

Blockpraktikum zur Statistik mit R

04. Oktober 2010

Till Breuer, Sebastian Mentemeier und Matti Schneider

Gliederung

- 1 Hinweise zum Praktikum
- 2 Programmstart, Befehlsmodus, Funktionen, Variablen
- 3 Datenstrukturen
 - Vektoren
 - Matrizen
 - Listen
 - Datentabellen
 - Arrays, Zeitreihen, ...
- 4 Programmierung
 - Funktionen in R schreiben
 - Schleifen und Abfragen

Gliederung

- 1 Hinweise zum Praktikum
- 2 Programmstart, Befehlsmodus, Funktionen, Variablen
- 3 Datenstrukturen
 - Vektoren
 - Matrizen
 - Listen
 - Datentabellen
 - Arrays, Zeitreihen, ...
- 4 Programmierung
 - Funktionen in R schreiben
 - Schleifen und Abfragen

Organisation

- 9h - 17h im Computer-Lab 1: Theorie und betreute Aufgabenbearbeitung
- Es darf/soll in 2er Gruppen gearbeitet werden, um die Aufgaben zu lösen
- Materialien (Übungszettel, Folien, Lösungen, ...) befinden sich auf der Praktikums-Homepage: <http://wwwmath.uni-muenster.de/statistik/lehre/WS1011/PrakStat/>

Literatur



Silke Ahlers

Einführung in die Statistik mit R

<http://wwwmath.uni-muenster.de/statistik/lehre/WS1011/PrakStat/Skript.pdf>



Christine Duller

Einführung in die nichtparametrische Statistik mit SAS und R
Springer



Fahrmeir, Künstler, Pigeot, Tutz

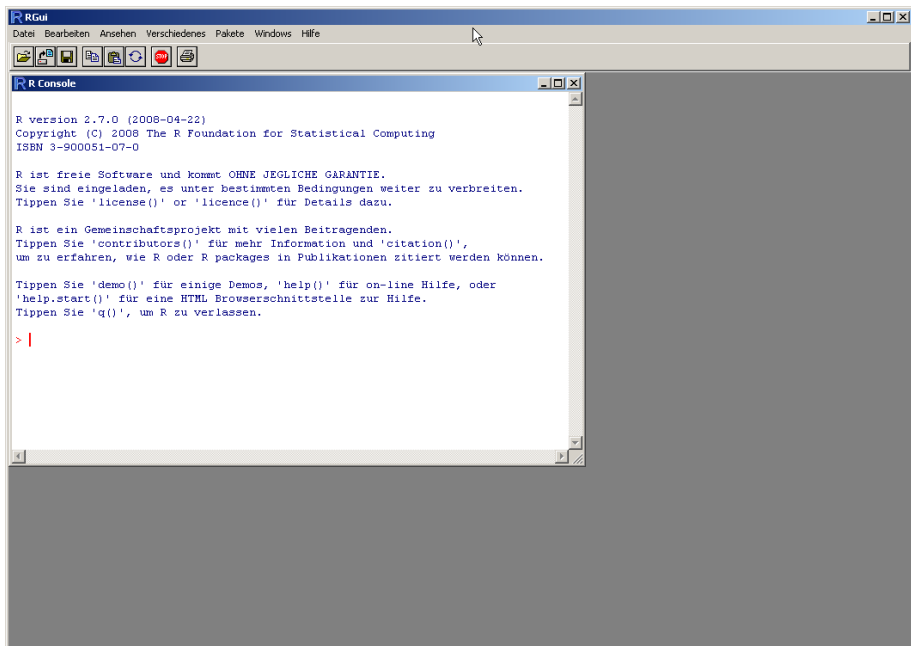
Statistik
Springer

Gliederung

- 1 Hinweise zum Praktikum
- 2 Programmstart, Befehlsmodus, Funktionen, Variablen
- 3 Datenstrukturen
 - Vektoren
 - Matrizen
 - Listen
 - Datentabellen
 - Arrays, Zeitreihen, ...
- 4 Programmierung
 - Funktionen in R schreiben
 - Schleifen und Abfragen

Programmstart

- In den **CIP-Pools** loggen Sie sich unter Windows wie gewohnt ein, und starten Sie dann die Remotedesktopverbindung. Diese finden Sie im Startmenü → (Alle) Programme → Zubehör. Geben Sie als Computer `zivtserv.uni-muenster.de` ein, und klicken Sie auf Verbinden. Es erscheint ein neues Anmeldefenster, in dem Sie sich wiederum mit Ihrem üblichen Benutzernamen und Kennwort einloggen. Wählen Sie als Domäne (Anmelden an) **UNI-MUENSTER**. Nun können Sie R direkt vom Desktop aus starten.
- Im **SR A** loggen Sie sich zunächst unter CentOS ein, öffnen Sie eine Konsole und führen Sie `rdesktop -f zivtserv.uni-muenster.de` aus. Loggen Sie sich unter Windows ein und starten Sie R (Start → Programme → R)



Befehlsmodus

- R bietet eine interaktive Umgebung, den *Befehlsmodus*, in dem man Daten direkt eingeben und analysieren kann
- Der Befehlsmodus dient auch als *Taschenrechner*, z. B. können die Grundrechenarten $+$, $-$, $*$, $/$ direkt eingegeben werden
- Zum Potenzieren muss $^$ benutzt werden

Beispiel

```
> 4+5*5.7
```

```
> 5/6-2
```

```
> 2^3
```

Mathematische Funktionen

- Die Eingabe $2^{0.5}$ liefert das Ergebnis $2^{0.5} = \sqrt{2} \cong 1.414214$
- Einfacher: Eingabe von `sqrt(2)` (`sqrt` = square root)
- Auch andere mathematische Funktionen sind bereits in R implementiert, etwa Logarithmus, Sinus, Cosinus, ...

Beispiel

```
> tan(7/8)
> exp(3.72)
> abs(sin(5))
```

- Eine Übersicht wichtiger Funktionen gibt es auf der Praktikums-Homepage: <http://wwwmath.uni-muenster.de/statistik/lehre/WS1011/PrakStat/R-Befehle.pdf>

obligatorische und optionale Argumente einer Funktion

Es gibt zwei verschiedene Typen von Funktionsargumenten: *obligatorische* und *optionale*. Beispiel:

- Die Funktion `round(x,digits)` rundet die Zahl `x` auf `digits` Stellen
- Die zu rundende Zahl `x` ist hierbei ein *obligatorisches Argument*
- `digits` dagegen ist *optional* mit Standardwert 0, d. h. wird es nicht im Funktionsaufruf übergeben, so rundet R auf eine ganze Zahl

Beispiel

```
> round(x=sqrt(2), digits=3)
> round(x=sqrt(2))
```

Argumentnamen

- Kennt man die Reihenfolge der Argumente im sogenannten *Kopf* der Funktion, so kann man Werte direkt eingeben
- Ist dies nicht der Fall, gibt man sie mit „Name = .“ ein

Beispiel

```
> round(sqrt(2),3)  
> round(digits=3,x=sqrt(2))
```

Hilfeseiten

- Welche Argumente eine Funktion besitzt, lässt sich mit den Hilfeseiten herausfinden, durch die Eingabe `help(round)` oder kürzer `?round`
- Auch mit `args` lässt sich herausfinden, welche Argumente eine Funktion erhalten kann, welche davon optional sind und welche Standardwerte sie ggfls. besitzen

Beispiel

```
> help(cos)
> ?choose
> args(round)
```

Variablenzuweisung I

- Will man mit dem Resultat einer Funktionsauswertung weitere Berechnungen durchführen, so ist es sinnvoll das Ergebnis einer Variable zuzuweisen
- Dies geschieht mit dem *Zuweisungsoperator* `<-`
- Links von diesem Operator steht der Name der Variablen, dem die Daten zugeordnet werden sollen, z. B. `y <- tan(7/8)`
- Durch Eingabe des Variablennames in den Befehlsmodus ruft man den Inhalt der Variablen auf

Beispiel

```
> y <- tan(7/8)
> y
> y+y
```

Variablenzuweisung II

- R achtet auf Groß- und Kleinschreibung: Die Eingabe von Y liefert einen Fehler
- Variablennamen dürfen aus Buchstaben, Zahlen und dem Punkt „.“ bestehen
- Sie beginnen aber immer mit einem Buchstaben
- Will man einen Wert an eine Variable nach rechts übergeben, so geschieht dies mit Hilfe von `->`

Beispiel

```
> 2*y->z
```

```
> z
```

Variablen im Workspace

- Eine Auflistung aller im Workspace benutzten Variablen gibt es mit `ls()`
- Will man eine Variable löschen, so geschieht dies mit `rm()`

Beispiel

```
> ls()  
> rm(z)  
> z
```


Skripte

Eine längere Funktion oder eine Abfolge von vielen Befehlen, sollten nicht direkt im Befehlsmodus definiert werden, sondern in einem *Skript*

- Unter *Datei* → *Neues Skript* kann man ein neues Skript erstellen
- Die in einem Skript geschriebenen Befehle werden erst im Befehlsmodus ausgeführt, wenn man sie markiert und `Strg + R` drückt
- Kommentare schreibt man mit `#`, alle Zeichen dahinter werden ignoriert und nicht im Befehlsmodus ausgeführt
- Mit `Strg + S` speichert man ein Skript als `.R` Datei

Die Lösungen der Aufgaben müssen ebenfalls als R Skript gespeichert werden.

Gliederung

- 1 Hinweise zum Praktikum
- 2 Programmstart, Befehlsmodus, Funktionen, Variablen
- 3 Datenstrukturen**
 - Vektoren
 - Matrizen
 - Listen
 - Datentabellen
 - Arrays, Zeitreihen, ...
- 4 Programmierung
 - Funktionen in R schreiben
 - Schleifen und Abfragen

Vektorenerstellung

- Vektoren erzeugt man in R mit der Funktion `c`
- Die Erzeugung eines Vektors mit den Daten x_1, \dots, x_n erhält man durch Eingabe von `c(x1, ..., xn)`
- Variablenzuweisung funktioniert für Vektoren genauso wie mit “normalen” Zahlen

Beispiel

```
> c(3,0,-4,16)
> c(sin(-5),-5.66, -5*6)
> Messung <- c(6,7,5,5,12)
> Messung
```

Häufig genutzte Vektoren I

Es gibt eine Reihe von Funktionen, die spezielle Vektoren erstellen:

- Einen Vektor mit n Nullen erhält man mit `numeric(n)`
- Der *Doppelpunktoperator* `n:m` erstellt den Vektor $(n, n+1, \dots, m)$ im Fall $n < m$ bzw. den Vektor $(n, n-1, \dots, m)$ im Fall $n > m$
- Der *Sequenz-Befehl* `seq(from, to, by)` erstellt man den Vektor, dessen 1. Eintrag `from` ist und es folgen Werte im Abstand `by` bis zum Punkt `to` (bzw. dem nächstkleineren Wert, falls `to` kein Vielfaches)

Beispiel

```
> numeric(10)
> 1:10
> -3:-7
> seq(from=0,to=100,by=2)
```

Häufig genutzte Vektoren II

- Will man das Intervall $[a, b]$ in n gleichgroße Stücke teilen:
`seq(from=a, to=b, length.out=n)`
- Mit dem *Repeat-Befehl* `rep(x, times=n)` wird der Vektor x sooft wiederholt, wie das Argument `times` angibt
 - Ist n eine natürliche Zahl so wird x genau n mal hintereinander geschrieben
 - Ist n ein Vektor gleicher Länge, so wird der i -te Eintrag von x genau n_i mal dupliziert

Beispiel

```
> seq(0,1,length.out=1000)
> rep(1:2,3)
> rep(1:5,5:1)
```

Komponentenzugriff I

Auf die Einträge eines Vektors x greift man mit $x[n]$ zu. Beispiele:

- Die i -te Komponente von x erhält man durch $x[i]$
- Die 2., 4. und 7. Komponente erhält man durch $x[c(2,4,7)]$
- Negative Werte lassen den entsprechenden Eintrag weg, mit $x[c(-1,-3)]$ würde man alle Einträge bis auf den 1. und 3. erhalten
- Negative und positive Werte dürfen nicht gemischt werden

Beispiel

- > Messung[4]
- > Messung[2:5]
- > Messung[-3]

Logische Operatoren

Will man Komponenten nach Bedingungen auswählen, so geschieht dies unter Benutzung von *logischen Operatoren*

==	gleich	!=	ungleich
<	kleiner	>	größer
<=	kleiner gleich	>=	größer gleich

Diese Operationen liefern als Ergebnis TRUE oder FALSE bzw. einen Vektor mit diesen Einträgen

Beispiel

```
> 4 < 2
```

```
> 2*6 == 12
```

```
> Messung >= 7
```

Komponentenzugriff II

Besteht bei $x[n]$ der Vektor n aus TRUE und FALSE Einträgen, so enthält $x[n]$ den i -ten Eintrag von x , falls n_i TRUE ist

Beispiel:

- Durch `Messung[Messung >= 7]` erhält man alle Messungen, die einen Wert größer (oder gleich) 7 besitzen
- Die Positionen dieser Werte innerhalb des Vektors kann man mit der Funktion `which` herausfinden

Beispiel

```
> Messung[Messung>6]
> which(Messung>=7)
> Messung[Messung==5]
```


Logische Vektorfunktionen

- Die Funktion `any(n)` überprüft, ob in einem logischen Vektor `n` *mindestens* ein `TRUE` vorkommt
- Die Funktion `all(n)` überprüft, ob in einem logischen Vektor `n` *alle* Einträge `TRUE` sind
- Mit `TRUE` und `FALSE` lässt sich rechnen, als wären es die Zahlen 1 bzw. 0

Beispiel

```
> any(c(TRUE, FALSE, TRUE))  
> any(Messung < 6)  
> all(Messung < 6)  
> exp(TRUE) + 5 * TRUE
```

Weitere Vektorfunktionen

Es gibt eine Fülle nützlicher Funktionen, die man auf einen Vektor anwenden kann

- Mit `sum(x)` erhält man etwa die Summe aller Komponenten von `x`
- Mit `min(x)` das Minimum der Einträge, mit `max(x)` das Maximum
- Mit `length(x)` die Anzahl der Komponenten von `x`
- Mit `sort(x)` sortiert man `x` *aufsteigend*, das optionale Argument `decreasing` ändert dies

Beispiel

```
> prod(Messung)
> sort(Messung)
> sort(Messung, decreasing = TRUE)
```

Vektorarithmetik I

- Auch elementare Funktionen (Addition einer Zahl, Sinusfunktion, etc.) lassen sich auf Vektoren anwenden
- Die Auswertung geschieht *komponentenweise*
- Vorsicht: Der Doppelpunktoperator bindet stärker als Addition, Multiplikation, usw., d. h. $1:5+1$ ist von $1:(5+1)$ verschieden

Beispiel

```
> log(Messung)
> Messung * 4
> round(exp(Messung), 2)
> 1:5+1
> 1:(5+1)
```

Vektorarithmetik II

- Ebenso nützlich ist die Vektorarithmetik $v+w$, $v*w$, $v^{\wedge}w$, ... für Vektoren $v = (v_1, \dots, v_n)$ und $w = (w_1, \dots, w_m)$
- Für $m = n$ geschieht die Auswertung komponentenweise
- Ist n ein Vielfaches von m , so geschieht die Auswertung zyklisch, d. h.

$$v + w = (v_1 + w_1, \dots, v_m + w_m, v_{m+1} + w_1, \dots, v_n + w_m)$$

Der entstehende Vektor hat demnach Länge n

Beispiel

```
> Messung - 1:5  
> c(8,2,4,3) + c(-10,20)  
> (1:6)^(2:3)
```

Qualitative Merkmale (Faktoren)

- Für die Eingabe von Daten bei einem qualitativem Merkmal müssen die Komponenten aus *Zeichenketten* bestehen, die in Hochkommata eingeschlossen sind, z. B. `c("blau", "grün", "gelb")`
- Zur Beseitigung der Hochkommata dient die Funktion `factor`, diese macht aus einem Vektor ein *Faktor*, d. h. ein qualitatives Merkmal
- Die *Level* (unterschiedliche Ausprägungen) des Faktors `f` erhält man mit `levels(f)`

Beispiel

```
> Geschlecht <- c("m", "w", "w", "m", "w")  
> Geschlecht <- factor(Geschlecht)  
> Geschlecht  
> levels(Geschlecht)  
> Messung[Geschlecht=="w"]
```

Einträge benennen

Die Einträge eines Vektors v lassen sich für einen einfacheren Zugriff benennen

- Syntax: `v <- c(Name1=Wert1, Name2=Wert2, ...)`
- Die Ausgabe im Befehlsmodus ist dann zweizeilig: In der 1. Zeile stehen die Namen, in der 2. die Einträge (Werte) des Vektors
- Alternativer Zugriff auf die Einträge von v mit `v["Name1"]`

Beispiel

```
> v <- c(Vorname="Max", Nachname="Mustermann")  
> v  
> v["Nachname"]
```

Einträge umbenennen

- Abfrage der Namenseinträge durch `names(v)`
- Umbenennen mit `names(v) <- n` wobei `n` ein Vektor gleicher Länge wie `v` ist, der aus Zeichenketten besteht
- Löschung mit `names(v) <- NULL`

Beispiel

```
> names(Messung) <- c("a", "b", "c", "d", "e")  
> Messung  
> Messung["e"]
```

Die Abschnitte 1 (Grundlagen) und 2 (Vektoren) des Aufgabenblattes können jetzt bearbeitet werden.

Matrizen

- Mit dem Befehl `matrix(data,nrow,ncol,byrow(=FALSE))` lässt sich eine *Matrix* in R erzeugen.
- `data` ist der Vektor, mit dem die Matrix gefüllt werden soll
- `nrow` die Anzahl der Zeilen, `ncol` die Anzahl der Spalten (es genügt, wenn man eines angibt)
- Optional: `byrow = TRUE` gibt an, dass `data` Zeilenweise in die Matrix eingefüllt wird

Beispiel

```
> matrix(1:9,3)
> matrix(1:12,3,byrow=TRUE)
> A<-matrix(c(3,4,0,1),2)
> B<-matrix(9:6,2)
```

Zugriff auf Matrizeneinträge

Auf die Einträge einer Matrix greift man wieder mit dem `[]` Operator zu. Hier muss dieser jedoch die Koordinate des Eintrags enthalten.

- `A[3,2]` liefert den 3. Zeileneintrag der 2. Spalte der Matrix `A`
- Mit `A[,2]` erhält man die komplette 2. Spalte von `A`
- Mit `A[3,]` die komplette 3. Zeile
- Mit `A[c(2,7,13),]` die 2., 7. und 13. Zeile (als Matrix)

Beispiel

```
> A[1,1] + A[2,2]
```

```
> A[2,] * A[,1]
```

Matrizenfunktionen

Auch für Matrizen gibt es in R eingebaute Funktionen

- Transponieren `t(A)`
- Matrizenmultiplikation `A %*% B`
- Determinantenberechnung `det(A)`
- lösen linearer Gleichungssysteme $Ax = b$ mit `solve(A,b)`
- ...

Beispiel

```
> A %*% B  
> det(A)  
> solve(A,c(5,2))
```

Listen

Eine Liste besteht aus Elementen beliebigen Typs

- Erstellung mit `l<-list(name1=wert1, name2=wert2, ...)`
- Zugriff auf das 1. Element mit `l$name1` oder `l[[1]]`

Beispiel

```
> l<-list(Text=c("bla","blu"), Mat=A)
> l
> l[[1]]
> l$Mat
```

Einträge einer Liste herausfinden

Viele Funktionen in R liefern eine Liste als Ergebnis zurück. Die Namen der Einträge einer Liste `l` findet man mit Hilfe von `str(l)` heraus. Beispiel:

- Die Funktion `eigen(A)` berechnet numerisch Eigenwerte und -vektoren der Matrix `A`
- Wendet man `str` darauf an, erhält man das Resultat

```
List of 2
```

```
$ values : num [1:2] 3 1
```

```
$ vectors: num [1:2, 1:2] 0.447 0.894 0 1
```

Beispiel

```
> eig<-eigen(A)
> str(eig)
> eig$values
> eig$vectors
```

Datentabellen

- Nachteil von Matrizen: Nur Zahlenwerte als Einträge
- In *Datentabellen* dagegen können Werte von Merkmalen *unterschiedlichen* Typs (Zahlen, Faktoren) gespeichert werden
- Jedem Merkmal muss dabei die gleiche Anzahl von Beobachtungen zugrundeliegen
- Befehl `data.frame(merkmal1=werte1,merkmal2=werte2,...)`

Beispiel

```
> Tabelle <- data.frame(Geschlecht = c("m", "w", "w"),  
  Alter = c(24,32,20))  
  
> Tabelle
```

Zugriff auf Datentabellen

- Zugriff auf Einträge wie bei Matrizen mit dem `[]` Operator
- Auf die Spalten kann man, wie bei Listen, mit dem `$` Operator zugreifen, d. h. die „Alter“-Spalte erhält man durch `Tabelle$Alter`

Beispiel

```
> Tabelle[3,2]  
> Tabelle[2,]  
> Tabelle$Geschlecht
```

Hinzufügen von Spalten oder Zeilen

- Spalten hinzufügen mit `cbind(x1, x2, ...)`
- Die Funktion “verbindet” die Datentabellen (oder Vektoren, Matrizen) `x1, x2, ...` zu einer Datentabelle (oder Matrix)
- `x1, x2, ...` müssen die gleiche Anzahl von Zeilen besitzen, oder die Zeilenanzahlen sind Vielfache voneinander (zyklisches Verbinden)
- Analog: Zeilen verbinden mit `rbind(x1, x2, ...)`

Beispiel

```
> rbind(Tabelle, c("m",27))  
> cbind(B, c(9,4))  
> rbind(1:3, 9:7, -2:0)  
> rbind(matrix(1:16,4),0,-1:-2,101:104)
```


Teiltabellen

- Die Funktion `subset(x, condition, select)` bietet die Möglichkeit, aus einer Tabelle `x` kleinere Datensätze gemäß der Bedingung `condition` auszuwählen
- Mit dem optionalen Argument `select` ist es möglich, nur gewisse Spalten auszuwählen

Beispiel

```
> subset(Tabelle, Geschlecht == "w")  
> subset(Tabelle, Geschlecht == "w", select = Alter)
```

split Funktion

- Die Funktion `split(x,f)` teilt Tabelle `x` in Gruppen ein, die durch den Faktor (Zeichenkettenvektor) `f` definiert werden
- Das Ergebnis ist eine Liste mit sovielen Einträgen, wie `f` Level besitzt
- Jeder Eintrag der Ergebnisliste ist eine Tabelle und es gilt:
Die Zeile i von `x` befindet sich in der Tabelle zur Gruppe g , falls der i -te Eintrag in `f` genau g ist

Beispiel

```
> s <- split(Tabelle, Tabelle$Geschlecht)
> s
> s$m
> s$w
```

Einlesen aus externen Dateien

Damit man große Datensätze nicht mit der Hand abtippen muss, gibt es einige Funktionen, mit denen es möglich ist, externe Dateien einzulesen:

<code>read.table("Pfad")</code>	liest externen Datensatz ein
<code>read.csv("Pfad")</code>	liest durch Kommata getrennte Spalten
<code>read.csv2("Pfad")</code>	liest durch Semikola getrennte Spalten, Komma als Dezimalkomma
<code>read.delim("Pfad")</code>	liest Tab-getrennte Spalten
<code>read.delim2("Pfad")</code>	liest Tab-getrennte Spalten, Komma als Dezimalkomma

Optionale Argumente: `header=TRUE` sorgt dafür, dass die 1. Zeile für die Spaltennamen genutzt wird, in `dec` gibt man den Dezimalpunkt der Datei an ("," oder ".")

Weitere Datenstrukturen

Es gibt noch einige weitere Datenstrukturen, etwa

- *Array* = d -dimensionalen Datensatz (R Befehl: `array`)
- *Zeitreihe* = Vektor mit Zeitangabe (R Befehl: `ts`)
- ...
- Welche Struktur eine Variable hat, lässt sich mit dem Befehl `str` herausfinden

Beispiel

```
> str(Tabelle)
> str(A)
> str(1)
```

Gliederung

- 1 Hinweise zum Praktikum
- 2 Programmstart, Befehlsmodus, Funktionen, Variablen
- 3 Datenstrukturen
 - Vektoren
 - Matrizen
 - Listen
 - Datentabellen
 - Arrays, Zeitreihen, ...
- 4 Programmierung
 - Funktionen in R schreiben
 - Schleifen und Abfragen

Eigene Funktionen schreiben

Eine eigene Funktion `f` erstellt man in R mit

```
f <- function(Argumente) {  
  Körper der Funktion  
  return(Ergebnis)  
}
```

- Die *Argumente* müssen in der Form `arg1, arg2, ...` angegeben werden, optionale Argumente gibt man durch `arg=Wert` an
- Im *Körper der Funktion* können beliebig viele Anweisungen stehen
- Alle im Körper definierten Variablen sind lokal, d. h. man kann sie außerhalb der Funktion nicht nutzen
- Das *Ergebnis* ist das, was die Funktion am Ende “ausspuckt”, also etwa eine Zahl, ein Vektor oder eine Liste

Beispiel: eigene Funktion schreiben

Wir wollen eine Funktion `f` schreiben, die $f(x, y) = x^y - x$ berechnet. `y` soll dabei ein optionales Argument mit Standardwert 2 sein. Dies geht mit

```
f <- function(x, y=2) {  
  erg<-x^y-x  
  return(erg)  
}
```

Danach steht `f` zur Verfügung und wird auch unter `ls()` aufgelistet

Beispiel

```
> f(3)  
> f(3,4)  
> f(1:9,4)
```

Abkürzungen

In einigen Spezialfällen lässt der Code übersichtlicher gestalten

- Ist der Körper der Funktion leer, so kann auf die geschweiften Klammern verzichtet werden (in der Funktion darf dann nur eine Berechnung stehen!)
- Auf die Eingabe von "return" kann verzichtet werden: Die letzte Berechnung im Körper wird dann zurückgegeben (dies darf keine Variablenzuweisung sein!)
- Will man mehrere Anweisungen in *eine* Zeile schreiben, muss man sie mit einem Semikolon voneinander trennen

Beispiel

```
> f2 <- function(x,y=2) { erg<-x^y-x; return(erg) }  
> f3 <- function(x,y=2) { erg<-x^y-x; erg }  
> f4 <- function(x,y=2) x^y-x
```


Allgemeine Programmierhinweise I

- Benutzen Sie ► Skripte
- Benutzen Sie Variablen, um ein Skript flexibel ändern zu können:
Wenn man etwa mit 100 Simulationen arbeiten muss, setzt man `n<-100` und arbeitet mit `n`, um später leicht `n = 1000` Simulationen durchführen zu können
- Verwenden Sie aussagekräftige Variablennamen um den Code lesbarer zu machen: Lieber `sim.anz` statt `n` als Namen für die Simulationsanzahl wählen
- Kommentieren Sie Ihren Code so, dass Sie auch später noch verstehen, was er bewirkt (Kommentare schreibt man mit `#`, alle Zeichen dahinter werden ignoriert)

Allgemeine Programmierhinweise II

- Benutzen Sie die Vektorarithmetik: Mit ihr lassen sich viele `for` Schleifen umgehen und ein Skript braucht in der Regel weniger Laufzeit
- Speichern Sie ein Skript regelmäßig (mit `Strg + S`) um bei einem Programmabsturz nicht alles neu schreiben zu müssen
- Rücken Sie den Code so ein, dass man auf einen Blick sieht, wo sich zusammengehörige `{ }` Klammern befinden

for Schleife

Schleifenprozesse sind Vorgänge, die vom Programm immer wiederholt werden, bis eine gewisse Bedingung erfüllt ist. Die wichtigsten Schleifen sind die `for`- und die `while`-Schleife.

- Die `for`-Schleife unterliegt folgender Syntax:
`for(Name in Vektor) { Körper der Schleife }`
- Dadurch wird eine Variable, die „Name“ heißt, schrittweise gleich den Elementen des Vektors „Vektor“ gesetzt
- In jedem Schritt wird für den entsprechenden Wert des Vektors der zugehörige Befehl aus den geschweiften Klammern ausgeführt
- Die Befehle im Körper der Schleife werden also sooft ausgeführt, wie es Elemente im Vektor „Vektor“ gibt
- Gibt es nur *einen* Befehl, so können die geschweiften Klammern weggelassen werden

Beispiele: for Schleife

Beispiel

Wir wollen die Zahlen von 1 bis 100 aufaddieren.

```
> sum <- 0  
> for (j in 1:100) sum <- sum + j  
> sum
```

Beispiel

Wir wollen die ersten 12 Fibonacci-Zahlen erzeugen:

```
> Fibo <- numeric(12)  
> Fibo[1] <- Fibo[2] <- 1  
> for (i in 3:12) Fibo[i] <- Fibo[i-2]+Fibo[i-1]  
> Fibo
```

Beispiele: for Schleife

Beispiel

Wir wollen die Zahlen von 1 bis 100 aufaddieren.

```
> sum <- 0  
> for (j in 1:100) sum <- sum + j  
> sum
```

Beispiel

Wir wollen die ersten 12 Fibonacci-Zahlen erzeugen:

```
> Fibo <- numeric(12)  
> Fibo[1] <- Fibo[2] <- 1  
> for (i in 3:12) Fibo[i] <- Fibo[i-2]+Fibo[i-1]  
> Fibo
```

while Schleife

Bei einer for-Schleife ist klar, wie oft die Befehle im Körper ausgeführt werden

- Will man aber einen Vorgang wiederholen bei dem man nicht weiß wann die Schleife abbrechen soll, so hilft die `while` Schleife
- Syntax: `while(Bedingung) { Körper der Schleife }`
- Vor jedem Schritt wird die Bedingung überprüft: Solange diese `TRUE` ist, werden die Befehle ausgeführt, tritt `FALSE` ein, wird die Schleife beendet

Beispiele: while Schleife

Beispiel

Wir wollen die Zahlen von 1 bis n aufaddieren, solange bis die Summe größer als 10000 ist

```
> sum <- 0; n<-1  
> while (sum <= 10000) { sum <- sum + n; n <- n + 1 }
```

Beispiel

Wir wollen alle Fibonacci-Zahlen auflisten, die kleiner als 300 sind.

```
> Fib1 <- Fib2 <- Fibonacci <- 1  
> while(Fib2<300) {  
  Fibonacci <- c(Fibonacci,Fib2)  
  oldFib2 <- Fib2  
  Fib2 <- Fib1+Fib2  
  Fib1 <- oldFib2 }
```

Beispiele: while Schleife

Beispiel

Wir wollen die Zahlen von 1 bis n aufaddieren, solange bis die Summe größer als 10000 ist

```
> sum <- 0; n<-1  
> while (sum <= 10000) { sum <- sum + n; n <- n + 1 }
```

Beispiel

Wir wollen alle Fibonacci-Zahlen auflisten, die kleiner als 300 sind.

```
> Fib1 <- Fib2 <- Fibonacci <- 1  
> while(Fib2<300) {  
  Fibonacci <- c(Fibonacci,Fib2)  
  oldFib2 <- Fib2  
  Fib2 <- Fib1+Fib2  
  Fib1 <- oldFib2 }
```


if-else Abfragen

Häufig müssen in die Definition einer Funktion Fallunterscheidungen bzgl. des Outputs in Abhängigkeit von den eingesetzten Werten vorgenommen werden.

- Dazu verwendet man Anweisungen der Art
`if(Bedingung) { Befehlsfolge }`
`else { Befehlsfolge }`
- Liefert die Bedingung TRUE so wird die Befehlsfolge hinter `if` ausgeführt, im anderen Fall die hinter `else`
- Will man im else-Fall nichts tun, so kann man diese Zeile auch weglassen
- Soll nur *ein* Befehl ausgeführt werden, kann man auf die `{}` Klammern verzichten

Beispiel: if-else Abfragen

Beispiel

$g(x)$ soll $\sin(\sqrt{x})$ berechnen, falls $x \geq 0$ ist, andernfalls NaN ausgeben:

```
> g<-function(x) {  
  if(x >= 0) return(sin(sqrt(x)))  
  else return(NaN)}
```

Beispiel

Indikatorfunktion auf dem abgeschlossenen Einheitsintervall:

```
> indikator<-function(x) {  
  if(x < 0) return(0)  
  else {  
    if(x>1) return(0)  
    else return(1)  
  }  
}
```

Beispiel: if-else Abfragen

Beispiel

$g(x)$ soll $\sin(\sqrt{x})$ berechnen, falls $x \geq 0$ ist, andernfalls NaN ausgeben:

```
> g<-function(x) {  
  if(x >= 0) return(sin(sqrt(x)))  
  else return(NaN)}  
#> [1] NaN
```

Beispiel

Indikatorfunktion auf dem abgeschlossenen Einheitsintervall:

```
> indikator<-function(x) {  
  if(x < 0) return(0)  
  else {  
    if(x>1) return(0)  
    else return(1)  
  }  
}
```

Logische Operatoren II

Da in den Fällen $x < 0$ und $x > 1$ der gleiche Wert zurückgegeben wird, könnte man auch beide Bedingungen mit einem logischen ODER miteinander verknüpfen

&	logisches UND
	logisches ODER
!	logisches NICHT

Beispiel

```
> indikator2<-function(x) {  
  if(x < 0 | x > 1) return(0)  
  else return(1)  
}
```

ifelse Funktion

Noch einfacher wird es, wenn man die ifelse Funktion benutzt

- Syntax: `ifelse(Bedingung, Wert1, Wert2)`
- Wie gehabt: Liefert die Bedingung TRUE so wird Wert1 zurückgegeben, andernfalls Wert2
- Sowohl Bedingung als auch die Werte können vektorwertig sein (die Ausgabe ist ein Vektor der so lang ist wie der Bedingungsvektor)

Beispiel

```
> indikator3<-function(x) ifelse(x<0 | x>1, 0, 1)
> ifelse(c(TRUE,FALSE,TRUE,TRUE), 1:2, 0)
```