login_name == "D1") |>
  select(variable_id, value, code_id, code_score, code_status)
```

```
# A tibble: 4 × 5
  variable_id value code_id code_score code_status
  <chr>       <chr>   <int>      <int> <chr>
1 01a         21          1          1 CODING_COMPLETE
2 01b         112         1          1 CODING_COMPLETE
3 01c         1102       11          1 CODING_COMPLETE
4 01          3           1          2 CODING_COMPLETE
```

Skript: 02_daten_verstehen.R

# Welche Variable kommt in die Skalierung?

Die Item-Verknüpfung wählt 01 aus. 01a, 01b und 01c bleiben als Ausgangsvariablen nachvollziehbar.

```r
pc_items <- add_item_id(pc_coded, pc_units)
pc_items <- filter(pc_items, !is.na(item_id))
select(pc_items, login_name, item_id, value, code_id, code_score)
```

```
# A tibble: 4 × 5
  login_name item_id    value code_id code_score
  <chr>      <chr>      <chr>   <int>      <int>
1 D1         RECHNEN_01 3           1          2
2 D2         RECHNEN_01 2           2          1
3 D3         RECHNEN_01 1           0          0
4 D4         RECHNEN_01 0           0          0
```

Skript: 02_daten_verstehen.R

# Pause und Ausprobieren

## Ca. 10 Minuten

**Skripte** 00**,** 01 **und** 02 **ansehen.**

In **00** die Paketversionen abfragen und Pakete laden; Installationszeilen nur bei Bedarf ausführen. Anschließend **02** von oben nach unten ausführen und die Ausgaben ansehen.

**01** bleibt die Abrufvorlage für eigene Zugangsdaten. Offene Fragen für die anschließende Runde festhalten.

# Fragen?

# 2 · Studie und Skalierung

**Beispiel 2: die vollständige Aufbereitung einer synthetischen Lesestudie.**

- 40 vorgesehene Personen, 6 Units und 8 Items.
- Zwei Booklets mit teilweise gemeinsamen Aufgaben.
- Automatische, abgeleitete und manuelle Kodierungen.
- Teilpunkte und gezielt fehlende Antworten.

**Einlesen → Kodieren → Design ergänzen → Personen und Items auswählen → Kennwerte → Export**

Skript: 03_datenaufbereitung.R

# R-Projekt öffnen und Pakete laden

Das Arbeitsverzeichnis ist der entpackte Projektordner.

```r
library(eatPrepTBA) # lädt eatAutoCode mit; nicht separat laden
library(dplyr)
library(tidyr)
person_columns <- c("group_id", "login_name", "login_code")

dir.create("Daten/Ergebnisse", showWarnings = FALSE)
```

Alle Pfade beziehen sich auf diesen Ordner. Die Ausgaben landen in `Daten/Ergebnisse/`.

Zum Skript: 03_datenaufbereitung.R

# Synthetische Studiendaten einlesen

```r
units <- readRDS("Daten/units.rds")
design <- readRDS("Daten/design.rds")
responses <- read_responses("Daten/Responses.csv")

# Nach einem API-Abruf alternativ:
# responses <- readRDS("Daten/EigeneDaten/responses.rds")

count(responses, booklet_id)
saveRDS(responses, "Daten/Ergebnisse/01_responses.rds")
```

```
# A tibble: 4 × 4
  login_name booklet_id unit_key response_progress
  <chr>      <chr>      <chr>    <chr>
1 P001       B1         U1       some
2 P001       B1         U2       some
3 P001       B1         U3       some
4 P001       B1         U4       some
```

Eine Zeile gehört zu einer Person und Unit. Die einzelnen Antworten stehen noch im JSON.

# Units und Booklets der Lesestudie

Ausschnitte aus den mitgelieferten synthetischen R-Tabellen:

```
# A tibble: 6 × 2
  unit_key unit_label
  <chr>    <chr>
1 U1       Bibliothek
2 U2       Aushang
3 U3       Zwei Textbelege
4 U4       Begruendung
5 U5       Busfahrplan
6 U6       Museum
```

```
# A tibble: 5 × 3
  booklet_id unit_booklet_no unit_key
  <chr>                <dbl> <chr>
1 B1                       1 U1
2 B1                       2 U2
3 B1                       3 U3
4 B1                       4 U4
5 B1                       5 U5
```

# Eine Designzeile bedeutet: diese Variable ist vorgesehen

Das Design enthält auch Einträge, zu denen später keine Antwort gespeichert wird.

```
# A tibble: 6 × 4
  login_name booklet_id unit_key variable_id
  <chr>      <chr>      <chr>    <chr>
1 P001       B1         U1       Q1
2 P001       B1         U1       Q2
3 P001       B1         U2       Q1
4 P001       B1         U3       Q1
5 P001       B1         U3       Q2
6 P001       B1         U3       TOTAL
```

B1 enthält U1–U5; B2 enthält U1–U4 und U6. Für Personen mit B1 ist U6 deshalb nicht vorgesehen.

# Antworten mit dem Testdesign abgleichen

```r
response_keys <- c("group_id", "login_name", "login_code",
                   "booklet_id", "unit_key")

unknown_units <- anti_join(responses, units, by = "unit_key")
outside_design <- anti_join(responses, design, by = response_keys)
missing_rows <- anti_join(design, responses, by = response_keys)
```

```
# A tibble: 3 × 2
  Fall                      Anzahl
  <chr>                      <int>
1 Antwort ohne Unit              0
2 Antwort außerhalb Design       0
3 Designzeile ohne Antwort      11
```

Vorgesehene Einträge ohne Antwort werden später ergänzt. Antworten ohne passende Unit oder Designzuordnung erfordern einen Abgleich der Datenstände.

# code_responses(): Eingang und Ausgabe

| Argument | Bedeutung |
|---|---|
| `responses` | Ursprüngliche Antworten aus `get_responses()` oder `read_responses()`; noch nicht entpackt |
| `units` | Passende Aufgabeninformationen mit `coding_scheme` und `unit_variables` |
| `prepare = FALSE` | Kodiert neu; `value` bleibt eine Listenspalte |
| `prepare = TRUE` | Entpackt anschließend `value` und ergänzt unter anderem `variable_source_type` und `code_type` |

Im folgenden Ablauf: **`prepare = TRUE`**.

`prepare_coded()` liest dagegen bereits gespeicherte Kodierungen aus; es führt keine neue Kodierung durch.

# code_responses(): weitere Argumente

| Argument | Bedeutung |
|---|---|
| `codes_manual = NULL` | Optional: Tabelle mit manuellen Codes. Scores stammen aus dem Kodierschema; abhängige Variablen werden neu berechnet. |
| `missings = NULL` | Optional: Zuordnung manueller Missing-Codes mit `code_id`, `code_status`, `code_score`, `code_type`. Ohne Angabe gelten Paketvoreinstellungen. |
| `overwrite = FALSE` | Verwendet vorhandene `unit_codes`. Bei `TRUE` werden diese Metadaten erneut aus dem Kodierschema aufgebaut. |

`missings` betrifft hier das Einsetzen manueller Codes. Die endgültige Missing-Einordnung folgt mit `complete_design()`.

`code_responses()`: vollständige Hilfe

# Automatisch kodieren

```r
coded_auto <- code_responses(responses, units, prepare = TRUE)
count(coded_auto, unit_key, code_status)
saveRDS(coded_auto, "Daten/Ergebnisse/02_coded_auto.rds")
```

```
# A tibble: 4 × 5
  unit_key variable_id code_id code_score code_status
  <chr>    <chr>          <int>      <dbl> <chr>
1 U3       Q1                 0          0 CODING_COMPLETE
2 U3       Q2                 0          0 CODING_COMPLETE
3 U3       TOTAL              0          0 CODING_COMPLETE
4 U4       TEXT              NA         NA CODING_INCOMPLETE
```

U3/TOTAL ist bereits abgeleitet; U4/TEXT benötigt eine manuelle Bewertung.

# Manuelle Codes: Freitextantworten bewerten

```r
codes_manual <- readr::read_csv("Daten/codes_manual.csv",
  col_types = readr::cols(.default = "c", code_id = "i"))
```

```r
codes_manual |>
  select(login_name, value, code_id) |>
  head(3) |> print(width = 100)
```

```
# A tibble: 3 × 3
  login_name value                                          code_id
  <chr>      <chr>                                            <int>
1 P001       Die Bibliothek ist am Samstag offen.                 1
2 P002       Ja, laut Aushang: Samstag 10 bis 14 Uhr.             2
3 P003       Nein, am Wochenende ist immer geschlossen.           0
```

`value` zeigt den Antworttext, `code_id` die manuelle Bewertung. Die sechs Kennungen ordnen diese der ursprünglichen Antwort zu. Die Punktzahl stammt aus dem Kodierschema.

# Manuelle Codes einsetzen

```r
coded <- code_responses(responses, units, prepare = TRUE,
                        codes_manual = codes_manual)
count(coded, code_status, code_type)
saveRDS(coded, "Daten/Ergebnisse/03_coded.rds")
```

```
# A tibble: 4 × 5
  login_name variable_id code_id code_score code_status
  <chr>      <chr>          <int>      <dbl> <chr>
1 P001       TEXT               1        0.5 CODING_COMPLETE
2 P002       TEXT               2        1   CODING_COMPLETE
3 P003       TEXT               0        0   CODING_COMPLETE
4 P004       TEXT               1        0.5 CODING_COMPLETE
```

Die ursprünglichen Antworten werden erneut übergeben. Ohne manuelle Kodierung genügt `coded <- coded_auto`.

# Was complete_design() ergänzt

1. **Vorgesehene Fälle:** Das Testdesign legt fest, welche Person welche Variablen bearbeiten sollte. Fehlende Antwortzeilen werden ergänzt.
2. **Bearbeitungsreihenfolge:** Innerhalb jedes Testlets werden frühere Lücken und fehlende Antworten am Ende unterschieden. Eine Bearbeitung im nächsten Testlet verschiebt diese Grenze nicht.
3. **Itembezug:** Verknüpfte Itemvariablen und ihre Quellen bestimmen die Grenze; unabhängige Hilfsvariablen tun das nicht.
4. **Missing-Regeln:** Optionen und Missing-Schema bestimmen `code_type`, `code_id` und `code_score`. Der technische `code_status` bleibt erhalten.

Szenarien: Testlets · Itemgrenze und Hilfsvariablen

# complete_design(): Daten und Zuordnung

| Argument | Bedeutung |
|---|---|
| `coded` | Kodierte Antworten mit `prepare = TRUE` |
| `units` | Aufgabeninformationen einschließlich Kodierschema und Positionen innerhalb der Unit |
| `design` | Alle vorgesehenen Personen, Booklets und Variablen mit ihrer Reihenfolge |
| `identifiers` | Personenkennungen; Standard: `group_id`, `login_name`, `login_code` |
| `overwrite = FALSE` | Vorhandene `unit_codes` verwenden; `TRUE` bereitet sie erneut auf |
| `missings = NULL` | Optional: Tabelle mit `code_id`, `code_status`, `code_score`, `code_type`; legt Missing-Codes und Scores fest |

Die endgültige Missing-Art steht in **`code_type`**. Der technische `code_status` bleibt erhalten.

# complete_design(): Umgang mit fehlenden Antworten

| Argument | Wirkung bei TRUE |
|---|---|
| `recode_omissions_to_not_reached` | Abschließende Auslassungen innerhalb eines Testlets werden zu Not reached. Innere Auslassungen bleiben bestehen. |
| `preserve_not_reached` | Vorhandene oder beim Ergänzen erkannte Not-reached-Fälle bleiben erhalten, auch wenn später bearbeitete Variablen folgen. |

Beide Optionen sind standardmäßig **FALSE** und unabhängig voneinander.

Die Reihenfolge im Testdesign und die Positionen innerhalb der Unit bestimmen, ob eine fehlende Antwort vor oder nach späterer Bearbeitung liegt.

`complete_design()`: vollständige Hilfe

# Design ergänzen und Missings zuweisen

```r
design_coded <- complete_design(coded, units, design,
  recode_omissions_to_not_reached = FALSE, preserve_not_reached = FALSE)

count(design_coded, id_used, code_type)
saveRDS(design_coded, "Daten/Ergebnisse/04_design_coded.rds")
```

```
# A tibble: 3 × 5
  login_name unit_key variable_id code_type           code_score
  <chr>      <chr>    <chr>       <chr>                    <dbl>
1 P002       U2       Q1          MISSING_BY_OMISSION          0
2 P003       U5       Q1          MISSING_NOT_REACHED         NA
3 P003       U5       Q2          MISSING_NOT_REACHED         NA
```

P002/U2 fehlt vor späterer Bearbeitung: Auslassung. P003/U5 fehlt am Testletende: nicht erreicht.

# Missing-Fälle in derselben Beispielstudie

| Fall | Situation | Ergebnis mit Paketvoreinstellungen |
|---|---|---|
| **P002 · U2** | Antwortzeile fehlt, danach weitere Bearbeitung | Auslassung: −99 / 0 Punkte |
| **P003 · U5** | Antwortzeilen fehlen am Testletende | Not reached: −96 / NA |
| **P004 · U1/Q1** | Ungültige Antwort | Invalid response: −98 / 0 Punkte |
| **P005 · U4/TEXT** | Manuell als Auslassung kodiert, danach Bearbeitung | Auslassung: −99 / 0 Punkte |
| **P006 · U5** | Angezeigt, aber leer; danach keine Bearbeitung | Auslassung: −99 / 0 Punkte |

**Fehlende Zeile und angezeigte, leere Antwort sind unterschiedliche Ausgangsfälle.**

Szenarien: fehlende Zeilen · Ungültige Antworten

# Missing-Fälle im R-Objekt ansehen

```r
design_coded |>
  filter(code_id < 0, id_used) |>
  select(login_name, unit_key, variable_id, code_status, code_id, code_score) |>
  print()
```

```
# A tibble: 7 × 6
  login_name unit_key variable_id code_status code_id code_score
  <chr>      <chr>    <chr>       <chr>          <dbl>      <dbl>
1 P002       U2       Q1          <NA>             -99          0
2 P003       U5       Q1          <NA>             -96         NA
3 P003       U5       Q2          <NA>             -96         NA
4 P004       U1       Q1          INVALID          -98          0
5 P005       U4       TEXT        DISPLAYED        -99          0
6 P006       U5       Q1          DISPLAYED        -99          0
7 P006       U5       Q2          DISPLAYED        -99          0
```

`NA` in `code_status` kennzeichnet hier ergänzte Zeilen. Der zugewiesene Missing-Code steht in `code_id`.

# Missing-Art und Punktwert auseinanderhalten

| Art | Code | Standardscore nach `complete_design()` |
|---|---|---|
| Keine Kodierung (`NO_CODING`) | −93 | NA |
| Not reached | −96 | NA |
| Coding impossible | −97 | NA |
| Invalid response | −98 | 0 |
| By omission | −99 | 0 |
| By design | außerhalb des vorgesehenen Designs | NA beim Umformen |

Das sind Paketvoreinstellungen. Studienspezifische Regeln ausdrücklich festlegen.

# Status, Code-Art, Code-ID und Score unterscheiden

## Beispiel: P006 · U5/Q1 nach `complete_design()`

| Spalte | Aussage | Wert im Beispiel |
|---|---|---|
| `code_status` | Technischer Bearbeitungs- bzw. Kodierstatus | `DISPLAYED`: angezeigt |
| `code_type` | Bedeutung der Bewertung, einschließlich Missing-Art | `MISSING_BY_OMISSION`: ausgelassen |
| `code_id` | Kennnummer des Codes innerhalb der Variable | `-99` |
| `code_score` | Zugehörige Punktzahl für die Auswertung | `0` |

Mit der Regel „abschließende Auslassungen als nicht erreicht“ bleibt der Status **`DISPLAYED`**. Die übrigen Felder ändern sich zu **`MISSING_NOT_REACHED`**, **−96** und **`NA`**.

Der Status allein beschreibt damit weder die endgültige Missing-Art noch die Punktzahl. Die Code-ID ist ebenfalls keine Punktzahl.

# Abschließende Auslassungen umkodieren

```r
omissions_reclassified <- complete_design(coded, units, design,
  recode_omissions_to_not_reached = TRUE, preserve_not_reached = FALSE)

count(omissions_reclassified, id_used, code_type)
```

```
# A tibble: 4 × 5
  Regel       login_name unit_key code_type           code_score
  <chr>       <chr>      <chr>    <chr>                    <dbl>
1 Standard    P006       U5       MISSING_BY_OMISSION          0
2 Standard    P006       U5       MISSING_BY_OMISSION          0
3 Alternative P006       U5       MISSING_NOT_REACHED         NA
4 Alternative P006       U5       MISSING_NOT_REACHED         NA
```

Bei P006/U5 werden abschließende Auslassungen zu Not reached.

# Not reached vor späterer Bearbeitung erhalten

```r
not_reached_preserved <- complete_design(coded, units, design,
  recode_omissions_to_not_reached = FALSE, preserve_not_reached = TRUE)
```

```
# A tibble: 2 × 5
  Regel       login_name unit_key code_type           code_score
  <chr>       <chr>      <chr>    <chr>                    <dbl>
1 Standard    P002       U2       MISSING_BY_OMISSION          0
2 Alternative P002       U2       MISSING_NOT_REACHED         NA
```

Bei **P002/U2** erkennt die ergänzte Zeile zunächst Not reached. `preserve_not_reached = TRUE` erhält diese Einordnung trotz späterer Bearbeitung.

Die gewünschte Kombination beider Optionen im ursprünglichen `design_coded`-Aufruf einstellen.

# Personen für die Auswertung auswählen

```r
analysis_data <- filter(design_coded, id_used)
count(design_coded, id_used)
```

```
# A tibble: 4 × 5
  login_name unit_key variable_id id_used code_score
  <chr>      <chr>    <chr>       <lgl>        <dbl>
1 P001       U1       Q1          TRUE             0
2 P001       U1       Q2          TRUE             1
3 P001       U2       Q1          TRUE             0
4 P001       U3       Q1          TRUE             0
```

```
# A tibble: 1 × 1
  n_personen
       <int>
1         39
```

`id_used` bedeutet: mindestens ein gespeicherter Kodierstatus liegt vor. P040 ohne Antworten entfällt; es bleiben 39 Personen.

# Item-IDs: Unit-Kurzname und Itemnummer

| Fach / Schulstufe | Kompetenzbereich | Projekt / Pool | Unitnummer | Itemnummer |
|---|---|---|---|---|
| D | H | B | 001 | 01 |
| Deutsch Primar | Zuhören | BT | Unit 001 | Item 01 |

**DHB001** + **01** → **DHB00101**

Die ersten sechs Stellen gehören in den **Unit-Kurznamen**. Die fortlaufende zweistellige **Itemnummer** steht in den Item-Metadaten.

Im Workshop sind die Unitnamen vereinfacht: **U1** + **01** → **U101**. Die Itemnummer wird als Text gespeichert.

Item-ID-Konventionen und vollständige Wertelisten (IQB-intern)

# Variablen-IDs und Itemnummern unterscheiden

| Rolle | Benennung laut Leitfaden | Beispiel in der Studie |
|---|---|---|
| Direkt kodierter Itemwert | `01`, `02`, … | `Q1` → Item `01` |
| Basisvariablen für eine Ableitung | `01a`, `01b`, … | U3: `Q1`, `Q2` |
| Abgeleiteter Itemwert | `01` | U3: `TOTAL` → Item `01` |
| Prüfung auf einzigartige Eingaben | `01_unique` | Hilfsvariable, kein zusätzliches Item |
| Medien ohne Itemwertung | z. B. `_audio01` | Fortschritt statt Itemscore |

Die **Metadaten-Verknüpfung** bestimmt, welche Variable den Itemwert liefert. Die freie Item-ID `RECHNEN_01` aus Beispiel 1 wird hier durch lokale Itemnummern `01`, `02`, … abgelöst.

Leitfaden: Elemente und Variablen – Benennungskonventionen dort noch als **nicht final abgestimmt** gekennzeichnet.

# Item-Verknüpfungen ergänzen

Variablen ohne Item-ID ansehen und nur verknüpfte Items in den Skalierungsdatensatz aufnehmen.

```r
design_items <- add_item_id(design_coded, units)
item_data <- add_item_id(analysis_data, units)

unmapped <- item_data |>
  filter(is.na(item_id) | trimws(item_id) == "") |>
  distinct(unit_key, variable_id)
print(unmapped) # sollte U3/Q1 und U3/Q2 zeigen

item_data <- filter(item_data, !is.na(item_id), trimws(item_id) != "")
item_data <- mutate(item_data, item = paste0(unit_key, item_id))
```

`paste0(unit_key, item_id)` verbindet Unit und Itemnummer direkt: `U3` + `01` → `U301`. Führende Nullen bleiben erhalten.

Zum Skript: 03_datenaufbereitung.R

# Aus Variablen werden Auswertungsitems

```r
item_data |>
  filter(login_name == "P001", unit_key %in% c("U3", "U4")) |>
  select(unit_key, variable_id, item_id, item, code_score) |>
  print()
```

```
# A tibble: 2 × 5
  unit_key variable_id item_id item  code_score
  <chr>    <chr>       <chr>   <chr>      <dbl>
1 U3       TOTAL       01      U301         0
2 U4       TEXT        01      U401         0.5
```

U3/TOTAL wird ein Item; die Basisvariablen Q1 und Q2 erhalten keine zusätzlichen Spalten im Skalierungsdatensatz.

# Offene Itemkodierungen erkennen

```r
unresolved <- item_data |>
  filter(code_status %in% c("UNSET", "CODING_INCOMPLETE", "DERIVE_PENDING",
                            "CODING_ERROR", "DERIVE_ERROR",
                            "INTENDED_INCOMPLETE"))
```

```
# A tibble: 0 × 4
# ℹ 4 variables: login_name <chr>, item <chr>, code_status <chr>, code_type <chr>
```

Hier ist die Tabelle leer. Der Filter wählt ausschließlich die sechs genannten Werte von `code_status` aus; `code_type` ist kein Auswahlkriterium.

# Missing-Anteile je Person

```r
missing_summary <- summarise_missings(design_items)
missing_summary_used <- summarise_missings(design_items, used_only = TRUE)

select(missing_summary, all_of(person_columns), id_used, n_items,
       n_not_reached, n_by_omission, prop_missing)
```

```
# A tibble: 4 × 5
  login_name n_items n_not_reached n_by_omission prop_missing
  <chr>        <int>         <int>         <int>        <dbl>
1 P001             7             0             0        0
2 P002             7             0             1        0.143
3 P003             7             2             0        0.286
4 P004             7             0             0        0.143
```

`n_items` zählt vorgesehene Items. Nicht vorgesehene Aufgaben gehen nicht als fehlende Antworten in den Nenner ein.

# Antwortkategorie, Code und Score

## MC-Item U101 aus der synthetischen Lesestudie

| Antwortkategorie | Code | Score |
|---|---|---|
| A · richtige Auswahl | 1 | 1 |
| B · Distraktor | 0 | 0 |
| C · Distraktor | 0 | 0 |
| Ungültige Antwort | −98 | 0 |

**Kategorien** unterscheiden die gewählten Antworten.

**Codes** fassen die Bewertung zusammen.

**Scores** geben die Punktzahl an.

# Psychometrische Kennwerte berechnen

```r
domains <- distinct(units, unit_key)
domains$domain <- "Lesen"

psychometrics <- evaluate_psychometrics(analysis_data, units, domains = domains)
psychometrics <- add_item_id(psychometrics, units)

psychometrics <- psychometrics |>
  mutate(item = if_else(is.na(item_id), NA_character_, paste0(unit_key, item_id))) |>
  relocate(unit_key, variable_id, item, item_id)
```

```
# A tibble: 4 × 5
  item  code_id code_n category_id category_n
  <chr>   <dbl>  <int> <chr>            <int>
1 U101      -98      1 <NA>                NA
2 U101        0     17 B                    5
3 U101        0     17 C                   12
4 U101        1     21 A                   21
```

Vollständige Ausgabe: **37 Zeilen, 21 Spalten**. Codes wiederholen sich je Antwortkategorie.

# Kennungen und Herkunft der Variablen

```r
variable_info <- psychometrics |>
  filter(unit_key == "U3") |>
  distinct(unit_key, variable_id, item, item_id, variable_source_type)
print(variable_info) # Q1/Q2: BASE; TOTAL: SUM_SCORE
```

```
# A tibble: 3 × 5
  unit_key variable_id item  item_id variable_source_type
  <chr>    <chr>       <chr> <chr>   <chr>
1 U3       Q1          <NA>  <NA>    BASE
2 U3       Q2          <NA>  <NA>    BASE
3 U3       TOTAL       U301  01      SUM_SCORE
```

`unit_key` und `variable_id` identifizieren die Variable. `item_id` ist die Itemnummer aus den Metadaten, `item` die zusammengesetzte Kennung.

**`BASE`:** direkte Antwortvariable. **`SUM_SCORE`:** aus den Scores anderer Variablen abgeleitet. Q1 und Q2 bleiben in der Kennwertetabelle, auch ohne eigene Item-ID.

# Code-ID, Code-Art und Punktzahl

```r
code_info <- psychometrics |>
  filter(item == "U301") |>
  distinct(code_id, code_type, code_score)
print(code_info) # Code 1: Teilpunkte; Code 2: volle Punktzahl
```

```
# A tibble: 3 × 3
  code_id code_type      code_score
    <dbl> <chr>               <dbl>
1       0 RESIDUAL_AUTO         0
2       1 PARTIAL_CREDIT        0.5
3       2 FULL_CREDIT           1
```

`code_id` bezeichnet den Code; `code_type` beschreibt seine Bedeutung. `code_score` ist die zugehörige Punktzahl.

Bei U301 bedeutet Code 1 **Teilpunkte** (`PARTIAL_CREDIT`), Code 2 **volle Punktzahl** (`FULL_CREDIT`). `RESIDUAL_AUTO` ist hier der automatische Restcode mit 0 Punkten; Missing-Arten stehen ebenfalls in `code_type`.

# Antwortkategorien: Welche Auswahl tritt wie oft auf?

```r
category_stats <- psychometrics |>
  filter(unit_key == "U1", item_id == "01", !is.na(category_id)) |>
  select(category_id, code_id, category_n, category_p_valid, category_pbc)
print(category_stats) # sollte A: 21, B: 5 und C: 12 Antworten zeigen
```

```
# A tibble: 3 × 5
  category_id code_id category_n category_p_valid category_pbc
  <chr>         <dbl>      <int>            <dbl>        <dbl>
1 B                 0          5            0.128      0.00492
2 C                 0         12            0.308     -0.338
3 A                 1         21            0.538      0.312
```

B wurde fünfmal gewählt, C zwölfmal. Beide erhalten Code 0; ihre Häufigkeiten und Zusammenhänge mit dem Bereichsscore bleiben dennoch unterscheidbar.

# Codes: Welche Bewertungen treten wie oft auf?

```r
code_stats_mc <- psychometrics |>
  filter(unit_key == "U1", item_id == "01") |>
  select(code_id, code_score, code_n, code_p_valid, code_pbc) |>
  distinct()
print(code_stats_mc) # sollte für Code 0 insgesamt 17 Antworten zeigen
```

```
# A tibble: 3 × 5
  code_id code_score code_n code_p_valid code_pbc
    <dbl>      <dbl>  <int>        <dbl>    <dbl>
1     -98          0      1       0.0256 -0.00871
2       0          0     17       0.436  -0.308
3       1          1     21       0.538   0.309
```

Code 0 umfasst B und C: **5 + 12 = 17 Antworten**. Die ungültige Antwort bleibt als Code −98 erkennbar. `distinct()` entfernt wiederholte Codezeilen aus der gemeinsamen Kategorie- und Codeausgabe.

# Fallzahlen: insgesamt und mit gültigem Score

```r
denominator_stats <- psychometrics |>
  filter(item == "U501") |>
  distinct(code_id, code_score, code_n, code_n_total, code_n_valid)
print(denominator_stats) # insgesamt 20 Fälle, davon 19 mit gültigem Score
```

```
# A tibble: 4 × 5
  code_id code_score code_n code_n_total code_n_valid
    <dbl>      <dbl>  <int>        <int>        <int>
1     -99          0      1           20           19
2     -96         NA      1           20           19
3       0          0     12           20           19
4       1          1      6           20           19
```

**U501:** `code_n` zählt den jeweiligen Code. `code_n_total` zählt alle **20 vorgesehenen Fälle** unter den ausgewählten Personen; `code_n_valid` die **19 Fälle mit Score ungleich `NA`**.

„Valid“ bedeutet hier **vorhandener Score**. Die Auslassung mit Score 0 zählt mit, Not reached mit `NA` nicht.

# Antwortkategorien: Bezeichnungen und Anteile

```r
category_details <- psychometrics |>
  filter(item == "U501", !is.na(category_id)) |>
  select(category_id, category_label, category_n,
         category_p_total, category_p_valid)
print(category_details)
```

```
# A tibble: 3 × 5
  category_id category_label category_n category_p_total category_p_valid
  <chr>       <chr>                <int>            <dbl>            <dbl>
1 B           Distraktor B             7             0.35            0.368
2 C           Distraktor C             5             0.25            0.263
3 A           Richtig                  6             0.3             0.316
```

`category_id` ist der Antwortwert, `category_label` seine Bezeichnung. `category_n` zählt diese Kategorie.

Auch hier gelten die Nenner **20 insgesamt** und **19 mit gültigem Score**: Distraktor B wurde siebenmal gewählt → `category_p_total` = **35 %**, `category_p_valid` ≈ **36,8 %**.

Bei Variablen ohne solche Antwortkategorien können die `category_*`-Spalten `NA` enthalten.

# PBC: punktbiseriale Korrelation

```r
discrimination_stats <- psychometrics |>
  filter(item == "U101", !is.na(category_id)) |>
  select(category_id, code_id, domain, category_pbc, code_pbc)
print(discrimination_stats)
```

```
# A tibble: 3 × 5
  category_id code_id domain category_pbc code_pbc
  <chr>          <dbl> <chr>         <dbl>    <dbl>
1 B                  0 Lesen       0.00492   -0.308
2 C                  0 Lesen      -0.338     -0.308
3 A                  1 Lesen       0.312      0.309
```

`code_pbc` und `category_pbc` korrelieren das **Auftreten** eines Codes bzw. einer Kategorie mit dem mittleren Itemscore der Person im Bereich `domain`.

# PBC richtig einordnen

- **Vergleichsscore:** mittlerer Score der verknüpften Items im Bereich. Fehlende Scores werden ausgelassen.
- **Positives Vorzeichen:** Der Code bzw. die Kategorie tritt häufiger bei höheren Bereichsscores auf; bei negativem Vorzeichen häufiger bei niedrigeren.
- **Keine Part-whole-Korrektur:** Der eigene Itemscore bleibt im Bereichsmittelwert enthalten.
- **Unterschiedliche Fallbasis:** Codes berücksichtigen auch Missing-Codes; Kategorien nur Antworten mit `code_id >= 0`.

Bei U101: **Code 1 ≈ 0,309** auf Basis von 39 Fällen; **Kategorie A ≈ 0,312** auf Basis von 38 Fällen.

`evaluate_psychometrics()`: Funktionsreferenz

# Scores: Welche Punktzahlen gehen in die Skalierung?

```r
score_stats <- item_data |>
  filter(unit_key == "U1", item_id == "01") |>
  count(code_score, name = "score_n") |>
  mutate(score_p = score_n / sum(score_n))
print(score_stats) # sollte 18 Personen mit 0 und 21 mit 1 Punkt zeigen
```

```
# A tibble: 2 × 3
  code_score score_n score_p
       <dbl>   <int>   <dbl>
1          0      18   0.462
2          1      21   0.538
```

**Score 0: 17 falsche + 1 ungültige Antwort = 18 Personen.**

`evaluate_psychometrics()` liefert Kategorie- und Codekennwerte. Die Scorehäufigkeiten werden hier direkt aus den Person-Item-Zeilen berechnet.

# Den Skalierungsdatensatz erstellen

```r
scaling_data <- item_data |>
  select(group_id, login_name, login_code, item, code_score) |>
  pivot_wider(names_from = item, values_from = code_score) |>
  arrange(group_id, login_name, login_code)
```

```
# A tibble: 4 × 4
  login_name  U101  U301  U501
  <chr>      <dbl> <dbl> <dbl>
1 P001           0     0     0
2 P002           0     1     1
3 P003           0     0    NA
4 P004           0     0     0
```

Eine Zeile ist eine Person; jede Itemspalte enthält Punkte. **P001/U601 ist NA, weil U6 nicht zu Booklet B1 gehört** (Missing by Design). P003/U501 ist dagegen nicht erreicht. Die Long-Daten bewahren den Unterschied.

# Ergebnisse aus einer durchgehenden Aufbereitung

Alle Ausgaben liegen gemeinsam unter `Daten/Ergebnisse/`.

| RDS-Datei | Zustand derselben Beispielstudie |
|---|---|
| `01_responses.rds` | Eingelesene Antworten |
| `02_coded_auto.rds` | Automatisch kodierte Antworten |
| `03_coded.rds` | Manuelle Codes ergänzt |
| `04_design_coded.rds` | Design ergänzt und Missings zugewiesen |
| `05_scaling_data.rds` | Ausgewählte Personen und Items im breiten Format |

```r
saveRDS(analysis_data, "Daten/Ergebnisse/analysis_long.rds")
saveRDS(design_items, "Daten/Ergebnisse/design_items.rds")
saveRDS(scaling_data, "Daten/Ergebnisse/05_scaling_data.rds")
```

`analysis_long.rds` und `design_items.rds` bewahren zusätzlich Codes, Status und Missing-Arten.

# Pause und Ausprobieren

## Ca. 10 Minuten

03_datenaufbereitung.R von oben nach unten ausführen, sofern noch nicht parallel geschehen.

Die Ausgaben zu Kodierung, Missings und Kennwerten ansehen. Den fertigen Skalierungsdatensatz unter **`Daten/Ergebnisse/scaling_data.csv`** öffnen.

Offene Fragen und Fehlermeldungen für die anschließende Fragerunde festhalten.

# Fragen?

# 3 · Logs und Codebücher

**Logdaten:** Ereignisse überblicken, Bearbeitungszeiten und Fokusverluste auswerten.

**Codebücher:** Kodierschemata dokumentieren und als Excel, CSV oder DOCX exportieren.

04_logs.R · 05_kodiermanuale.R · 06_kodiermanuale_studio.R

# Logdaten einlesen und einzelne Ereignisse ansehen

```r
logs <- read_logs("Daten/Logs.csv")
# Alternativ nach API-Abruf: logs <- readRDS("Daten/EigeneDaten/logs.rds")

# Ältere Exporte haben keinen getrennten Unit-Alias.
if (!"unit_alias" %in% names(logs)) logs$unit_alias <- logs$unit_key

count(logs, group_id, login_name, booklet_id) # P001: 9, P002: 6 Ereignisse
```

```
# A tibble: 4 × 4
  login_name unit_key ts            log_entry
  <chr>      <chr>    <chr>         <chr>
1 P001       U1       1780000000100 "PLAYER = LOADING"
2 P001       U1       1780000000450 "PLAYER = RUNNING"
3 P001       U1       1780000004000 "FOCUS : \"HAS_NOT\""
4 P001       U1       1780000006500 "FOCUS : \"HAS\""
```

`ts` enthält den Zeitstempel, `log_entry` das protokollierte Ereignis.

# Welche Ereignistypen sind vorhanden?

```r
inventory <- summarise_log_inventory(logs)
```

```
# A tibble: 5 × 4
  log_type            n known_log_type supported_parser
  <chr>           <int> <lgl>          <lgl>
1 FOCUS               3 TRUE           TRUE
2 PLAYER              3 TRUE           TRUE
3 CURRENT_PAGE_NR     2 TRUE           TRUE
4 PAGE_COUNT          2 TRUE           TRUE
5 CONNECTION          1 TRUE           TRUE
```

`known_log_type` kennzeichnet bekannte Ereignistypen; `supported_parser` zeigt, ob ein Parser für ihre Aufbereitung verfügbar ist.

Hilfe: summarise_log_inventory()

# Auffälligkeiten vor der Zeitinterpretation prüfen

```r
anomalies <- detect_log_anomalies(logs)
# QC = Quality Control: Qualitätsprüfung, hier zusammengefasst je Session.
qc <- summarise_log_qc(logs, anomalies = anomalies)

anomalies |>
  select(login_name, booklet_id, anomaly_code, severity, evidence)
```

```
# A tibble: 7 × 3
  login_name anomaly_code                severity
  <chr>      <chr>                       <ord>
1 P002       loading_without_running     warning
2 P002       connection_lost             warning
3 P002       last_connection_lost        critical
4 P002       focus_lost_never_regained   warning
5 P002       page_nr_exceeds_page_count  warning
6 P002       runtime_error               critical
7 P002       missing_loadcomplete        warning
```

`anomaly_code` bezeichnet die Auffälligkeit, `severity` ihre Dringlichkeit. `evidence` enthält den zugehörigen Beleg aus den Logs.

# Qualitätsprüfung pro Session

**QC = Quality Control**, hier die Zusammenfassung der Hinweise je Session.

```
qc |>
  select(login_name, booklet_id, log_qc_flag, n_anomalies) |>
  print() # P001: ok; P002: critical
```

```
# A tibble: 2 × 4
  login_name booklet_id log_qc_flag n_anomalies
  <chr>      <chr>      <chr>              <int>
1 P001       B1         ok                     0
2 P002       B1         critical               7
```

P002 hat **sieben Hinweise**: unter anderem Laden ohne erfolgreichen Start (`loading_without_running`), Playerfehler (`runtime_error`) und zuletzt verlorene Verbindung (`last_connection_lost`).

`log_qc_flag` richtet sich nach dem schwersten Hinweis: `critical` vor `warning` vor `info`; ohne Hinweise steht `ok`. Mehrere Hinweise können dasselbe Problem beschreiben. Ein Flag begründet eine Prüfung, keinen automatischen Ausschluss.

# Meldungen bei estimate_unit_times() einordnen

| Konsolenausgabe | Bedeutung |
|---|---|
| **„Design tibble with block info not provided …“** | Ohne `full_design` lassen sich automatische Blockwechsel nicht erkennen. Jedes neue Unit-Laden nach Fokusverlust gilt dann als Fokus-Rückkehr. |
| **„Berechne Unit-Bearbeitungs- und Ladezeiten“** | Die Berechnung der Unit-Zeiten läuft. |
| **„Berechne Focus-Events“** | Fokusverluste und Rückkehrereignisse werden zugeordnet. |
| **„Keine Seiten-IDs …“** | Ohne `CURRENT_PAGE_ID` entstehen keine Seitenzeiten. Unit-Zeiten sind weiterhin möglich; `CURRENT_PAGE_NR` und `PAGE_COUNT` reichen für Seitenzeiten nicht. |

Bei **automatischen Blockwechseln** das passende `full_design` mit `testlet_no` und `block_self_switch = FALSE` übergeben.

`include_environment = FALSE` lässt nur Browser- und Geräteangaben weg. Die Meldungen sind Hinweise zum Ablauf, keine Abbruchfehler.

# Beobachtete Unit- und Ladezeiten in Sekunden

```r
unit_times <- estimate_unit_times(logs, include_environment = FALSE)

# Millisekunden in Sekunden umrechnen:
times <- unit_times |>
  transmute(group_id, login_name, login_code, booklet_id, unit_key,
            unit_n_play, unit_time_s = unit_time / 1000,
            unit_loadtime_s = unit_loadtime / 1000,
            n_run_no_load, n_failed_loadings)
```

```
# A tibble: 1 × 4
  login_name unit_key unit_time_s unit_loadtime_s
  <chr>      <chr>          <dbl>           <dbl>
1 P001       U1              11.6            0.35
```

Die Zeit reicht von `RUNNING` bis zum nächsten Unit-Start oder beobachteten Ende. Ohne `RUNNING` entsteht keine Zeitzeile. Fokusverluste sind enthalten.

# Abgeschlossene und offene Fokusverluste unterscheiden

```r
focus <- summarise_log_focus(logs)
focus_summary <- focus |>
  transmute(group_id, login_name, login_code, booklet_id,
            n_focus_lost, n_focus_regained,
            focus_lost_s = total_focus_lost_time / 1000,
            max_focus_lost_s = max_focus_lost_time / 1000,
            has_unresolved_focus_loss)
```

```
# A tibble: 2 × 4
  login_name n_focus_lost focus_lost_s has_unresolved_focus_loss
  <chr>             <int>        <dbl> <lgl>
1 P001                  1          2.5 FALSE
2 P002                  1          0   TRUE
```

`summarise_log_focus()` verwendet nur die protokollierten `FOCUS`-Ereignisse. Nur `HAS_NOT` → `HAS` ergibt eine abgeschlossene Dauer; Unit-Laden zählt hier nicht als Rückkehr. Bei P002 bleibt der Fokusverlust offen: 0 Sekunden bedeutet hier keine gemessene Dauer.

# Sessionübersicht erstellen

```r
session_keys <- c("group_id", "login_name", "login_code", "booklet_id")
session_overview <- qc |>
  select(all_of(session_keys), log_qc_flag) |>
  left_join(focus_summary, by = session_keys)
```

```
# A tibble: 2 × 4
  login_name log_qc_flag n_focus_lost focus_lost_s
  <chr>      <chr>               <int>        <dbl>
1 P001       ok                      1          2.5
2 P002       critical                1          0
```

Fokusdauer gilt pro Session, Bearbeitungszeit pro Unit. Die Session-Fokusdauer lässt sich daher nicht von jeder Unit-Zeit abziehen.

# Logergebnisse speichern

```r
saveRDS(list(inventory = inventory, anomalies = anomalies, qc = qc,
             unit_times = unit_times, focus = focus), "Daten/Ergebnisse/log_qc.rds")
```

Das Skript speichert außerdem Ereignisinventar, Auffälligkeiten und Sessionübersicht als CSV. Die RDS-Datei bewahrt die Tabellen für die weitere Arbeit in R.

# Codebücher und Kodiermanuale

**Kodierschemata dokumentieren und für die Auswertung oder Weitergabe aufbereiten.**

Ein **Codebuch** macht Variablen, Codes, Beschreibungen und Scores als Tabelle lesbar. Ein **Kodiermanual** ergänzt Anweisungen für die Bewertung von Antworten.

1. **Eigene Studie:** Codebuch aus den Kodierschemata in IQB-Studio erstellen; als Excel oder DOCX bereitstellen.
2. **Ohne Zugang üben:** Eine vorbereitete synthetische Tabelle verwenden, Units auswählen und exportieren.

# Mit eigenem Zugang: IQB-Studio vorbereiten

```r
studio_version <- "21.0.0" # Version der eigenen Anmeldeseite übernehmen
studio_workspace_id <- 1300 # Beispiel: eigene Workspace-ID einsetzen
selected_units <- c("UNIT-1", "UNIT-2") # Exakte Unit-Kurznamen; NULL für alle
missing_profile <- "IQB-Standard" # Eigenen Profilnamen prüfen; NULL ohne Profil
export_path <- "Daten/Ergebnisse/codebooks_studio"
dir.create(export_path, recursive = TRUE, showWarnings = FALSE)
```

```r
login <- login_studio(app_version = studio_version, keyring = TRUE)
workspace <- access_workspace(login, ws_id = studio_workspace_id)
```

Vorlage für eigene Daten: Version, Workspace und Unit-Auswahl anpassen. Die Anmeldung verwendet www.iqb-studio.de und den lokalen Passwortspeicher.

# Codebuch aus IQB-Studio abrufen

```r
cb_studio <- prepare_codebook(workspace, unit_keys = selected_units,
                              missings_profile = missing_profile,
                              only_coded = TRUE, show_score = TRUE,
                              derived = TRUE, manual = TRUE, closed = TRUE)
names(cb_studio)
head(cb_studio)
writexl::write_xlsx(cb_studio, file.path(export_path, "codebook.xlsx"))
```

`only_coded` wählt Variablen mit Codes. `derived`, `manual` und `closed` nehmen abgeleitete, manuell kodierte und geschlossen kodierbare Variablen auf.

`show_score = TRUE` fordert Punktzahlen an. Ab Version .9043 sind vorhandene Werte in `score` numerisch; fehlende Angaben bleiben `NA`. Die vereinfachte Lehrtabelle verwendet dafür später `code_score`.

Hilfe: prepare_codebook()

# DOCX-Kodiermanual direkt aus IQB-Studio herunterladen

```
download_codebook(workspace, path = export_path, unit_keys = selected_units,
                  format = "docx", general_instructions = TRUE,
                  missings_profile = missing_profile, show_score = TRUE)
```

`general_instructions = TRUE` übernimmt allgemeine Kodieranweisungen. `missings_profile` ergänzt das in IQB-Studio ausgewählte Missing-Profil.

Die DOCX-Datei wird unter `export_path` gespeichert. Ihr Name stammt aus dem Workspace-Namen. Der Download benötigt den eigenen IQB-Studio-Zugang.

Hilfe: download_codebook()

# Ohne Zugang: vorbereitete Codebuchtabelle laden

```r
cb <- readr::read_csv("Daten/codebook.csv", show_col_types = FALSE,
                      col_types = readr::cols(code_id = readr::col_character()))
count(cb, unit_key, variable_id) # 10 Variablen aus 6 Units
```

```r
cb |>
  select(unit_key, variable_id, code_id, code_label, code_score) |>
  head(4) |> print()
```

```
# A tibble: 4 × 5
  unit_key variable_id code_id code_label          code_score
  <chr>    <chr>       <chr>   <chr>                    <dbl>
1 U1       Q1          1       Richtige Auswahl A           1
2 U1       Q1          0       Andere Auswahl               0
3 U1       Q2          1       Richtige Auswahl A           1
4 U1       Q2          0       Andere Auswahl               0
```

Die synthetische Datei ersetzt hier den Abruf aus IQB-Studio. Das Einlesen ist nur der Einstieg in die lokale Übung.

# Units auswählen und Code von Score unterscheiden

```r
selected_cb <- cb |>
  filter(unit_key %in% c("U3", "U4")) |>
  select(unit_key, variable_id, code_id, code_label, code_score)
```

```r
selected_cb |>
  filter(unit_key == "U4", variable_id == "TEXT") |>
  print() # Code 2: Score 1; Code 1: Score 0.5
```

```
# A tibble: 3 × 5
  unit_key variable_id code_id code_label                          code_score
  <chr>    <chr>       <chr>   <chr>                                    <dbl>
1 U4       TEXT        2       Begruendung nennt den Textbeleg            1
2 U4       TEXT        1       Passende Aussage ohne Textbeleg            0.5
3 U4       TEXT        0       Unzutreffende Begruendung                  0
```

Bei U4/TEXT bedeutet Code 2 einen Punkt, Code 1 einen halben Punkt. Die Code-ID ist keine Punktzahl.

# Codebuch als Excel speichern

```
writexl::write_xlsx(list(Codebuch = cb, Auswahl = selected_cb),
                    "Daten/Ergebnisse/codebook.xlsx")
```

Excel enthält die Blätter `Codebuch` und `Auswahl`. Das Skript speichert die Auswahl außerdem als CSV.

# Pause und Ausprobieren

## Ca. 10 Minuten

04_logs.R und 05_kodiermanuale.R ausführen, sofern noch nicht parallel geschehen.

Zeiten und Fokusverluste ansehen und das Excel-Codebuch unter `Daten/Ergebnisse/codebook.xlsx` öffnen. 06 ist die Vorlage für eigene IQB-Studio-Zugänge.

Offene Fragen für die anschließende Runde festhalten.

# Fragen?

# Auf eigene Daten übertragen

1. **Daten beziehen:** In `01_datenabruf.R` Zugänge und Workspace eintragen oder vorhandene Exporte einlesen.
2. **Aufbereitung anpassen:** In `03_datenaufbereitung.R` Pfade, manuelle Codes, Missing-Regeln und Bereichszuordnung festlegen.
3. **Ergebnisse sichern:** Skalierungsdatensatz, Long-Daten und verwendete Syntax gemeinsam aufbewahren.

Workshop-Repository · Handout · Nachschlagen

Downloadpaket mit R-Skripten und synthetischen Daten

# Quellen und Paketstand

- eatPrepTBA **0.9.8.9043** (`711f4f1`), eatAutoCode **0.5.0.9008**; geprüft am 25.10.2026.
- Standardworkflow und API-Einführung.
- Logdaten und Codebücher.
- Codegestaltung: eatPrep-Workshop 2024.
- Partial Credit in IQB-Studio: Original-Screenshots, dazu passendes synthetisches Rechenbeispiel.
- Original-Paketlogo; IQB-Styleguide und Präsentationsvorlage.

Lehrdaten synthetisch; Quellen und technische Prüfung im Workshop-Repository.

# Vielen Dank für eure Aufmerksamkeit!

Gibt es Fragen?