



Programmierung und Deskriptive Statistik

BSc Psychologie WiSe 2024/25

Belinda Fleischmann

TODO

(2) R und VSCode Grundlagen

Getting started

Arithmetik, Logik und Präzedenz

Variablen

Datenstrukturen

Programmierübungen und Selbstkontrollfragen

Getting started

Arithmetik, Logik und Präzedenz

Variablen

Datenstrukturen

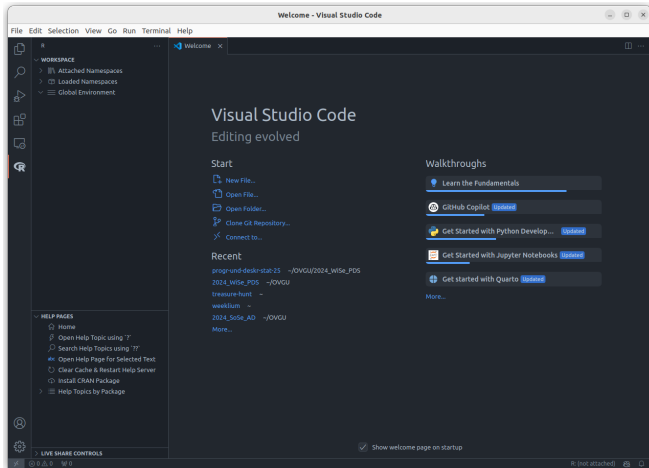
Programmierübungen und Selbstkontrollfragen

Installationen

- Die Programmiersprache [R](#)
- Die IDE [VSCode](#)
- VSCode für R startklar machen (Anleitung [hier](#))
- Optional: [radian](#)

Getting started

Mit VSCode und der R Extension vertraut machen

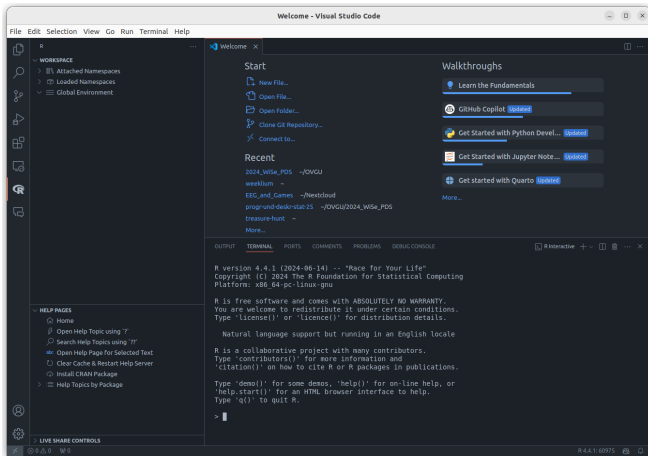


Öffnet eine R Console in VS Code

- GUI: *Terminal* → *New Terminal* →
 - In die Console hinter das "\$"-Zeichen "R" eingeben
- Keyboard Shortcut: Ctrl + Shift + P
 - In das sich öffnende Textfeld "R: Create R Terminal eingeben"

Getting started

R Console in VS Code



Die Basics

- Eingabe von R Befehlen bei >
- Autocomplete mit `Tab`
- Code ausführen mit `Enter`
- Vorherige Befehle mit Cursor `↑`
- Bereinigen des Konsolenoutputs mit `Ctrl` + `L`
- Code Ausführungsstopp mit `Esc` oder `Ctrl` + `C`

Beispiel für einen Befehl

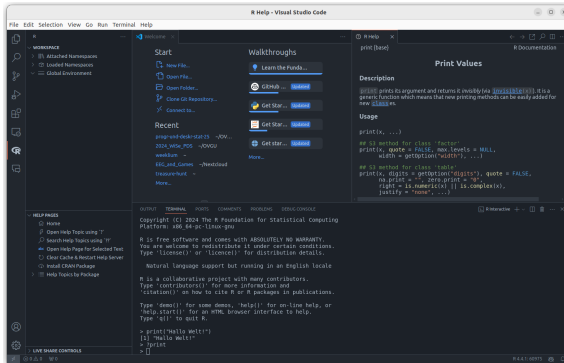
```
print("Hallo Welt!")
```

```
[1] "Hallo Welt!"
```

Anmerkung: **Code-Snippets dieser Form immer aktiv in der Konsole nachvollziehen!**

VSCode Interactive Viewers

Help Viewer



Mit dem Befehl `?funktionsname` oder über **HELP PAGES** links in der R Extension öffnen.

Beispiele

```
?print  
?sqrt
```

VS Code Wiki - Interactive viewers

Neue .R Datei erstellen

- GUI: *File* → *New File* → *R Document (r)*
- Keyboard Shortcut: Ctrl + N

Bestehende .R Datei öffnen

- GUI: *File* → *Open File*
- Keyboard Shortcut: Ctrl + O

Code in einem R Skript schreiben (Befehle in Programmiersprache formulieren)

```
print("Hallo Welt!") # Hinter Hashtags stehen dokumentierende Kommentare  
print("Hallo R!")    # Kommentare werden nicht ausgeführt
```

Anmerkung: **Code-Snippets dieser Form immer aktiv in einem R Skript dokumentieren!**

Befehle eines R Skripts ausführen

- Befehle an Console schicken:
 - Einzelnen Zeile, auf welcher der Cursor ruht: `Ctrl` + `Enter`
 - Ausführen aller Zeilen im Skript: `Ctrl` + `Shift` + `Enter`
- Ausführen des gesamten R Skripts ("Source"):
 - GUI: ▷ - Symbol / Run Source
 - Keyboard Shortcut: `Ctrl` + `Shift` + `S`
 - in R Console:

```
source("Pfad/zu/eurem/Skript.R")
```

- in bash Console:

```
Rscript Pfad/zu/eurem/Skript.R
```

Getting started

Arithmetik, Logik und Präzedenz

Variablen

Datenstrukturen

Programmierübungen und Selbstkontrollfragen

R Konsole als Taschenrechner

```
1 + 1
```

```
[1] 2
```

```
2 * 3
```

```
[1] 6
```

```
sqrt(4)
```

```
[1] 2
```

```
exp(0)
```

```
[1] 1
```

```
log(1)
```

```
[1] 0
```

Anmerkungen:

- `[1]` zeigt das erste und einzige Element des Ausgabevektors an
- Vektoren werden noch im Detail behandelt.

Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
^ oder **	Potenz
%*%	Matrixmultiplikation
%/%	Ganzzahlige Teilung ($5\%/%2 = 2$)
%%	Modulo ($5\% \% 2 = 1$)

- Matrixmultiplikation, Modulo, ganzzahlige Teilung benötigen wir zunächst nicht.
- Ganzzahlige Teilung gibt das Resultat der ganzzahligen Teilung an.
- Modulo gibt den ganzzahligen Rest bei ganzzahliger Teilung an.

- Die Boolesche Algebra und R kennen zwei *logische Werte*: TRUE und FALSE
- Bei Auswertung von Relationsoperatoren ergeben sich logische Werte

Relationsoperator	Bedeutung
==	Gleich
!=	Ungleich
<, >	Kleiner, Größer
<=, >=	Kleiner gleich, Größer gleich
	ODER
&	UND

- <, <=, >, >= werden zumeist auf numerische Werte angewendet.
- ==, != werden zumeist auf beliebige Datenstrukturen angewendet.
- | und & werden zumeist auf logische Werte angewendet.
- | implementiert das inklusive *oder*. Die Funktion xor() implementiert das exklusive ODER.

Aufruf	Bedeutung
<code>abs(x)</code>	Betrag
<code>sqrt(x)</code>	Wurzel
<code>ceiling(x)</code>	Aufrunden ($\text{ceiling}(2.7) = 3$)
<code>floor(x)</code>	Abrunden ($\text{floor}(2.7) = 2$)
<code>round(x)</code>	Mathematisches Runden ($\text{round}(2.5) = 2$)
<code>exp(x)</code>	Exponentialfunktion
<code>log(x)</code>	Logarithmus Funktion

- Hierbei handelt es sich um eine Auswahl. Einen vollständigen Überblick gibt

```
names(methods:::.BasicFunsList)
```

- R unterscheidet formal nicht zwischen Operatoren und Funktionen
- Operatoren können mit der Infix Notation als Funktionen genutzt werden

```
`+`(2,3)           # Infixnotation für 2 + 3
```

Operatorpräzedenz (Operatorrangfolge)

- Regeln der Form “Punktrechnung geht vor Strichrechnung”.
- Vordefinierte Operatorpräzedenz kann durch Klammern überschrieben werden.

```
2 * 3 + 4
```

```
[1] 10
```

```
2 * (3 + 4)
```

```
[1] 14
```

- Generelle Empfehlung:
 - Operatorrangfolge nicht raten oder folgern, sondern nachschlagen.
 - Lieber Klammern setzen, als keine Klammern setzen.
 - Immer nachschauen, ob Berechnungen die erwarteten Ergebnisse liefern.

Präzedenz und Syntax nachschlagen

?Syntax # öffnet R Help Fenster

Operator Syntax and Precedence

Description

Outlines R syntax and gives the precedence of operators.

Details

The following unary and binary operators are defined. They are listed in precedence groups, from highest to lowest.

:: :::	access variables in a namespace
\$ @	component / slot extraction
[[[indexing
^	exponentiation (right to left)
- +	unary minus and plus
:	sequence operator
%any% >	special operators (including %% and %/%)
* /	multiply, divide
+ -	(binary) add, subtract
< > <= >= == !=	ordering and comparison
!	negation
& &&	and
	or
~	as in formulae
-> ->>	rightwards assignment
<- <<-	assignment (right to left)
=	assignment (right to left)
?	help (unary and binary)

Within an expression operators of equal precedence are evaluated from left to right except where indicated.
(Note that = is not necessarily an operator.)

Präzedenz und Ausführungsreihenfolge arithmetischer Operatoren

Operator	Reihenfolge
\wedge	Rechts nach links
$-x, +x$	Unitäres Vorzeichen, links nach rechts
$*, /$	Links nach Rechts
$+, -$	Links nach Rechts

Beispiele

2^2^3	# $2^{(2^3)} = 2^8 = 256$
$(2^2)^3$	# $(2^2)^3 = 4^3 = 64$
-1^2	# $-(1^2) = -1$
$(-1)^2$	# $(-1)^2 = 1$
$2+3/4*5$	# $2+(3/4)*5 = 2+(0.75*5) = 2+3.75 = 5.75$
$2+3/(4*5)$	# $2+3/(4*5) = 2+3/20 = 2+0.15 = 2.15$

Getting started

Arithmetik, Logik und Präzedenz

Variablen

Datenstrukturen

Programmierübungen und Selbstkontrollfragen

In der Programmierung ist eine Variable ein abstrakter Behälter für eine Größe, welche im Verlauf eines Rechenprozesses auftritt. Im Normalfall wird eine Variable im Quelltext durch einen Namen bezeichnet und hat eine Adresse im Speicher einer Maschine. Der durch eine Variable repräsentierte Wert kann – im Unterschied zu einer Konstante – zur Laufzeit des Rechenprozesses verändert werden.

Wikipedia

Grundlagen

- Variablen sind vom Programmierenden benannte Platzhalter für Werte
- In 3GL Sprachen wird der Variablentyp durch eine Initialisierungsanweisung festgelegt:

```
VAR A : INTEGER      # A ist eine Variable vom Typ Integer (ganze Zahl)
```

- In 3GL Sprachen wird Variablen durch eine Zuweisungsanweisung ein Wert zugeschrieben:

```
A := 1               # Der Variable A wird der numerische Wert 1 zugewiesen
```

- In 4GL Sprachen wie Matlab, Python, R werden Variablen durch Zuweisung initialisiert:

```
a = 1               # a ist eine Variable vom Typ double, ihr Wert ist 1
```

- Der Zuweisungsbefehl in Matlab und Python ist =, der Zuweisungsbefehl in R ist <- oder =.
- Offiziell empfohlen für R ist <-.

```
a <- 1              # a ist eine Variable vom Typ double, ihr Wert ist 1
```

```
a = 1              # a ist eine Variable vom Typ double, ihr Wert ist 1
```


Beispiel

Greta geht ins Schreibwarengeschäft und kauft vier Hefte, zwei Stifte und einen Füller. Wie viele analoge Gegenstände kauft Greta insgesamt?

Wir definieren zunächst alle Variablen:

```
hefte <- 4      # Definition der Variable 'hefte' und Wertzuweisung 4
stifte <- 2     # Definition der Variable 'stifte' und Wertzuweisung 2
fueller <- 1    # Definition der Variable 'fueller' und Wertzuweisung 1
```

Nach Zuweisung existieren die Variablen im Arbeitsspeicher, dem sogenannten *Workspace*. Die Variablen können jetzt wie Zahlen in Berechnungen genutzt werden

```
gesamt <- hefte + stifte + fueller # Berechnung der Gegenstandsanzahl
print(gesamt)
```

```
[1] 7
```

Ein Heft kostet einen Euro, ein Stift kostet zwei Euro, und ein Füller kostet 10 Euro. Wie viel Euro muss Greta insgesamt bezahlen?

```
gesamtpreis <- hefte * 1 + stifte * 2 + fueller * 10 # Berechnung des Preises
print(gesamtpreis)
```

```
[1] 18
```

`print()` gibt Variablenwerte in der R Konsole aus.

Der Workspace | (globaler) Arbeitsbereich

- Enthält alle aktuell definierten Objekte, Variablen und Funktionen während einer R-Sitzung.
- Nach Zuweisung werden Variablen automatisch im Workspace gespeichert.
- Ausgabe aller existierenden benutzbaren Variablen im Arbeitsspeicher in der Konsole.

```
ls()           # Anzeigen aller Variablennamen im aktuellen Workspace
```

```
[1] "fueller"      "gesamt"      "gesamtpreis" "hefte"      "stifte"
```

Löschen von Variablen

- Löschen bestimmter Variablen

```
rm(gesamtpreis) # Löschen der Variable Gesamtpreis  
ls()
```

```
[1] "fueller" "gesamt"  "hefte"   "stifte"
```

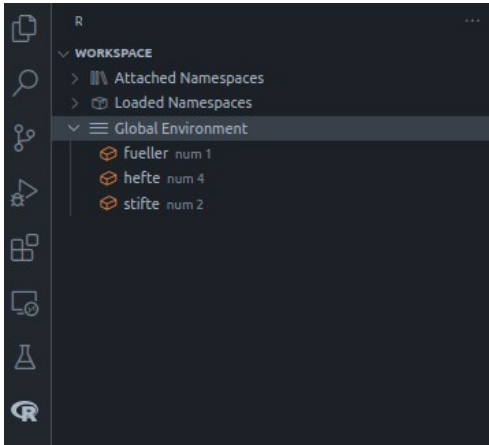
- Löschen aller Variablen im Workspace

```
rm(list = ls()) # Löschen aller Variablen  
ls()
```

```
character(0)
```

Workspace

In VSCode kann der Workspace in the R Extension pane angezeigt werden



Zulässige Variablennamen

- bestehen aus Buchstaben, Zahlen, Punkten (.) und Unterstrichen (_).
- beginnen mit einem Buchstaben oder . nicht gefolgt von einer Zahl.
- dürfen keine *reserved words* wie `for`, `if`, `NaN`, usw. sein (siehe `?reserved`).
- werden unter `?make.names()` beschrieben.

Sinnvolle Variablennamen

- sind kurz (\approx 1 bis 9 Zeichen) und **aussagekräftig**.
- bestehen nur aus Kleinbuchstaben und Unterstrichen.

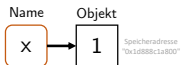
Anmerkung

- R ist *case-sensitive*. Das heißt z.B. $x \neq X$

Variablenrepräsentation | Binding

```
x <- 1
```

- Intuitiv wird eine Variable genannt x mit dem Wert 1 erzeugt.
- De-facto geschehen zwei Dinge:
 1. R erzeugt ein Objekt (Vektor mit Wert 1) mit Speicheradresse '0x1d888c1a800'.
 2. R verbindet dieses Objekt mit dem Namen x, der das Objekt im Speicher referenziert.

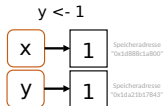
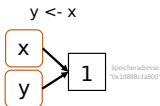


```
y <- x
```

- Intuitiv wird eine Variable genannt y mit Wert gleich dem Wert von x erzeugt.
- De-facto wird ein neuer Name y erzeugt, der dasselbe Objekt referenziert wie x.
- Das Objekt (Vektor mit Wert 1) wird nicht kopiert, R spart Arbeitsspeicher.

```
y <- 1
```

- R erzeugt ein *neues* Objekt (Vektor mit Wert 1) mit *eigener* Speicheradresse '0x1da21b17843'.
- De-facto wird ein neuer Name y erzeugt, der ein anderes Objekt referenziert wie x.



Speicheradressen | object address

- Speicheradressen können mit der Funktion `lobstr::obj_addr()` angezeigt werden.

```
library(lobstr) # Paket `lobstr` laden  
x <- 1  
obj_addr(x)
```

```
[1] "0x5ad84ecb12e8"
```

- `y` bekommt die Zuweisung `x`. De-facto referenziert `y` dieselbe Speicheradresse wie `x`.

```
y <- x  
obj_addr(y)
```

```
[1] "0x5ad84ecb12e8"
```

- `y` bekommt die Zuweisung zu einem neuen Objekt (Vektor mit Wert 1). De-facto referenziert `y` eine andere Speicheradresse wie `x`.

```
y <- 1  
obj_addr(x)
```

```
[1] "0x5ad84ecb12e8"
```

```
obj_addr(y)
```

```
[1] "0x5ad84f2dcac0"
```

Anmerkungen:

- Ausdrücke der Art `lobstr::obj_addr()` referieren eine Funktion, hier `obj_addr()`, bei gleichzeitiger Angabe des Pakets, hier `lobstr`.
- Bevor Funktionen eines Pakets verwendet werden können, muss es mit `library()` geladen werden. Ausnahmen sind R Standardpakete, die per default geladen werden, z.B. `base` oder `stats`.

Das Prinzip

- Ein R Paket ist eine Sammlung von R-Funktionen, Datensätzen und vordefinierten Skripten, die in einem strukturierten Format (Paket) gebündelt sind.
- Analog zu Computerprogrammen (wie MS Word) müssen R Pakete zuerst auf dem Rechner *installiert* werden, bevor sie verwendet werden können. Ausnahmen sind Standardpakete (*standard library*), die automatisch mit der Installation von R mitgeliefert werden.
- Analog dazu, dass Computerprogramme zuerst *gestartet* werden müssen, bevor damit gearbeitet werden kann, müssen R Pakete zuerst in die Workspace *geladen* werden, bevor damit gearbeitet werden kann.

Paket-Management

- Anzeige und Auswahl aller installierter Pakete in der R Extension: *HELP PAGES* → *Home* → *Packages*
- Paket-Management mit Befehlen:

```
installed.packages()      # Alle installierten Pakte anzeigen
install.packages("lobstr") # Paket "lobstr" installieren
library(lobstr)           # Paket "lobstr" laden
```

- Tipp: Wenn in einem Datenanalyseskript Pakete außerhalb der *standard library* verwendet werden, sollte das Laden des Pakets auch ein Befehl im Analyseskript sein!

Variablenrepräsentation | Copy-on-modify

```
x <- 1          # Objekt (0x74b) erzeugt, x referenziert Speicheradresse des Objektes
y <- x          # Der Variable y wird der selbe Wert wie x zugewiesen
obj_addr(y) == obj_addr(x) # Zeigt, dass y dieselbe Speicheradresse (0x74b) wie x referenziert
```

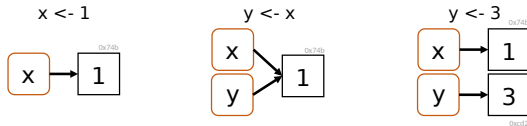
[1] TRUE

```
y <- 3          # y modifiziert, modifizierte Kopie (0xcd2) wird gespeichert
obj_addr(y) == obj_addr(x) # y und x referenzieren jetzt unterschiedliche Speicheradressen
```

[1] FALSE

```
y <- 1          # Auch wenn der Wert der Zuweisung gleich ist...
obj_addr(y) == obj_addr(x) # ...referenzieren y und x nicht die selbe Adresse
```

[1] FALSE



R Objekte sind *immutable*, können also nicht verändert werden.

Zur Immutability gibt allerdings zwei Ausnahmen, genannt *Modifications-in-place*

1. Objekte mit nur einem gebundenem Namen werden in-place modifiziert

```
x <- c(1,2) # Objekt (0x74b) erzeugt, x referenziert Speicheradresse des Objektes  
x[1] <- 2   # Objekt (0x74b) veraendert
```

- Dieses Verhalten ist allerdings nur in R, nicht innerhalb VSCode reproduzierbar.
2. Environments werden in-place modifiziert (→ Environments und Funktionen).

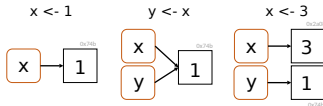
Copy-on-modify gilt auch in umgekehrter Reihenfolge

```
x <- 1          # Objekt (0x74b) erzeugt, x referenziert Speicheradresse des Objektes
y <- x          # Zuweisung y zu gleicher Speicherzelle wie x
obj_addr(y) == obj_addr(x) # zeigt, dass y dieselbe Speicheradresse wie x (0x74b) referenziert.
```

[1] TRUE

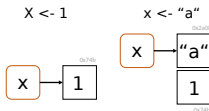
```
x <- 3          # Ein neues Objekt mit Wert 3 wird in Speicherzelle (0x2a08) erzeugt.
                # x referenziert jetzt Objekt (0x2a08)
                # y referenziert weiterhin Objekt (0x74b)
obj_addr(y) == obj_addr(x) # zeigt, dass y und x nicht mehr dieselbe Speicherzelle adressieren.
```

[1] FALSE



Unbinding

```
x <- 1      # x referenziert Objekt (0x74b)
x <- "a"     # x referenziert Objekt (0x2a08), Objekt (0x74b) jetzt ohne Referenz
```



Garbage collection

- Nicht referenzierte Objekte im Arbeitsspeicher werden automatisch gelöscht.
- Das Löschen geschieht meist erst dann, wenn es wirklich nötig ist.
- Es ist nicht nötig, aktiv die Garbage Collection Funktion `gc()` zu benutzen.

Getting started

Arithmetik, Logik und Präzedenz

Variablen

Datenstrukturen

Programmierübungen und Selbstkontrollfragen

Fundamentale Datenstrukturen

- Vordefiniert innerhalb der Programmiersprache
- Logische Werte (logical): TRUE, FALSE
- Ganze Zahlen (integer): int8 (-128,...,127), int16 (-32768,..., 32767)
- Gleitkommazahlen (single, double): 1.23456, 12.3456, 123.456, ...
- Zeichen (character): "a", "b", "c", "!"
- Datentyp-spezifische assoziierte Operationen
 - AND, OR (logical) +, - (integer) +,-,*, / (single), Zeichenkonkatenation (character)

Zusammengesetzte Datenstrukturen

- Vordefinierte Container zur Zusammenfassung mehrerer Variablen gleichen Datentyps. (z.B. Vektoren, Listen, Arrays, Matrizen, ...)
- Container-spezifische Operationen (z.B. Vektorindizierung, Matrixmultiplikation, ...)

Selbstdefinierte Datenstrukturen

- Definition eigener Datenstrukturen aus vordefinierten Datenstrukturen und Containern
- Definition eigener Operationen

Fundamentale Datenstrukturen

- Welche fundamentalen Datenstrukturen bietet die Sprache an?
- Welche Operationen darauf sind bereits definiert?
- Wie lautet die Syntax zur Definition einer Variable eines fundamentalen Datentyps?
- Wie lautet die Syntax, um vordefinierte Operationen aufzurufen?

Zusammengesetzte Datenstrukturen

- Welche Container und zugehörige Operationen bietet die Programmiersprache?
- Wie lautet die Syntax zum Umgang mit einem Container?

Selbstdefinierte Datenstrukturen

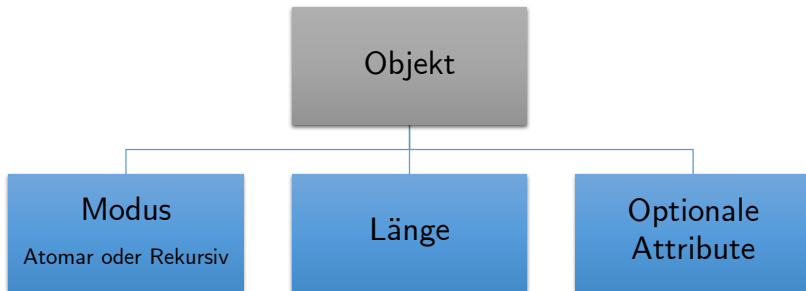
- Wie erzeugt man selbstdefinierte Datenstrukturen und zugehörige Operationen?
- Wie lautet die Syntax zum Umgang mit einer selbstdefinierten Datenstruktur?

Alles, was in R vorkommt, ist ein **Objekt**.

Jedem Objekt kann eindeutig zugeordnet werden:

- ein **Modus**
 - Atomar | Komponenten sind vom gleichen Datentyp.
 - Rekursiv | Komponenten können von unterschiedlichem Datentyp sein.
- eine **Länge**
- optional weitere **Attribute**

Alles, was in R vorkommt, ist ein **Objekt**.



Übersicht der R Datentypen

Datentyp	Erläuterung
logical	Die beiden logischen Werte TRUE und FALSE
double	Gleitkommazahlen
integer	Ganze Zahlen
complex	Komplexe Zahlen, hier nicht weiter besprochen
character	Zeichen und Zeichenketten (strings), 'x' oder "Hallo Welt!"
raw	Bytes, hier nicht weiter besprochen

- Double und integer werden zusammen auch als **numeric** bezeichnet.
- Viele weitere Typen, hier relevant sind **logical**, **double**, **integer**, **character**.

Übersicht der R Datentypen

Automatische Festlegung von Datentypen durch Zuweisung

```
b = TRUE          # logical
x = 2.5           # double
y = 1L            # (long) integer
c = 'a'           # character
```

Ausgabe von Datentypen durch typeof()

```
typeof(b)
```

```
[1] "logical"
```

```
typeof(x)
```

```
[1] "double"
```

```
typeof(y)
```

```
[1] "integer"
```

```
typeof(c)
```

```
[1] "character"
```

Übersicht der R Datentypen

Testen von Datentypen durch `is.*()`

```
is.logical(x)
```

```
[1] FALSE
```

```
is.double(x)
```

```
[1] TRUE
```

Übersicht atomare Datenstrukturen in R

Datenstruktur	Erläuterung
Vektor	Container von indizierten Komponenten identischen Typs
Matrix	Interpretation eines Vektors als zweidimensionaler Container
Array	Interpretation eines Vektors als mehrdimensionaler Container

⇒ (3) Vektoren und (4) Matrizen

Übersicht rekursive Datenstrukturen in R

Datenstruktur	Erläuterung
Liste	Container von indizierten Komponenten beliebigen Datentyps Insbesondere auch rekursive Struktur, z.B. Liste von Listen
Dataframe	Symbiose aus Liste und Matrix Jede Komponente ist Vektor beliebigen Datentyps identischer Länge

⇒ (5) Listen und Dataframes

R und RStudio

Arithmetik, Logik und Präzedenz

Variablen

Datenstrukturen

Programmierübungen und Selbstkontrollfragen

Getting Started

1. Klickt euch durch die VSCode User Interface. (Tipp: Schau in die [VSCode Online-Doku](#))
 - 1.1 Welche verschiedenen Views können in der *Activity Bar* ausgewählt.
 - 1.2 Wähle in der *Activity Bar* den *Explorer* aus. Wie kannst du einstellen, dass der Kursordner, den ihr letzte Woche erstellt habt, in der *Primary Sidebar* sichtbar ist?
 - 1.3 Wähle in der *Activity Bar* die *R Extension* aus. Was ist alles in der *Primary Sidebar* zu sehen?
 - 1.4 Wie kannst du es konfigurieren, dass die R-Console links und nicht unten im *Panel* lokalisiert ist?
 - 1.5 Wie kannst du eine *Secondary Sidebar* öffnen?
2. Gib ein paar einfache Befehle in die Console ein.
3. Öffne die Help-Page für die Funktion `print()`. Was ist der Unterschied zwischen `print(x="Hallo Welt")` und `print("Hallo Welt")`?
4. Was kann über das Argument `digits` spezifiziert werden? Suche unter **Examples** ein Beispiel für die Verwendung dieses Arguments. Was ist der Unterschied zwischen `print(pi)` und `print(pi, digits=2)`?
5. Gib unvollständige Befehl in die Console ein und drücke die Enter-Taste.
(Beispiele: `print("Hello", sqrt(4, x <-, sqrt(1 +)`). Wie kannst du den prompt `'>'` wiederherstellen?

5. Dokumentiert alle bisher eingeführten R Befehle in einem kommentierten R Skript und speichert es in euerem lokalen Kursordner.
 - 5.1 Wie lautet der vollständige Pfad für das R Skript?
 - 5.2 Welche zwei Möglichkeiten gibt es, alle Befehle des Skripts auszuführen? Worin unterscheiden sie sich?
6. Welche zwei Möglichkeiten gibt es, einen oder mehrere R-Befehl in der Console auszuführen?
7. Eläutere den Begriff der Operatorpräzedenz. Wie kann sdiese aus der R-Console heraus nachgeschlagen werden?

Variablen und Workspace

8. Definiere den Begriff der Variable im Kontext der Programmierung.
9. Erläutere die Begriffe Initialisierungsanweisung und Zuweisungsanweisung für Variablen. Wie sieht das in R aus?
10. Gib jeweils ein Beispiel für einen zulässigen und einen unzulässigen Variablennamen in R.
11. Erläutere den Begriff Workspace.
12. Öffne den R Workspace im VSCode User Interface. Führe Zeile für Zeile die Befehle aus dem Greta-Beispiel aus und beobachte, wie definierte Variablen im Workspace erscheinen.
13. Was passiert in der Console, wenn du auf das Kreuz neben der Variable klickst?
14. Erläutere die Prozesse, die R im Rahmen einer Zuweisungsanweisung der Form `x <- 1` durchführt.
15. Zeige, dass zwei Variablen `x` und `y`, die unabhängig voneinander mit dem Wert 7 definiert werden, unterschiedliche Speicheradressen referenzieren.
16. Zeige, dass zwei Variablen `x` und `y`, die abhängig voneinander mit dem Wert 7 definiert werden, dieselbe Speicheradresse referenzieren.
17. Demonstriere, dass R case-sensitive ist.
18. Erläutere die Begriffe *Copy-on-modify* und *Modify-in-place*.

17. Diskutiere die klassischen Datenstrukturen einer 3GL Programmiersprache.
18. Diskutiere die Organisation von Datenstrukturen in R.
19. Wodurch unterscheiden sich eine atomare und eine rekursive Datenstruktur in R?
20. Nenne und erläutere vier zentrale Datentypen in R.
21. Nenne und erläutere drei zentrale atomare Datenstrukturen in R.
22. Nenne und erläutere zwei zentrale rekursive Datenstrukturen in R.