1. R/RStudio Grundlagen

Wie benutze ich überhaupt R?

Dr. Paul Schmidt

Dieses Kapitel richtet sich hauptsächlich an Menschen, die noch nie mit R gearbeitet haben. Allerdings können auch Personen, die R bereits kennen, nützliche Einblicke gewinnen - nämlich in den Abschnitten, in denen ich betone, wie ich persönlich R nutze. Bevor du weitermachst, stelle sicher, dass du R und RStudio installiert hast und dass du das Einführungsvideo zu RStudio angesehen hast – insbesondere solltest du wissen, wie du die wichtigsten Panels wie die **Konsole**, den **Skripteditor** und die **Umgebung** nutzt.

> ℹ Weitere Quellen
>
> Dies ist vermutlich nicht das beste Tutorial der Welt. Deswegen hier mal eine Liste anderer R-Tutorials.

In diesen Kapiteln siehst du R-Code in grauen Boxen und die resultierende Ausgabe in grünen Boxen darunter. Du solltest also auf deinem PC immer das gleiche Ergebnis erzielen können, wenn du denselben Code wie in den grauen Boxen ausführst.

Grundlegende Codeausführung

Du kannst R wie einen Taschenrechner verwenden. Wenn du einfache Operationen schreibst und ausführst, gibt es das Ergebnis in der Konsole zurück:

```
2+3
```

```
[1] 5
```

Wie du siehst, erscheint das Ergebnis (also `5`) nach `[1]`. Darauf gehen wir später genauer ein, aber fürs Erste kannst du die `[1]` ignorieren.

In R spielt es normalerweise keine Rolle, ob und wie viele Leerzeichen du zwischen deine Zahlen, Operatoren usw. setzt. Daher funktioniert auch folgender Code:

```
  2   + 3
```

```
[1] 5
```

Es bleibt also deiner persönlichen Vorliebe überlassen, ob du Leerzeichen vor und nach Operatoren wie `+`, `-` usw. haben möchtest.

> Tipp
>
> Du kannst Kommentare zu deinem Code hinzufügen, indem du das Symbol `#` verwendest. In jeder Zeile wird **alles nach** dem `#` von R ignoriert. Dies ist nützlich, da du dir selbst oder anderen Notizen genau an der relevanten Stelle machen kannst. Zum Beispiel:

```
  2   + 3 # dies addiert 2 + 3
```

```
[1] 5
```

Funktionen

Ähnlich wie in Microsoft Excel kannst du Funktionen in R verwenden. Das erste Beispiel ist `sqrt()`, um die Quadratwurzel einer Zahl zu berechnen:

```
sqrt(9)
```

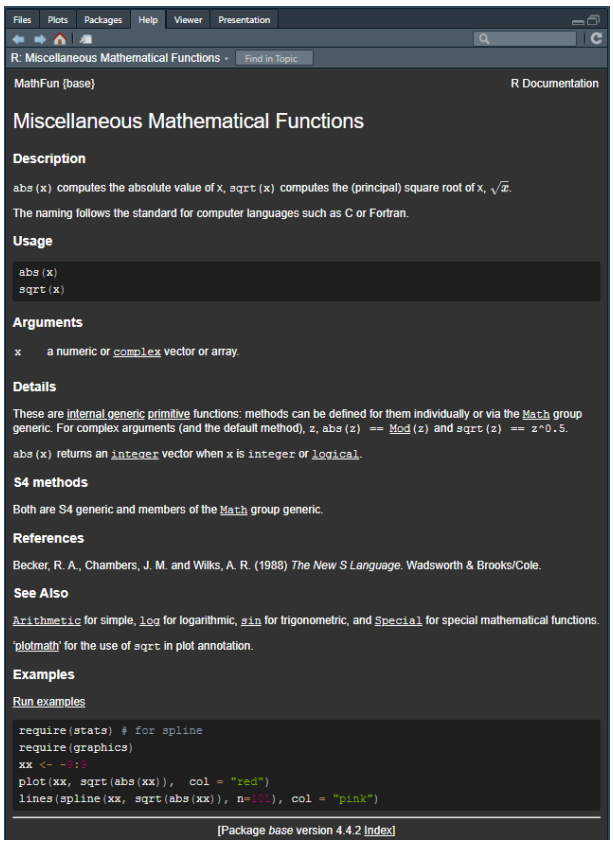
```
[1] 3
```

Genau wie in Excel funktioniert eine Funktion so: Man schreibt den Namen, dann Klammern `()` und (meistens) mindestens eine Information in die Klammern. Wenn du dich fragst, wie eine bestimmte Funktion funktioniert, kannst du jederzeit (auch ohne Internetverbindung) die Dokumentation, also das Handbuch der jeweiligen Funktion, aufrufen, indem du ein Fragezeichen vor den Funktionsnamen setzt und es dann ausführst:

```
?sqrt
```

Wenn das nicht funktioniert, kannst du es mit zwei Fragezeichen versuchen, also `??sqrt`. Das ist nötig, wenn du nach einer Funktion suchst, deren Paket du noch nicht geladen hast. Später werden wir klären, was ein Paket ist.

Die Dokumentation erscheint im **Hilfe-Panel** in RStudio (siehe Screenshot unten) und enthält viele Informationen zur Funktion. Das kann überwältigend wirken, aber es ist wichtig zu wissen, dass der Aufbau der Dokumentation immer gleich ist: Zuerst kommt eine Beschreibung, dann Verwendung und Argumente usw. und meistens am Ende Beispielcode.



Variablen

Neben eingebauten Funktionen kennt R auch bestimmte Dinge wie π oder das Alphabet, die in den eingebauten Konstanten `pi` und `letters` gespeichert sind. Beachte, dass diese keine Klammern haben:

```
pi
```

```
[1] 3.141593
```

```
letters
```

```
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
```

> **Tipp**
>
> Wie schon im oben genannten Video erklärt: Wenn du Code aus deinem Skript ausführen willst, kannst du entweder den Button `Run` oben rechts im Skripteditor klicken oder `Strg + Enter` drücken. Wenn du keinen spezifischen Teil markiert hast, wird die Zeile ausgeführt, in der dein Cursor steht, und der Cursor springt anschließend in die nächste Zeile. Wenn du jedoch einen Teil markiert hast, wird nur dieser Teil ausgeführt.

[1] Macht es einen Unterschied ob ich `<-` oder `=` nutze? Die kurze Antwort ist *Nein*. Die präzisere Antwort gibt's in diesem Video.

Viel nützlicher ist es jedoch, eigene Variablen in R zu definieren. Das geschieht mit dem Zuweisungsoperator `<-` oder dem Gleichheitszeichen `=`[1]. Ersteres ist in R üblicher, letzteres auch in anderen Programmiersprachen wie Python.

Beispiel: Der folgende Code gibt nichts in der Konsole aus, speichert aber die Zahl `5` in der Variablen `x`:

```
x <- 5
```

Um zu prüfen, ob die Variable `x` erstellt wurde und welchen Wert sie hat, kannst du einfach `x` ausführen:

```
x
```

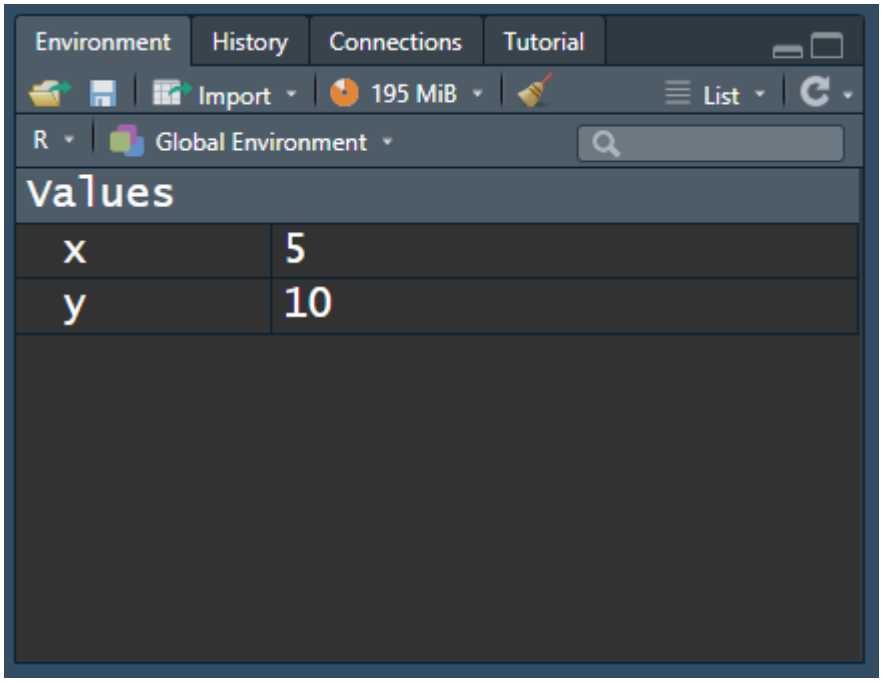
```
[1] 5
```

Wie gesagt, du kannst auch das Gleichheitszeichen `=` verwenden, um einer Variablen einen Wert zuzuweisen:

```
y = 10
y
```

```
[1] 10
```

Alle definierten Variablen kannst du im **Environment-Panel** in RStudio sehen. Dieses Panel zeigt alle bisher definierten Variablen und ihre Werte.



Eine Variable kann überschrieben werden, indem du ihr einen neuen Wert zuweist. Zum Beispiel, wenn wir nun den Wert `7` an `x` zuweisen:

```
x # aktueller Wert von x
```

```
[1] 5
```

```
x <- 7 # überschreibt den Wert von x
x # neuer Wert von x
```

```
[1] 7
```

Du kannst auch Operationen mit Variablen durchführen. Wenn du `x` und `y` wie oben definiert hast, kannst du sie addieren:

```
x + y
```

```
[1] 17
```

Das Ergebnis einer Operation kannst du auch in einer neuen Variablen speichern. Zum Beispiel so:

```
z <- x + y
z
```

```
[1] 17
```

Variablen können nicht nur Zahlen enthalten, sondern z.B. auch Texte. Und in der Praxis solltest du beschreibendere Variablennamen als `x`, `y` oder `z` verwenden. Zum Beispiel:

```
mytext <- "Dies ist mein Text"
mytext
```

```
[1] "Dies ist mein Text"
```

Wie du siehst, schreibt man Texte in Anführungszeichen. So erkennt R, dass es sich um Text (einen sogenannten **String**) und nicht um eine Variable handelt. Man kann `"` oder `'` verwenden, aber man muss am Anfang und Ende denselben Typ verwenden.

Datentypen

Wie du eben gesehen hast, kann R sowohl mit Zahlen als auch mit Text umgehen und noch mit vielen weiteren Datentypen. Jede Variable speichert nicht nur den Wert, sondern auch Informationen über den Typ der Daten. Mit `typeof()` kannst du den Datentyp prüfen:

```
typeof(x)
```

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

```
typeof(mytext)
```

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

`x` ist vom Typ `double` und `mytext` ist vom Typ `character`. Hier eine vereinfachte Übersicht einiger Datentypen:

- Zahlen
  - `integer` / `int`: ganze Zahl (z. B. *42*, *-1504*)
  - `numeric` / `num` & `double` / `dbl`: reelle Zahl (z. B. *3.14*, *0.051795*)
- Text
  - `character` / `chr`: Zeichenketten (z. B. *“hallo”*, *“Zwei Wörter”*)
- Faktor
  - `factor` / `fct`: kategorische Variable mit Levels (z. B. *Kontrolle*, *Behandlung*)
- TRUE/FALSE
  - `logical` / `logi`: logischer Wert (*TRUE* oder *FALSE*)

Vektoren

Statt mit einzelnen Zahlen oder Texten arbeiten wir meist mit ganzen Datensätzen. Ein erster Schritt dorthin ist der *Vektor*: eine **Sequenz von Elementen gleichen Datentyps**. Du kannst dir Vektoren wie eine Spalte in deinem Datensatz vorstellen. Ein Beispiel hatten wir schon: `letters` ist ein Vektor mit 26 Zeichen. Mit `length()` oder `str()` kannst du dir das ansehen:

```
length(letters)
```

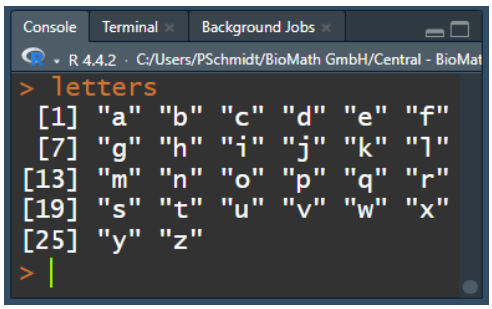
```
[1] 26
```

```
str(letters)
```

```
 chr [1:26] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" ...
```

Beim Ausgeben von `letters` erscheinen Zahlen in eckigen Klammern, z. B. `[1]`, `[20]`. Das sind die Indizes der Elemente im Vektor und das ist auch die `[1]`, die wir vorhin noch ignoriert haben.

Hier ist ein Screenshot aus RStudio:



Du kannst mit diesen Indizes auch gezielt Elemente abrufen:

```
letters[3]
```

```
[1] "c"
```

Eigene Vektoren erzeugst du mit `c()` und Kommas zwischen den Elementen:

```
mynumbers <- c(1, 4, 9, 12, 12, 12, 16)
mynumbers
```

```
[1]  1  4  9 12 12 12 16
```

```
mywords <- c("Hakuna", "Matata", "Simba")
mywords
```

```
[1] "Hakuna" "Matata" "Simba"
```

Funktionen wie `sqrt()` oder `mean()` arbeiten auch mit Vektoren:

```
sqrt(mynumbers)
```

```
[1] 1.000000 2.000000 3.000000 3.464102 3.464102 3.464102 4.000000
```

```
mean(mynumbers)
```

```
[1] 9.428571
```

Vergleichsoperatoren

Mit Vergleichsoperatoren kannst du Werte vergleichen. Die wichtigsten sind:

- Gleich (`==`)
- Ungleich (`!=`)
- Kleiner als (`<`)
- Größer als (`>`)
- Kleiner oder gleich (`<=`)
- Größer oder gleich (`>=`)

Beispiele:

```
5 == 5
```

```
[1] TRUE
```

```
3 < 4
```

```
[1] TRUE
```

```
5 <= 5
```

```
[1] TRUE
```

```
5 != 5
```

```
[1] FALSE
```

```
2 > 6
```

```
[1] FALSE
```

```
5 >= 4
```

```
[1] TRUE
```

Funktionsargumente

Bisher hatten die Funktionen, die wir verwendet haben, gemeinsam, dass sie nur eine Eingabe benötigten. Die wirklich interessanten Sachen in R passieren mit komplexeren Funktionen, die mehrere Eingaben benötigen. Nehmen wir `seq()` als Beispiel, was einfach genug erscheint, weil es eine Sequenz von Zahlen generiert:

```
seq(1, 10)
```

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

Wie man sieht, erzeugt die Eingabe von `1` und `10`, getrennt durch ein Komma, einen numerischen Vektor mit Zahlen von 1 bis 10. Wir möchten jedoch, dass man vollständig versteht, was hier vor sich geht, weil es sehr bei komplexeren Funktionen hilft.

Man kann die Zahlen vertauschen und die Funktion wird wie erwartet funktionieren:

```
seq(10, 1)
```

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

Das bedeutet also, dass die erste Eingabe immer der Startpunkt und die zweite immer der Endpunkt der Sequenz ist, richtig? Nun, ja standardmäßig, aber man kann es auch anders haben, wenn man spezifisch die **Namen der Argumente** verwendet. Die einzelnen Eingaben einer Funktion werden **Argumente** genannt und man kann immer die Reihenfolge der Argumente und ihre Namen in der Dokumentation einer Funktion nachschlagen.

Wenn man sich `?seq()` ansieht, steht dort `seq(from = 1, to = 1)`, also ist dieses `seq(10, 1)` expliziter ausgedrückt: `seq(from = 10, to = 1)`. Und es gibt sogar ein drittes Argument `by =`. Hier ist der Beweis, dass man die Funktion auch mit expliziten Argumentnamen schreiben kann und sie das exakt gleiche Ergebnis zurückgibt:

```
seq(from = 1, to = 10, by = 1)
```

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

Nochmals: Wenn man die Argumente nicht so ausschreibt, wird R einfach die Standardreihenfolge annehmen: Die erste gelieferte Zahl ist `from =`, die zweite ist `to =` und die dritte ist `by =` - so wurde diese Funktion einfach erstellt/programmiert. Wenn wir jedoch die Argumentnamen ausschreiben, können wir sie neu anordnen und jede beliebige Reihenfolge verwenden:

```
seq(1, 9, 2) # Beispiel A
```

```
[1] 1 3 5 7 9
```

```
seq(from = 1, to = 9, by = 2) # Beispiel B
```

```
[1] 1 3 5 7 9
```

```
seq(from = 1, by = 2, to = 9) # Beispiel C
```

```
[1] 1 3 5 7 9
```

```r
seq(1, 2, 9) # Beispiel D
```

```
[1] 1
```

Kurz gesagt: Wenn man versteht, warum die Beispiele A-C oben das gleiche Ergebnis produzieren, aber Beispiel D nicht, dann ist man bereit!

R-Pakete

Jede Funktion, die man jemals in R verwenden wird, ist immer Teil eines R-Pakets. Ein R-Paket ist eine Sammlung von Funktionen, Daten und Dokumentation.

base R

Die Funktionen, die wir bisher verwendet haben, sind Teil des sogenannten **base R**. Das ist die Sammlung von Funktionen/Paketen, die mit jeder Installation von R mitkommt. Selbst wenn man R gerade erst installiert hat, stehen bereits eine ganze Reihe von Paketen zur Verfügung, was man im **Packages**-Panel in RStudio sehen kann. Hier kann man alle installierten Pakete sehen und auch sehen, welche geladen sind. Ein geladenes Paket wird mit einem Häkchen davor angezeigt. Wenn man eine Funktion aus einem Paket verwenden möchte, muss man das Paket zuerst laden. Wie jedoch bereits gesagt, muss man die base R Funktionen/Pakete nicht laden, weil sie immer geladen sind, weshalb wir die Funktionen oben verwenden konnten.

Pakete laden

Um ein Paket zu laden, kann man die `library()`-Funktion verwenden. Zum Beispiel gibt es ein Paket namens `tools`, das standardmäßig auf dem Computer installiert ist, aber nicht geladen wird. Wenn man eine Funktion aus diesem Paket verwenden möchte, muss man es zuerst laden.

```
library(package_name_here)
```

Dieser Befehl muss **einmal jedes Mal (!)** ausgeführt werden, wenn man eine neue R-Sitzung öffnet, was grundsätzlich bedeutet, jedes Mal wenn man RStudio öffnet.

Pakete installieren

So richtig glänzt R aber durch die Fähigkeit, zusätzliche Pakete aus externen Quellen zu installieren. Grundsätzlich kann jeder eine Funktion erstellen, sie in ein Paket packen und online verfügbar machen. Einige Pakete sind sehr ausgereift und beliebt - so wurde z.B. das Paket `{ggplot2}` 168 Millionen Mal heruntergeladen. Um ein Paket zu installieren, ist der Standardbefehl `install.packages("package_name")`. Alternativ kann man auch auf den “Install”-Button oben links im **Packages**-Tab klicken und dort den *package_name* eingeben.

```
install.packages("package_name_here")
```

Sobald man ein Paket erfolgreich installiert hat, wird es in der Liste der Pakete im **Packages**-Tab erscheinen. Es wird jedoch kein Häkchen haben, was bedeutet, dass man es immer noch mit der `library()`-Funktion laden muss, wenn man seine Funktionen verwenden möchte:

- Ein Paket muss nur einmal installiert werden, aber
- Ein Paket muss jedes Mal geladen werden, wenn man eine neue R-Sitzung öffnet!

Zusammenfassung

Herzlichen Glückwunsch! Du hast die Einführung in die R-Grundlagen abgeschlossen und die ersten Schritte in die Welt der R-Programmierung gemacht. Du verfügst nun über die grundlegenden Fähigkeiten, die nötig sind, um eigenen R-Code zu schreiben und auszuführen.

> ℹ Wichtige Erkenntnisse
>
> 1. R kann als Taschenrechner mit Operationen wie Addition (+), Subtraktion (-), Multiplikation (*) und Division (/) verwendet werden.
> 2. Variablen ermöglichen es, Werte für spätere Verwendung zu speichern:
>   - Man verwendet den Zuweisungsoperator (`<-`) oder das Gleichheitszeichen (`=`), um Variablen zu erstellen
>   - Variablennamen sollten ihren Inhalt beschreiben
>   - Variablen können verschiedene Datentypen (Zahlen, Text, etc.) enthalten
> 3. Funktionen sind wichtige Werkzeuge in R:
>   - Sie führen spezifische Aufgaben mit Eingaben (Argumenten) aus
>   - Man kann auf die Funktionsdokumentation mit `?function_name` zugreifen
>   - Argumente können nach Position oder nach Namen spezifiziert werden
> 4. R hat mehrere wichtige Datenstrukturen:
>   - Vektoren speichern mehrere Werte des gleichen Typs
>   - Man greift auf Vektorelemente mit eckigen Klammern zu (z.B. `vector[3]`)
>   - Man erstellt Vektoren mit der `c()` -Funktion
> 5. Vergleichsoperatoren wie `==` , `!=` , `<` , `>` geben logische Werte (TRUE/FALSE) zurück
> 6. R-Pakete erweitern die Funktionalität:
>   - Man installiert Pakete einmal mit `install.packages("package_name")`
>   - Man lädt Pakete in jeder Sitzung mit `library(package_name)`
>   - Base R enthält viele eingebaute Funktionen und Pakete
> 7. Man verwendet Kommentare mit `#` , um Code zu dokumentieren und verständlicher zu machen

> ℹ Weitere Quellen
>
> Siehe auch Kapitel 1 Getting started with R and RStudio im Buch “An Introduction to R”

Bibliography