Code-Handout R Workshop @ CAA de 2019

Sophie C. Schmidt

23 September 2019

Taschenrechner

Man kann R wie einen Taschenrechner benutzen, die einfachen Rechenoperationen stehen zur Verfügung:

```
3 + 2

5 - 7

5 * 2

100 / 10
```

Oder auch:

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

Rechenergebnisse, wie z.B. das Ergebnis von 3*(4+2) können in Variablen gespeichert werden. Die Zuweisung des Ergebnisses zu einer Variablen geschieht mit dem Zuweisungsoperator "<-" und sieht im allgemeinen folgenderweise aus:

Variablenname <- Befehl

Wird nacheinander mehrmals dergleichen Variablen verschiedene Ergebnisse zugewiesen, enthält die Variable das Ergebnis der letzten Zuweisung. Um nachzusehen, was eine Variable enthält, kann man den Variablennamen in die Konsole eingeben und mit Enter abschicken oder oben rechts unter Environment nachsehen. R merkt sich NICHTS, es sei denn, ich weise es einer Variablen zu. Das heißt auch, wenn ich einen Befehl / eine Formel auf einen Datensatz anwende, bleibt das nur langfristig bestehen, wenn ich mit dem Befehl gleichzeitig entweder meinen alten Datensatz überschreibe ODER einen neuen entstehen lasse.

Ergo:

```
x <- 3*(4+2)
```

Oben rechts ist jetzt unter dem Reiter "Environment" der Wert "x" erschienen. Wir können dort immer ablesen, welche Variablen wir zur Zeit definiert haben.

Mit x kann ich jetzt weiterrechnen:

```
y <- x+2
```

Mit dem Befehl *class* kann ich testen, welcher Datentyp eine Variable hat:

```
class(x)
class(y)
```

Da x und y den gleichen Typ haben (*numeric*), kann ich sie in einem Vektor zusammenfassen:

```
z <- c(x,y)
```

Was ist passiert?

Das c() markiert, dass ich mehrere Werte in einer Reihe eingebe, die zusammengehören sollen. Mit <- habe ich diese Reihe der Variablen z zugeordnet.

Wenn ich das alles noch einmal mit anderen Werten mache, kann ich aus den zwei Vektoren einen Dataframe erstellen:

```
a <- "Hund"
b <- "Katze"
```

Die Hochkommas erklären R, dass es sich um Text handelt und nicht um Objekte (also andere Variablen). Vergisst man sie, kommt die Fehlermeldung "object 'Hund' not found", weil R nach etwas, das 'Hund' heißt, sucht und nicht findet.

Jetzt lassen sich diese beiden in einen Vektor zusammengefügt:

```
ab <- c(a,b)
```

Unter "Environment" oben rechts in Rstudio befinden sich jetzt alle neuen Variablen, die wir definiert haben. Wir können auch sehen, dass "ab" "chr" – also ein "character"-Vektor – ist, während z als "num" – numerical – markiert wird.

Jetzt bauen wir aus diesen beiden Vektoren einen Dataframe:

```
df <- data.frame(z, ab)
```

Unter Environment erscheint unter der Überschrift "Data" jetzt df. Auf den blauen Pfeil kann man klicken und sich anschauen, woraus der Dataframe zusammengesetzt ist. Wir können ihn uns auch anschauen, entweder durch "draufklicken" oder per Code:

```
View(df)
```

Cool! Wir haben Daten!

Aber eigentlich wollten wir archäologische Daten für den Workshop benutzen. Netterweise gibt es Menschen, die eine Menge archäologischer Daten als R-Paket zusammengeschnürt haben und zur Verfügung gestellt haben (David L. Carlson und Georg Roth). Installieren wir also das erste Paket! Der Befehl ist *install.packages* und das Paket heißt "archdata":

```
install.packages("archdata")
```

Nach erfolgreicher Installation (roter Text heißt in R nicht, dass Fehler passiert sind!), müssen wir das Paket noch in unsere Sitzung laden, damit wir damit umgehen können:

```
library(archdata)
```

Im Paket archdata liegen mehrere Datensätze. Informationen zu dem Paket finde ich entweder unten rechts unter dem Reiter "Help" (in der Suche nach archdata suchen) oder mit diesem Code:

```
?archdata
```

Ein einfaches Fragezeichen vor einem Funktions- oder Paketnamen führt einen zu der R-internen Hilfe. Immer ein guter Anfang, wenn irgendetwas nicht klappt (und Tipp-Fehler schon ausgeschlossen wurden).

Wir benutzen als erstes den Datensatz "BACups". Er wurde (wie die anderen) im RData-Format abgespeichert, weswegen wir ihn jetzt leicht mit einem einzigen Befehl in das Programm laden können:

```
data("BACups")
```

Eigene Datensätze lassen sich am leichtesten als csv, aber auch als excel-Datei in R laden. R kann man auch mit Datenbanken verbinden und es gibt inzwischen Pakete, die PDF-Tabellen für einen auslesen. Infos dazu gibt's am Ende.

Wir sollten uns den Datensatz aber vorher einmal kurz angucken. Wie ging das noch einmal?

Lagemaße u. ä.

Diesen Datensatz können wir jetzt schon einmal erkunden. Zum Beispiel uns die einzelnen Spalten anschauen.

- $ das Dollarzeichen steht zwischen Dataframe und dem Vector im Dataframe: df$vector, damit wählen wir also den Vector ("die Spalte") an.

```
BACups$Phase
```

Was R mir jetzt einfach "ausspuckt" ist die Abfolge der Werte, die in der Tabellenspalte "Phase" stehen, wobei die Zahlen in eckigen Klammern die Positionen markieren. Außerdem sagt er mir wie viele "levels" der Vektor hat, also wie viele unterschiedliche Werte und wie diese heißen: Protoappenine und Subappenine.

Tun wir das gleiche für einen numerischen Vektor:

```
BACups$RD
```

Das sind die Werte des Randdurchmessers. Ein bisschen unübersichtlich, nicht wahr?

Wenden wir doch ein paar Funktionen darauf an, um uns einen Überblick zu verschaffen. Wir weisen den durch die Funktion errechneten Wert immer gleich einer Variable zu:

Was ist der Mittelwert?

```
RD_mean <- mean(BACups$RD)
```

Du Funktion *mean* wird auf die Spalte "RD" des Dataframes "BACups" angewandt.

Und der Median?

```
RD_med <- median(BACups$RD)
```

Standardabweichung?

```
RD_sd <- sd(BACups$RD)
```

Varianz?

```
RD_var <- var(BACups$RD)
```

Größter und kleinster Wert?

```
RD_range <- range(BACups$RD)
```

Wie viel Werte sind das eigentlich insgesamt? Also wie viele Zeilen im Datensatz?

```
n_BACups <- nrow(BACups)
```

Geht das vielleicht etwas schneller?

```
summary(BACups$RD)
```

Nunja, zumindest Minimal- und Maximalwerte, Median, Mittelwert und Quantile lassen sich so auf einen Blick anzeigen.

Bestimmte Bereiche eines Datensatzes auswählen

- Eckige Klammern [ ]: Sie sind spannend, weil man mit ihnen Zeilen, Spalten und Felder eines Dataframes anwählen kann. Ein kleines Beispiel:

Will ich in dem Datensatz BACups zB die allererste Information (1. Zeile, 1. Spalte, was steht da?) herausholen, geht das so:

```
BACups_1_1 <- BACups[1,1]
```

Wie check ich, ob es geklappt hat? Richtig, oben rechts unter "Environment" steht jetzt BACups_1_1.

Ich kann aber auch die gesamte erste Zeile auslesen:

```
BACups_1 <- BACups[1,]
# BACups_1 ist rechts unter "data", weil es sich um einen Vector handelt. 1 observation, 6 variables steht daneben.
```

Natürlich lassen sich auch Spalten auswählen, hier die erste:

```
BACups_x_1 <- BACups[,1]
```

Folgerichtig kann man sich merken: In der eckigen Klammer hinter dem Datensatz kann man mit der ersten Zahl die Zeile bestimmen und mit der zweiten Zahl hinter einem

Komma die Spalte. Gerade Spalten haben häufig Namen, die kann man für die Auswahl auch nutzen. Aber dazu kommt später noch ein Beispiel.

Negativauswahl gibt es natürlich auch. Also: Gib mir alles außer diese Spalte:

```
BACups_vieles <- BACups[, -2]
```

Alles außer Spalte 2 ist jetzt dem neuen Datensatz BACups_vieles zugewiesen worden

Ganz toll ist auch die Auswahlmöglichkeit "von a bis x". Das geht mit Doppelpunkt:

```
BACups_x <- BACups[c(10:20),]
```

Schaut euch an, was entstanden ist. Alles klar?

Wie oben, hab ich dem Programm mit c() gesagt, dass die Werte zusammengehören. Mit Doppelpunkt sage ich dann vom 10. bis zum 20. Wert. Da die Zahlen VOR dem Komma sind, erklär ich R, dass ich gern die Zeilen ausgewählt hätte.

Häufig brauchen wir aber nicht irgendwelche 1. Zeile oder 2. Reihe, sondern alle Einträge mit einem bestimmten Wert. z. B. nur die protoappeninen bronzezeitlichen Tassen. Hier führen viele Wege nach Rom:

Daten auswählen

Wie ist also der Mittelwert des Randdurchmessers nur von protoappeninen bronzezeitlichen Tassen? Dafür (wie so für so vieles) gibt es unterschiedliche Wege in R. Schauen wir uns zwei kurz an:

1. *subset*:

Diese Funktion gehört zu base R. Ich erstelle einen neuen Datensatz, der besteht aus dem alten Datensatz, da wo in der Spalte Phase genau (Operator "==") "Protoappenine" steht. Ich hab den Code mal kommentiert:

```
# erstellen eines neuen Datensatzes nur der protoappeninen Tassen
BACups_proto <- subset(BACups, BACups$Phase == "Protoapennine")

# Mittelwert berechnen:
mean(BACups_proto$RD)
```

2. *filter*

Die Filter-Funktion gehört zum sogenannten "tidyverse". Das Tidyverse ist wie ein bestimmter Dialekt von R. Eine Reihe von Paketen folgt diesem Dialekt und diese Pakete arbeiten besonders gut miteinander. Da diese neuen Pakete auch einiges vereinfachen, erfreuen sie sich zunehmender Beliebtheit und wenn man nach Lösungen googelt, findet man Anleitungen, die "tidy" Lösungen erklären. Im Tidyverse gibt es eine Besonderheit, die man kennen sollte: Die sogenannte "*pipe*". Mit dem Befehl %>% wird das Ergebnis einer Zeile in die nächste überführt. Wir benötigen zwei Pakete als nächstes: "dplyr" und "magrittr", die mit *install.packages* installiert werden müssen.

Im Beispiel schicke ich damit den gesamten Datensatz BACups in den Filter, der in der nächsten Zeile beschrieben wird, "filtere" ihn und schick ihn gefiltert weiter in die nächste Zeile, in der ich die Spalte definiere und in die letzte, wo es dann um die Berechnung des Mittelwertes geht:

```
# filter funktion
library(dplyr)
# zur Vereinfachung der Pipe gibt es
library(magrittr)

BACups %>%
  filter(Phase == "Protoapennine") %>%
  use_series(RD) %>%    # das sagt, nimm die Spalte RD, braucht Paket
magrittr
  mean()
```

Wie man sieht, ist der "Kernbefehl" (" Phase == "Protoapennine" ") fast genau gleich wie bei der subset-Funktion. Es sind auch nicht weniger Zeilen Code. Es ist aber eventuell lesbarer. Und wenn ich mir vorstelle, dass ich meine Daten vllt noch nach 20 anderen Variablen filtern möchte, will ich nicht jedesmal einen extra Datensatz erstellen müssen. Manchmal wird es dann schwierig, sich sinnvolle Namen für die Datensätze auszudenken. So sehe ich immer, welche Filter ich genau angewandt habe. Allerdings: Brauche ich diese Datensätze noch für andere Berechnungen, ist subset die bessere Lösung. Oder ich weise das gefilterte einer Variablen zu, das geht auch:

```
BACups_protapp <- BACups %>%
  filter(Phase == "Protoapennine")
```

Option: Funktionen schreiben (falls vor der Pause noch Zeit ist)

Wir haben jetzt schon Funktionen angewandt. Man kann sich in R aber auch Funktionen selber definieren. Die Syntax dafür ist immer gleich:

myfunction <- function(x) { das wird mit x passieren }

Die neue Funktion heißt "myfunction", sie wird auf eine Variable x angewandt und was mit x passiert, wird in den geschweiften Klammern definiert. Stellen wir uns vor, ich möchte eine Funktion für die Berechnung des doppelten Mittelwertes. Die könnte z. B. so aussehen:

```
zwei_m <- function(x){
  2*sum(x)/length(x)
  }
```

Ich rechne zwei mal die Summe von x (d.h. x muss ein Vektor sein) und teile dies durch die Länge des Vektors (Anzahl der Einträge). Ich nenne die Funktion zwei_m.

zwei_m kann ich jetzt anwenden:

```
zwei_m(BACups$RD)
```

Es heißt, wenn man eine Abfolge von Berechnungen mehr als 3mal benutzt, sollte man sie in eine Funktion packen, damit man den Code nicht immer kopieren und einfügen muss. Außerdem kann man so Fehler vermeiden, weil man nur noch die Funktion aufrufen muss und nicht mehr den ganzen Code wiederholen.

Pause 15min

Jetzt geht es um die Erstellung von Graphen. Hier gibt es tausende von Möglichkeiten und es werden immer mehr entwickelt.

In base hat R sehr einfache Funktionen, mit denen man sehr schnell gute funktionale Graphiken erstellen kann, z. B.:

```
hist(BACups$RD)
```

Auch in r-base kann man an den Graphiken noch viele Veränderungen vornehmen (sowohl an der Berechnung der Graphik als auch was Titel, Achsenbeschriftung und ähnliches angeht). Allerdings sind diese Graphiken nicht so elegant und ich selber arbeite mit ggplot:

ggplot

ggplot2 ist ein Paket, dase von Hadley Wickham entwickelt wurde, und viele Funktionen zur Visualisierung von Daten bietet (für eine Übersicht und Inspirationen siehe z. B: http://r-statistics.co/Top50-Ggplot2-Visualizations-MasterList-R-Code.html und https://www.r-graph-gallery.com/). Es folgt einer eigenen Logik, der "Grammatik der Diagramme" und gehört zum Tidyverse. Grundlegend ist eine Art "Layer-Konzept", dass ich mit jeder weiteren Zeile Code ein neues Layer zu dem Diagramm hinzufüge, wie bei der Bildbearbeitung mit Photoshop oder Gimp.

Erarbeiten wir uns das Schritt für Schritt.

3 Dinge gibt es in jedem ggplot, die definiert werden müssen:

- Welche Daten es benutzen soll ggplot(data =),
- welche Art von Diagramm es bauen soll (geom) und
- wie das Diagramm aussehen soll (aes() von aesthetics), damit überhaupt etwas entsteht, also z. B., was auf der x- und der y-Achse abgetragen werden soll.

Alles andere danach sind reine Verschönerungsmaßnahmen. Mit "scales" lassen sich die Achsen und Legenden verändern, mit "theme" Hintergrundfarbe u. ä. (für mehr Infos siehe: https://r-intro.tadaa-data.de/book/visualisierung.html)

Nehmen wir uns ein Beispiel vor und erarbeiten es uns der Reihe nach.

Säulendiagramme

Ein Säulendiagramm eignet sich zur Darstellung nominaler und ordinaler Variablen. Ihr könnt es ja mal mit metrischen Probieren, dann seht ihr schnell, warum das nicht gut ist.

Als erstes müssen wir das Paket ggplot2 installieren (*install.packages*) und aufrufen

```
library(ggplot2)
```

Dann bauen wir ein erstes einfaches Säulendiagramm. Der Befehl für diese Art des Diagramms ist *geom_bar*.

Die Information *data =* kann entweder direkt in die runden Klammern hinter ggplot() geschrieben werden ODER dem geom_bar hinzugefügt.

Wie aber soll das Säulendiagramm (geom_bar) aussehen, welche Spalte des Datensatzes soll genau wie dargestellt werden? Das ist die Information die in *aes()* eingegeben werden muss.

Wir möchten jetzt also ein Säulendiagramm bauen, dass auf der x-Achse die verschiedenen Phasen des BACups-Datensatzes und die Häufigkeiten (wie viele Datensätze aus den verschiedenen Phasen gibt es) auf der y-Achse zeigt:

```
ggplot()+
  geom_bar(data = BACups, aes(x = Phase))
```

Das + am Ende jeder Zeile sagt R, dass der Befehl in der nächsten Zeile weiter geht, ähnlich wie bei der pipe.

Das ist doch schonmal was. Die Information die wir wollen, wird schnell und einfach angezeigt.

Aber schön ist es noch nicht.

Geben wir den Achsen eine andere Beschriftung. Mit dem "labs"-Befehl lassen sich die Achsenbeschriftungen und die Überschriften ändern:

```
ggplot()+
  geom_bar(data = BACups, aes(x = Phase))+
  labs(y = "Häufigkeit",
       title = "Vorkommen der zwei Phasen")
```

Wir können auch die Säulen bunt einfärben. Der Befehl *fill* gibt den Balken unterschiedliche Farben, je nach den Angaben in der Spalte, die ich spezifiziere (hier wieder Phase):

```
ggplot()+
  geom_bar(data = BACups, aes(x = Phase, fill = Phase))+
  labs(y = "Häufigkeit",
       title = "Vorkommen der zwei Phasen")
```

Oder einen anderen Look wählen (ein anderes Thema. Es gibt *theme_classic, theme_grey, theme_minimal* und *theme_bw*):

```
ggplot(data = BACups)+
  geom_bar(aes(x = Phase, fill = Phase))+
  labs(y = "Häufigkeit",
       title = "Vorkommen der zwei Phasen")+
  theme_bw()
```

EXTRA Aufgabe:

Überlegt bitte, was in dem nächsten Code Chunk passiert. Die Hilfe kann mit ?Suchbegriff abgerufen werden.

```
data("EndScrapers")
ggplot(data = EndScrapers)+
  geom_col(aes(x = Site, fill = Width, y = Freq))+
  labs(y = "Häufigkeit",
       title = "Anzahl der Steinartefakte nach Breite und Fundort")+
  theme_bw()
```

Streudiagramme

Bei Streudiagrammen kann ich zwei Variablen gegeneinander plotten.

Wir tragen auf der X- und auf der Y-Achse metrische Daten ab. Das gehört zu den aesthetics-Elementen, deshalb tun wir die Info in die Klammern hinter *aes()*:

```
ggplot(data = BACups)+
  geom_point(aes(x = RD, y = ND))
```

Jetzt können wir damit wieder die Dinge tun, die wir mit dem Balkendiagramm gemacht hatten, also die Achsen beschriften, einen Titel vergeben und den Style ändern:

```
ggplot(data = BACups)+
  geom_point(aes(x = RD, y = ND)) +
  labs(x =" Randdurchmesser",
       y ="Nackendurchmesser",
       title = "Rand- und Nackendurchmesser im Verhältnis zueinander")+
  theme_classic()
```

Was kann man noch tolles machen? Wie wäre es mit einer Linearen Regression?

Dafür fügen wir dem bestehenden Plot ein Layer hinzu. Der Befehl für Regressionslinien ist geom_smooth und wir benutzen die Methode “linear model” (lineare Regression), abgekürzt “lm” (es gibt noch mehr…). Mit *se* kann man den “standard error” hinzufügen (*se = TRUE*) oder nicht (*se = FALSE*).

```
ggplot(data = BACups)+
  geom_point(aes(x = RD, y = ND)) +
  geom_smooth(aes(x = RD, y = ND), method = "lm",
              se = TRUE)+
  labs(x =" Randdurchmesser",
       y ="Nackendurchmesser",
```

```
       title = "Rand- und Nackendurchmesser im Verhältnis zueinander")+
  theme_classic()
```

Was geht noch? Die Form und Farbe der Punkte von einer Variablen bestimmen lassen!

Welches Merkmal, das ich in der Tabelle als Spalte aufgenommen habe die Form der Punkte bestimmt lege ich mit *shape* fest, die Farbe mit *color*.

```
ggplot(data = BACups)+
  geom_point(aes(x = H, y = SD, shape = Phase, color = Phase)) +
  labs(x =" Höhe des Gefäßes",
       y ="Schulterdurchmesser",
       title = "Höhe des Gefäßes im Verhältnis zum Schulterdurchmesser")+
  theme_bw()
```

Form und Farbe kann man natürlich auch von unterschiedlichen Parametern bestimmen lassen. Da diese Eigenschaften jedoch nominaler Art sein sollten und wir keinen zweiten nominale Variable in dem BACups-Datensatz haben, benutzen wir doch mal einen anderen.

Die Snodgrass-Daten beinhalten "Information on the size, location and contents of 91 house pits at the Snodgrass site which was occupied between about CE 1325-1420". Ich schlag vor, ihr ladet ihn und schaut ihn euch kurz an:

```
data("Snodgrass")
View(Snodgrass)

ggplot(data = Snodgrass)+
  geom_point(aes(x = Width, y = Length, shape = Segment, color = Inside))+
  labs(x =" Breite des Hauses",
       y ="Länge des Hauses",
       title = "Häuser in Snodgrass")+
  theme_bw()
```

Hmmmhh, interessant. Aber ich vermute, der normale Leser des Diagramms kann nicht erkennen, was "Inside" für eine Information beinhaltet. Kann man vllt die Beschriftung der Legende ändern?

Man kann!!

```
ggplot(data = Snodgrass)+
  geom_point(aes(x = Width, y = Length, shape = Segment, color = Inside))+
  labs(x = " Breite des Hauses",
       y = "Länge des Hauses",
       title = "Häuser in Snodgrass")+
  theme_bw()+
 scale_colour_discrete(name = "Innerhalb der Mauer oder nicht",
                           breaks = c("Inside","Outside"),
                           labels = c("innerhalb", "außerhalb")) +
scale_shape_discrete(name  ="Grabungsareal",
                           breaks = c("1","2","3"),
                           labels = c("Areal 1", "Areal 2", "Areal 3"))
```

Was bedeutet das alles?

Mit *scale_colour_discrete* kann ich Legenden (*scales*) verändern, die mit *color* innerhalb des aesthetics-Bereichs meines Codes für die Graphik definiert werden und die DISKRET sind (also v.a. nominale / ordinale Daten).

Hier benenne ich den Legendentitel mit *name =* um.

breaks bezeichnet die Werte in meiner Spalte, die dann mit den *labels* in der nächsten Zeile umbenannt werden.

Das gleiche kann ich mit der Legende für die FORM der Punkte machen: *scale_shape_discrete*. Voll gut!

Facettierung!

Jetzt wird es nochmal richtig cool.

Mit der Facettierung kann man ein oder zwei weitere Variablen auswählen, die den Datensatz unterteilen und in unterschiedlichen, aber gleich skalierten!, Graphiken darstellen. Es gibt zwei Möglichkeiten: *facet_grid()* und *facet_wrap()*. In beiden kann man ein bis zwei Variablen definieren, mit denen die Graphiken unterteilt werden. Während bei *facet_grid* immer alle Graphiken angezeigt werden, auch wenn sie leer sind, werden bei *facet_wrap* nur die angezeigt, in denen Daten vorkommen.

Vielleicht ein kleines Bsp: Im Snodgrass-Datensatz wollen wir die Breite und Länge der Häuser darstellen. Wir unterteilen die Plots aber danach, ob die Häuser innerhalb oder außerhalb der Mauer sind und danach, wie viele Figürinchen (effigies) in den Häusern gefunden wurden. Die Variablen, nach denen facettiert wird, werden mit ~ voneinander getrennt:

```
ggplot(data = Snodgrass)+
  geom_point(aes(x = Width, y = Length))+
  facet_grid(Inside~Effigies)
```

Wie man sieht, gibt es einige Plots, die leer sind. *facet_wrap* bricht die rigide Gitterstruktur auf und zeigt nur die plots, die auch Daten beinhalten:

```
ggplot(data = Snodgrass)+
  geom_point(aes(x = Width, y = Length))+
  facet_wrap(Inside~Effigies)
```

Sehr praktisch ist auch die Möglichkeit, hier die Variablen noch genauer zu spezifizieren. z.B. möchte ich gern nicht die genaue Anzahl von Figurinen, sondern nur ob es überhaupt welche in einem Haus gibt oder nicht der Information gegenüber stellen, ob sich das Haus innerhalb oder außerhalb der Mauer befindet:

```
ggplot(data = Snodgrass)+
  geom_point(aes(x = Width, y = Length))+
  facet_grid(Inside~Effigies > 0)
```

Weiterführendes zu ggplot

ggplot hat noch viel viel mehr Möglichkeiten. Um das vorgeführt zu bekommen, empfehle ich den Blogpost hier zu lesen, der zeigt, wie sich so eine Visualisierung entwickeln kann und am Ende richtig richtig gut aussieht:
https://cedricscherer.netlify.com/2019/05/17/the-evolution-of-a-ggplot-ep.-1/

Hilfen, um mit ggplot zurechtzukommen sind:

- die Schummelzettel: https://www.rstudio.com/wp-content/uploads/2015/06/ggplot2-german.pdf
- dieses R-Intro-Buch: https://r-intro.tadaa-data.de/book/visualisierung.html
- das englische R-Cookbook: http://www.cookbook-r.com/Graphs/

eigene Daten einladen

Je nachdem, wie die eigenen Daten gespeichert sind, benötigt man unterschiedliche Funktionen in R. Was base-r am einfachsten kann, ist eine csv-Datei einladen. Der Befehl dafür heißt "read.csv2" (eine Weiterentwicklung von read.csv). Mit diesem Befehl können wir auch spezifizieren, mit welchem Zeichen die Spalten voneinander getrennt wurden. Im Beispiel gebe ich an, dass eine Semikolon-getrennte Tabelle ist. Außerdem ist es ein deutscher Datensatz und der Dezimaltrenner ist das Komma:

```
mydata <- read.csv2("Pfad/zur/Datei/meineDatei.csv", sep = ";", dec = ",")
```

Für Excel-Daten muss man ein eigenes Paket installieren. Ich empfehle "openxlsx". Hier kann ich z. B. spezifizieren, welches sheet in der Excel-Arbeitsmappe als Tabelle eingelesen werden soll:

```
install.packages("openxlsx")
library(openxlsx)

mydata <- read.xlsx("Pfad/zur/Datei/meineDatei.xlsx", sheet = 1)
```

weitere spannende Dinge in R:

räumliche Analysen

Mit R kann man auch alle möglichen räumlichen Analysen machen. Man kann shape-files und raster-Daten bearbeiten. Ein kleines Beispiel, wie man eine einfache Karte plotten kann, lässt sich an dem Arnhofen-Datensatz zeigen

Man benötigt das Paket "spatstat". In dem Datensatz gibt es das Arbeitsgebiet (window) als Polygon, welches mit der Funktion *owin* zu dieser Art von Objekt übertragen wird. Ein point process pattern object besteht aus den x- und y-Daten sowie diesem window:

```r
library(spatstat)
data("Arnhofen")

# generate observation window object; note the polygonal outline.
arnwin <- owin(poly=Arnhofen$window[, 1:2])

# generate point process pattern object from points and owin object
app <- ppp(Arnhofen$points$x, Arnhofen$points$y, arnwin)

plot(app)
```

Die Daten sind so für geostatistische Analysen vorbereitet. Zum Beispiel für eine Kerndichteschätzung:

```r
plot(density.ppp(app))
```

- das wichtigste Buch in Zusammenhang mit spatstat: Baddeley, A., Rubak, E. and Turner, R. (2015) Spatial Point Patterns: Methodology and Applications with R. Chapman and Hall/CRC Press.
- weitere Pakete für räumliche Analysen sind z. B.: sp, sf, maptools

R Markdown

Markdown ist eine Auszeichnungssprache, die so einfach ist, dass sie menschenlesbar bleibt. Dokumente in Markdown können mit unterschiedlichen Hilfsprogrammen z. B. in PDFs, html-Seiten oder docx-Dateien umgewandelt werden. In einem RMardown-Dokument kann man Text und Code in sogenannten "Code-Chunks" schreiben und bei der Umwandlung wird der Code ausgeführt, d. h. es erscheinen die Graphiken. Es ist sehr praktisch, wenn man alle wichtigen Informationen zusammen haben möchte. In RMarkdown lassen sich auch Zitationen automatisieren (über zotero, csl und bib-Datei). Es gibt neben Textdokumenten auch die Möglichkeit, websites, Präsentationen und Poster zu erstellen.

- Pakete: rmarkdown, knitr
- The Definitive Guide: https://bookdown.org/yihui/rmarkdown/
- offizielle website: https://rmarkdown.rstudio.com/index.html

rrtools:

Genau wie man Pakete mit Funktionen herunterladen kann, lassen sich auch eigene Projekte als R-Paket speichern. Damit man Text, selbstgeschriebene Funktionen und Daten in einem Paket zusammen hat, erstellt rrtools (reproducible research tools) eine Ordnerstruktur und legt ein template-paper (als RMardown-Dokument) an. Diese Paket kann man a) anderen Leuten zur Verfügung stellen, die damit alles haben, um die Forschung zu reproduzieren und/oder b) hervorragend mit git und github verknüpfen, um eine ordentliche Versionskontrolle zu gewährleisten.

- Pakete: rrtools, rrtools.addin
- https://github.com/benmarwick/rrtools

- https://github.com/nevrome/rrtools.addin (GUI für rrtools)

Hilfestellungen

Was mach ich, wenn ich eine Fehlermeldung bekomme?

1. Fehlermeldung lesen und überlegen, ob sie einen Hinweis gibt, was das Problem sein könnte
- zwei Seiten mit Erklärungen zu "den häufigsten Fehlern": https://bookdown.org/chesterismay/rbasics/6-errors.html und http://medeiros.ng/2016/06/five-common-mistakes-in-r-programming/
2. Checken, dass man sich auch ja nicht vertipppt hat: Groß- und Kleinschreibung, Kommasetzung, Klammersetzung, einfach ein Buchstabe zu viel?
3. Manchmal ist es hilfreich, noch einmal nachzuschauen, ob man die Syntax auch richtig hat. Da hilft z. B. die Hilfe-Funktion in R: ?Suchbegriff oder rechts unten im Fenster unter dem Reiter "Help".
4. Ist vielleicht "weiter oben im Code" ein Fehler passiert, der dann als Folgefehler erst an dieser Stelle auffällt?
5. googeln: Online gibt es viele Leute, die das gleiche Problem schon hatten. Beim googeln hilft es, den Paket- und den Funktionsnamen "mitzugoogeln". Da "R" als Buchstabe sehr unspezifisch ist, gibt es außerdem https://rseek.org/ , eine Art "personalisiertes Google".

Es lassen sich online diverse Tutorials und Anleitungen finden. Hier eine Auswahl:

- Ben Marwicks Buch "How to do archaeological Science using R": https://benmarwick.github.io/How-To-Do-Archaeological-Science-Using-R/
- Rafael Irizarrys Buch "Introduction to Data Science. Data Analysis and Prediction Algorithms with R": https://rafalab.github.io/dsbook/getting-started.html
- Roger Pengs Buch: https://bookdown.org/rdpeng/rprogdatascience/
- Nathaniel Phillips "YaRrr! The Pirates Guide to R": https://ndphillips.github.io/piratesguide.html
- das deutsche Wikibook zu R: https://de.wikibooks.org/wiki/GNU_R und
- das englische R-Cookbook: http://www.cookbook-r.com/
- Video-Tutorials von ISAAK: https://isaakiel.github.io/screencasts.html
- …
- Dieses Tutorial ist online unter: https://github.com/SCSchmidt/RworkshopCAAde