Skript R Workshop @ CAA de 2019

Sophie C. Schmidt

4 September 2019

## Einführung

### Programmieren

Programmieren ist so schwer nicht. Ich ersetze das “klicken auf einen button” mit “Text, der genau sagt, was getan werden soll”. Man nennt die Umsetzungsform jedoch nicht umsonst “ProgrammierSPRACHE”. Wie bei einer echten Sprache, muss man sich an gewisse Regeln halten. Im Gegensatz zu normalen menschlichen Sprachen, verzeiht einem der Computer jedoch kleine Fehler nicht. Das ist eigentlich die größte Schwierigkeit: Exakt korrekt zu schreiben. Keine Kommafehler, keine Klammer vergessen, Groß- und Kleinschreibung beachten…

Analysen in Code (egal ob R oder Python oder eine andere Skript- oder Programmiersprache) haben den großen Vorteil, dass sie deutlich leichter reproduzierbar und replizierbar sind als welche, die in Maus-gesteuerter Software erstellt werden (siehe alles von Ben Marwick). Ich kann den Code jemand anderem geben und er/sie kann bis ins letzte Detail nachvollziehen, was berechnet wurde. Benutze ich keinen Code brauche ich dafür Beschreibungen, die eventuell ausarten könnten.

Wenn sich in meinem Datensatz eine Kleinigkeit ändert (es könnte ja mal sein, dass eventuell ein Fehler passiert war), kann ich die gesamte Analyse sehr sehr schnell einfach wieder durchführen – der Code ist ja noch da, und der bleibt gleich. Wiederhole ich nach einem Jahr eine Analyse mit anderen Daten, geht das sehr sehr schnell – der Code kann einfach wieder benutzt werden.

Das ist ein Riesenvorteil.

### Warum R

R wurde 1992 von Ross Ihaka und Robert Gentleman, zwei Statistikern, in Auckland als open und free source - Alternative zu der Sprache “S” entwickelt. Da sie auch plattformunabhängig ist, ist sie immer und überall benutzbar.

R wurde von Statistikern entwickelt und hat deshalb insbesondere für statistische Analysen eine große Menge an Paketen und definierten Funktionen, die von der Gemeinschaft beständig weiterentwickelt und erweitert werden. Sie werden auf dem Comprehensive R Archive Network (CRAN) zur Verfügung gestellt. Damit findet man eigentlich für jedes Problem eine Lösung. Inzwischen kann man mir R ein GIS ersetzen, es lassen sich interaktive und animierte Graphiken erstellen, Websites und Präsentationsfolien bauen und sicherlich noch mehr, das ich nicht kenne…

Online gibt es eine Menge Ressourcen und Hilfestellungen, die einem die Arbeit mit R erleichtern. Zugegebenermaßen ist der Einstieg nicht ganz leicht, aber der Mehraufwand lohnt sich! Es gibt eine wachsende Gemeinschaft von Archäologen, die es nutzen.

R selber hat keine schöne Benutzeroberfläche, sondern wird von der Konsole aus “gesteuert”. Dafür benutzern wir Rstudio (es gibt noch andere wie RCommander, RGui), das einem Skript, Konsole, Programmierumgebung und noch weiteres übersichtlich in Fenstern anordnet.

## 1. Einführung in die Grundlagen

R ist eine objektorientierte Sprache. Das heißt, alles ist in R entweder ein Objekt oder eine Funktion. Eine Funktion “macht” immer etwas mit einem Objekt. Wie im Mathe-Unterricht: f(x) = 2\*x rechnet für jedes x den zugehörigen Wert (y) aus, der genau das Doppelte von x ist. Nur können die Funktionen in R deutlich komplizierter werden…

Die Funktionen sind in Paketen gespeichert, die sich auf einander beziehen: Wenn in Paket A eine Funktion f(x) liegt, in der die Funktion g(x) aus Paket B benutzt wird, besteht eine Abhängigkeit (*dependency*) von Paket A zu Paket B. Installiere ich Paket A, wird in der Regel automatisch Paket B mitinstalliert, damit ich die Funktion f(x) auch wirklich benutzen kann. Manchmal kommen aber trotzdem Fehlermeldungen wie “error: could not find function”, dann kann es sein, dass eine *dependency* nicht mitinstalliert wurde ODER nicht geladen wurde. Denn: Wenn wir Funktionen aus einem Paket brauchen, müssen wir es installieren und jedesmal, wenn wir es benutzen, am Anfang einer R-Sitzung laden (mit der “library” oder “require”-Funktion, Beispiele später). Die wichtigsten Funktionen sind in R “base” vorinstalliert. Manchmal haben Funktionen in unterschiedlichen Paketen den gleichen Namen. Dann gibt R eine Warnung aus, “objects are masked”, d.h. die Funktionen des neu geladenen Pakets “überschreiben” die alten. Nicht irritieren lassen, meistens interessiert uns das nicht.

Man kann in R auch selber Funktionen schreiben, das führt für den Workshop aber zu weit.

Funktionen erkennt man daran, dass hinter dem Funktionsbefehl oder -namen in runden Klammern die Objekte stehen, auf die die Funktion angewandt wird. z. B.:

mean(x)

–> mean ist der Funktionsname, auf x wird die Funktion ausgeführt.

Was könnte x, können Objekte sein? (Tatsächlich können Funktionen auch als Objekte behandelt werden)

Wir betrachten hier nur die wichtigsten Typen von Objekten: Skalare, Vektoren und Dataframes. Diese werden in R in der Regel durch Variablen kodiert. Ein Skalar ist nur *ein einziger* Wert. Es kann sich um eine Zahl handeln, als *integer* also Ganzzahl oder *numeric* als Kommazahl. Ein Skalar kann aber auch eine Abfolge von Buchstaben sein, *character* genannt, also ein Wort oder ein Kürzel.

Vektoren sind eine Reihe von Skalaren gleichen Typs. Also eine Reihe von *integer* oder mehrere *character*-Einträge hintereinander. Wenn aber in einem Vektor sowohl Zahlen als auch Texteintragungen auftauchen, werden auch die Zahlen als Text gespeichert und man kann nicht mehr mit ihnen rechnen. Einen Vektor kann man sich auch als Spalte einer Tabelle vorstellen, wobei die Spalteneinträge immer die gleiche Datentypen beinhalten müssen.

Mehrere Vektoren können zu einem *Dataframe* zusammengefasst werden. Ein Dataframe ist wie eine Tabelle: Die unterschiedlichen Spalten können unterschiedliche Datentypen beinhalten und besitzen Spaltennamen. Die Zeilen werden als “row.names” entweder gezählt oder tatsächlich benannt.

Beispiele kommen gleich!

Noch eine Kleinigkeit vorneweg:

R ist case-sensitive! Groß- und Kleinschreibung sind also stets zu beachten. Für Objektnamen (zum Beispiel von Funktionen und Variablen) sind neben den alphanumerischen Zeichen auch der Punkt und der Unterstrich erlaubt. Objektnamen mit Unterstrich sind allerdings eher selten anzutreffen, häufiger wird der Punkt benutzt, um Objektbezeichner zu strukturieren. Leerzeichen sind nie eine gute Idee!

## Ins kalte Wasser!

Rstudio öffnen und Fenster anschauen: oben rechts: Environment = Programmierumgebung, history = letzte Befehle; unten rechts: files = Ordner, in dem ich gerade arbeite, plots = Reiter unter dem Bilder gezeigt werden, packages = welche Pakete sind installiert und geladen, help = Hilfe (immer gut!) unten: Console und Terminal.

Als eiserne Regel benutzt man immer ein Skript, um Code zu schreiben und nicht direkt die Konsole. Der Vorteil ist: Alles im Skript kann ich speichern (und sollte ich auch möglichst häufig, Strg+S is your best friend), was ich in die Konsole tippe, wird nicht gespeichert. Also Skript anlegen und abspeichern ist das erste was wir machen. Dafür gibt es links oben ein kleines Symbol (weißes Blatt mit grün umrandeten +).

Mit dem # Hashtag kommentiere ich Text aus. D. h. der Text wird nicht als Code behandelt. Sehr sehr wichtig ist das, weil ich damit meinen Code kommentieren kann. Kommentare erleichtern einem das Leben massiv. In der Regel hat man nämlich nach spätestens einer Woche vergessen, was der Code tun konnte und sollte und wenn dann daneben ein hilfreiches Kommentar zu finden ist, erinnert man sich wieder.

Deswegen sollten wir alle in die oberste Zeile schreiben, was das für eine Datei ist und wer sie erstellt hat.

Die Skriptdatei ist eine einfache Textdatei, die die Dateiendung .R besitzen. Das Format .RData (oder kurz .Rda) wird verwendet, um ein R-Objekt, beispielsweise einen Datensatz, oder eine Kollektion von R-Objekten, also Daten und Funktionen, im R internen binären serialisierten Format abzuspeichern, wobei diese Dateien zusätzlich standard-komprimiert sind. Die gesamte Arbeitsumgebung kann so ebenfalls als .RData-Datei gespeichert werden.

Der in einem Skript geschriebene Code wird nicht automatisch sofort ausgeführt, sondern die Ausführung muss beauftragt werden, in dem man ihn mit Strg + ENTER zur Konsole sendet. Was in die Konsole eingetippt wird, wird sofort durch ENTER ausgeführt.

### Taschenrechner

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

3 + 2

## [1] 5

5 - 7

## [1] -2

5 \* 2

## [1] 10

100 / 10

## [1] 10

Oder auch:

3\*(4+2)

## [1] 18

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

Da diese beiden Werte 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 diese beiden in einen Vektor zusammengefügt:

ab <- c(a,b)

Unter “Environment” oben rechts in Rstudio sind 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 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

## starting httpd help server ... done

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.

## 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

## [1] Subapennine Subapennine Subapennine Subapennine Subapennine   
## [6] Subapennine Subapennine Subapennine Subapennine Subapennine   
## [11] Subapennine Subapennine Subapennine Subapennine Subapennine   
## [16] Subapennine Subapennine Subapennine Subapennine Subapennine   
## [21] Subapennine Subapennine Subapennine Subapennine Subapennine   
## [26] Subapennine Subapennine Subapennine Subapennine Subapennine   
## [31] Subapennine Subapennine Subapennine Subapennine Subapennine   
## [36] Subapennine Subapennine Subapennine Subapennine Subapennine   
## [41] Protoapennine Protoapennine Protoapennine Protoapennine Protoapennine  
## [46] Protoapennine Protoapennine Protoapennine Protoapennine Protoapennine  
## [51] Protoapennine Protoapennine Protoapennine Protoapennine Protoapennine  
## [56] Protoapennine Protoapennine Protoapennine Protoapennine Protoapennine  
## Levels: Protoapennine Subapennine

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: Protoappenin und Subappenin.

Tun wir das gleiche für einen numerischen Vektor:

BACups$RD

## [1] 11.1 9.5 20.8 19.5 15.5 11.7 10.8 15.0 18.5 11.0 9.0 9.0 12.1 10.7  
## [15] 19.5 20.0 18.0 19.0 24.0 15.2 13.2 29.5 12.0 10.1 22.0 29.0 9.8 8.7  
## [29] 19.1 10.0 10.5 9.0 18.5 19.0 18.3 18.5 17.2 9.5 9.0 19.5 11.0 15.0  
## [43] 8.5 19.0 10.1 11.5 9.0 10.5 11.0 19.0 8.8 9.5 13.3 8.0 8.0 15.8  
## [57] 8.4 12.9 13.0 6.6

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)

## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 6.600 9.725 12.050 14.020 18.500 29.500

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 insbesondere 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:

# erstellen eines neuen Datensatzes nur der protoappeninen Tassen  
BACups\_proto <- subset(BACups, BACups$Phase == "Protoapennine")  
  
# Mittelwert berechnen:  
mean(BACups\_proto$RD)

## [1] 11.445

1. 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 %>% (das ist das Rohr [manche übersetzen es mit Pfeife, wieso auch immer, Rohr ist viel sinniger]) wird das Ergebnis einer Zeile in die nächste überführt.

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:

library(tidyverse)

## -- Attaching packages ------------------------------------------------------------------------------------------------------ tidyverse 1.2.1 --

## v ggplot2 3.2.1 v purrr 0.3.2  
## v tibble 2.1.3 v dplyr 0.8.3  
## v tidyr 0.8.3 v stringr 1.4.0  
## v readr 1.3.1 v forcats 0.4.0

## -- Conflicts --------------------------------------------------------------------------------------------------------- tidyverse\_conflicts() --  
## x dplyr::filter() masks stats::filter()  
## x dplyr::lag() masks stats::lag()

# zur Vereinfachung der Pipe gibt es   
library(magrittr)

##   
## Attaching package: 'magrittr'

## The following object is masked from 'package:purrr':  
##   
## set\_names

## The following object is masked from 'package:tidyr':  
##   
## extract

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

## [1] 11.445

## ohne magrittr   
#BACups %>%  
# filter(Phase == "Protoapennine") %>%  
# `$`("RD") %>%  
# 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. Brauche ich diese Datensätze aber vllt noch für andere Berechnungen, ist subset die bessere Lösung.

Beides funktioniert gleichermaßen gut.

# Pause 15min

## ggplot

ggplot wurde von Hadley Wickham entwickelt, ist ein Paket mit vielen Funktionen zur Visualisierung von Daten und folgt einer “Grammatik der Diagramme”.

Erarbeiten wir uns das Schritt für Schritt.

Wir müssen dem Programm sagen: Welche Daten es benutzen soll (data = ), welche Art von Diagramm es bauen soll (geom\_xxx) und wie das Diagramm aussehen soll (aes von aesthetics), damit überhaupt etwas entsteht.

Alles andere danach sind reine Verschönerungsmaßnahmen. ;-)

## ggplot- Logik!

Zur Wiederholung empfehle ich folgenden Link: <https://r-intro.tadaa-data.de/book/visualisierung.html> Ja, ich bin zu faul um das abzutippen. Gleichzeitig lernt ihr so aber auch eine weitere gute Ressource kennen, die auf R auf deutsch erklärt und frei zugänglich ist.

Auf zu den Diagrammtypen jetzt:

Jetzt aber zu den Diagrammen. Es geht im Folgenden um

* Säulendiagramme
* Liniendiagramme
* Dichtediagramme
* Facettierungen
* geom\_raster
* ggplot-Hilfen

# ein Säulendiagramm

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.

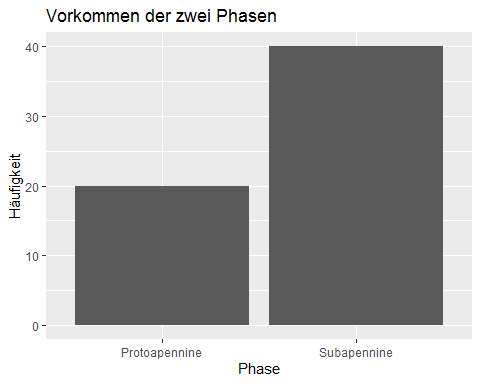
library(ggplot2)  
ggplot(data = BACups)+ #data = bezieht sich auf den Datensatz mit dem ich arbeite, meist ein dataframe  
 geom\_bar(aes(x = Phase)) #geom\_bar bedeutet, ich hätte gern ein Balkendiagramm, aesthetic: ich will, dass auf der X-Achse die Phasen abgetragen werden



Jetzt kann man viele Dinge verschönern

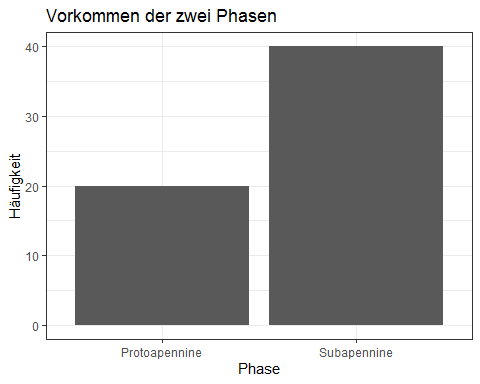
1. B. Den Achsen eine andere Beschriftung geben:

ggplot(data = BACups)+   
 geom\_bar(aes(x = Phase))+  
 labs(y = "Häufigkeit", #der y-Achse einen neuen Namen geben  
 title = "Vorkommen der zwei Phasen") # dem ganzen Plot eine Überschrift geben



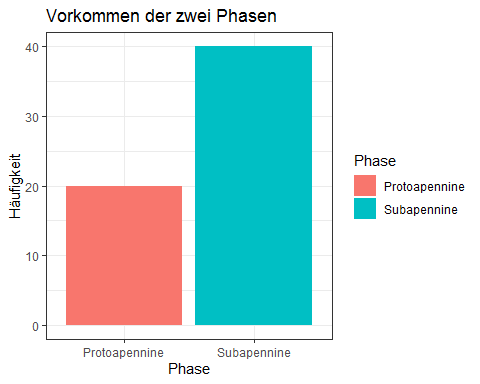
Oder einen anderen Look wählen (ein anderes Thema):

ggplot(data = BACups)+   
 geom\_bar(aes(x = Phase))+   
 labs(y = "Häufigkeit",  
 title = "Vorkommen der zwei Phasen")+  
 theme\_bw() #theme\_classic, theme\_grey, theme\_minimal



Oder die Säulen bunt einfärben:

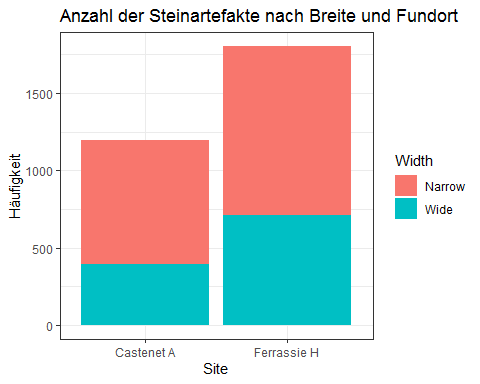
ggplot(data = BACups)+ #data = bezieht sich auf den Datensatz mit dem ich arbeite, meist ein dataframe  
 geom\_bar(aes(x = Phase, fill = Phase))+ # fill gibt den Balken unterschiedliche Farben, je nach den Angaben in der Spalte Phase  
 labs(y = "Häufigkeit",  
 title = "Vorkommen der zwei Phasen")+  
 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()



## Liniendiagramme!

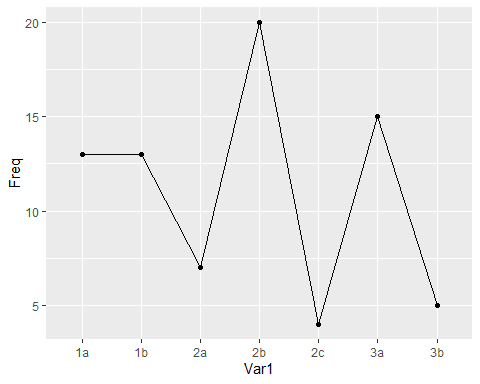
Liniendiagramme sind nur sinnvoll, wenn auf der X-Achse eine Abfolge erstellt werden kann, also mindestens ordinale Daten abgetragen werden können. Für zeitliche Entwicklungen eignen sie sich super, aber es muss beachtet werden, dass jedem X-Wert nur ein Y-Wert zurgeordnet werden darf. Das ist der Grund, warum wir in dem Bsp erst eine neue Tabelle bauen müssen.

Im Bornholmer Datensatz gibt es eine schöne Abfolge von verschiedenen Perioden und ich will einfach nur darstellen, wie viele Fundstellen es pro Periode gibt. Vielleicht benutze ich das für eine Bevölkerungsdichtenrekonstruktion, wer weiß.

Der table-Befehl zählt, wie häufig eine Variable mit einer anderen vorkommt. Da ich nur eine Variable angeben, zählt er einfach, wie häufig diese vorkommt. Da ggplot nicht mit dem Format table arbeiten kann, wandeln wir bh\_table noch in einen data frame um.

Schaut euch die Tabelle bh\_table einmal mit View an, dann versteht ihr, warum ich im Liniendiagramm die Variablen so gewählt habe, wie sie da stehen.

data("Bornholm")  
  
bh\_table <- table(Bornholm$Period) # Häufigkeiten zählen   
bh\_table <- as.data.frame(bh\_table) # als Dataframe überspeichern  
  
  
ggplot(data = bh\_table)+  
 geom\_point(aes(x = Var1, y = Freq))+  
 geom\_line(aes(x = Var1, y = Freq, group = 1)) # group = 1 ist für geom\_line wichtig, weil es sonst Daten gruppieren soll. Ich habe die Gruppierung aber schon vorher mit dem table-Befehl erledigt.



# ein Punktdiagramm über einem Liniendiagramm "markiert" die Stellen, wo ich Datenpunkte habe  
# wie ihr seht, kann man unterschiedliche Diagrammtypen übereinander plotten!

Aufgabe: Das ist zwar ein ordentliches Diagramm, aber die Beschriftung ist eher hässlich. Bitte kramt den Code der letzten Sitzung raus und beschriftet die Achsen angemessen.

Viel Erfolg!

### Liniendiagramm mit mehreren Linien und der notwendige Umstellungsspaß mit den Daten

Häufig will ich ja gar nicht nur eine Linie darstellen, sondern mehrere Verläufe vergleichen. Selbstverständlich geht das auch mit R. Mit R geht aaaaalles.

Ich muss allerdings erst ein bisschen die Daten in eine Form bringen, mit der ich arbeiten kann. Ich möchte jetzt gern wissen, wie häufig welche Fundtypen in welcher Periode auftauchen. Die Fundtypen sind die ganzen komischen Kürzel im Datensatz Bornholm.

Was ich kreiieren möchte, ist eine Tabelle mit den Spalten: Periode, Fundtypus, Häufigkeit in der Periode. Site und Number interessieren mich nicht mehr. Ich wandle ein “breites” Datenformat in ein “langes” um. Allgemein geht ggplot lieber mit langen Datensätzen um als mit breiten.

Was meine ich damit? Was passiert hier?

Vielleicht erklärt dieser Blogpost mehr: <http://archaeoinformatics.net/r-seperate-gather-spread/> Das Vorgehen “gather” brauchen wir hier auch: Es ist etwas komplizierter als die Tabelle vorhin, deswegen benutzen wir die beiden neuen Pakete tidyr und dplyr. Beide gehören zu einer R-Philosophie, die sich tidyverse nennt. Das ist eine Reihe von Paketen, die gut miteinander zurecht kommen und ähnliche Syntaxen verwenden. ggplot gehört auch dazu.

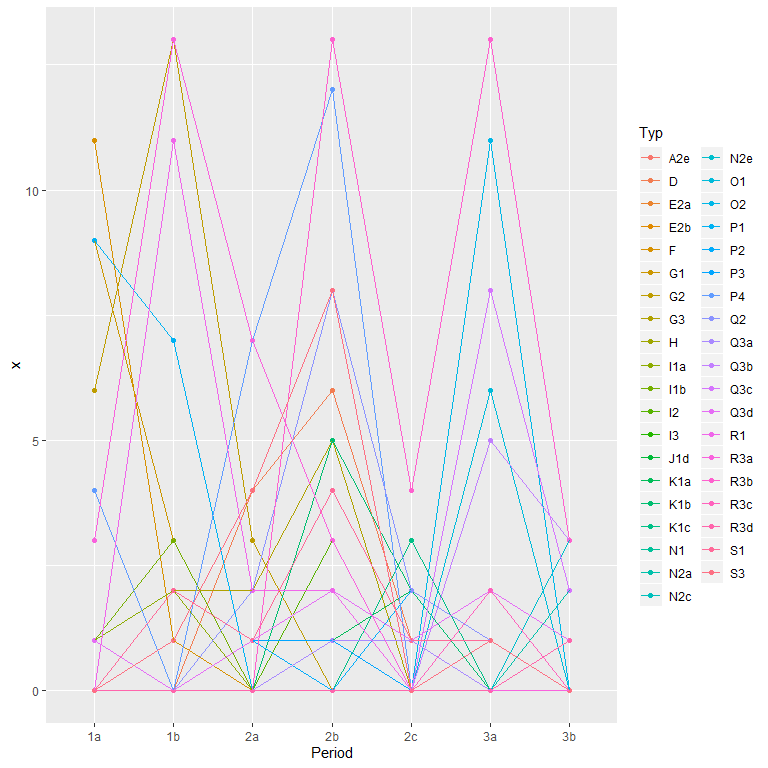
Sie benutzen ein neues Zeichen, das in R base keine Rolle spielt: Die “Pipe” %>%

Pipes sind auch aus anderen Programmiersprachen bekannt. Sie sagen eigentlich nur “was ich gerade in dieser Zeile gemacht habe, übertrage ich auch in die nächste” und man spart sich eine Reihe von “Zwischensicherungen” in Variablen.

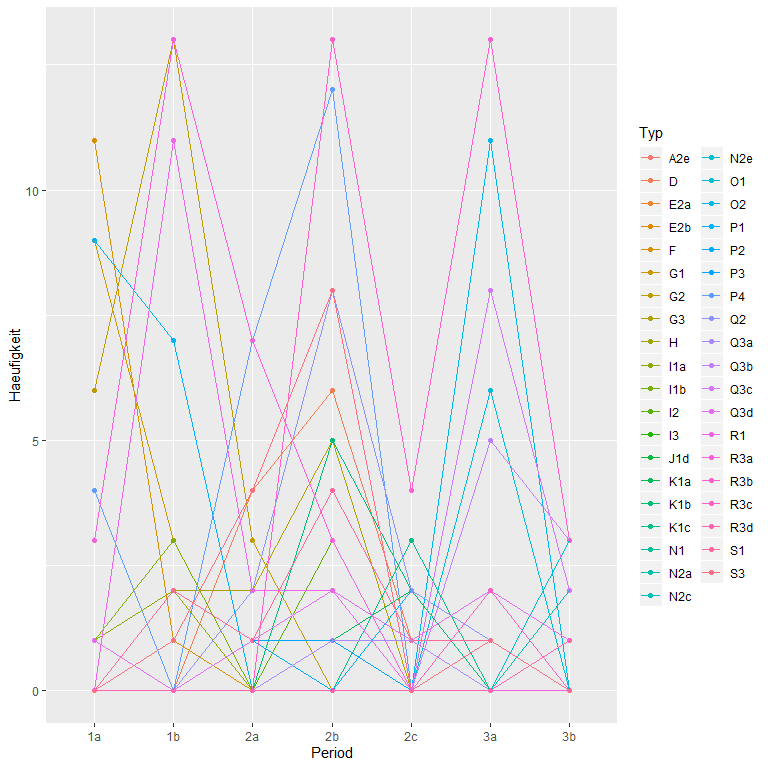
Als Bsp mach ich diese Umformung der Daten auf beide Weisen:

Zuerst oldschool:

Bornh1 <- Bornholm[, -c(1:2)] #mit eckigen Klammern kann ich aus dem Datensatz Bornholm bestimmte Spalten entfernen: - Spalte 1 bis 2  
  
Bornh2 <- gather(Bornh1, key = "Typ", value = "Haeufigkeit", "N2c":"A2e") # das ist der Code der die Umformung vornimmt.  
#Schaut euch Bornh2 einmal an, damit ihr versteht, was passiert. Manchmal gibt es jetzt gleiche Periode und Typ mit unterschiedlichen Häufigkeiten. Das muss noch einmal zusammengefasst werden: aggregate!  
  
Bornh3 <- aggregate(Bornh2$Haeufigkeit, by = list(Typ = Bornh2$Typ, Period = Bornh2$Period), FUN = sum)  
# jetzt ist dummerweise Haeufigkeit in x umbenannt worden  
  
ggplot(data = Bornh3)+  
 geom\_point(aes(x = Period, y = x, color = Typ))+  
 geom\_line(aes(x = Period, y = x, color = Typ, group = Typ)) # group bestimmt, welche Punkte verbunden werden

 Jetzt das gleiche in tidy code. Wie ihr seht, liegt der Unterschied v.a. darin, dass ich nicht dauernd neue Variablen benenne:

Bornholm %>%  
 select(-Site, -Number) %>%  
 gather(key = "Typ", value = "x", "N2c":"A2e") %>%  
 group\_by(Typ, Period)%>% #ich gruppiere mein Daten, wie bei aggregate  
 summarize(Haeufigkeit = sum(x))%>% #ich summiere sie jetzt und geb den Namen Haeufigkeit für die Spalte  
 ggplot()+  
 geom\_point(aes(x = Period, y = Haeufigkeit, color = Typ))+  
 geom\_line(aes(x = Period, y = Haeufigkeit, color = Typ, group = Typ))



# Nach tidyverse-Logik ist das leichter zu lesen. Was denkt ihr?

Aber oje oje, weg vom Code, hin zum Plot: Was ist denn da passiert?

Schöne Idee war das ja, mit den Typenhäufigkeiten nach Periode, aber…

Wir erkennen ein Problem: Zu viele Informationen auf einmal sind keine gute Idee.

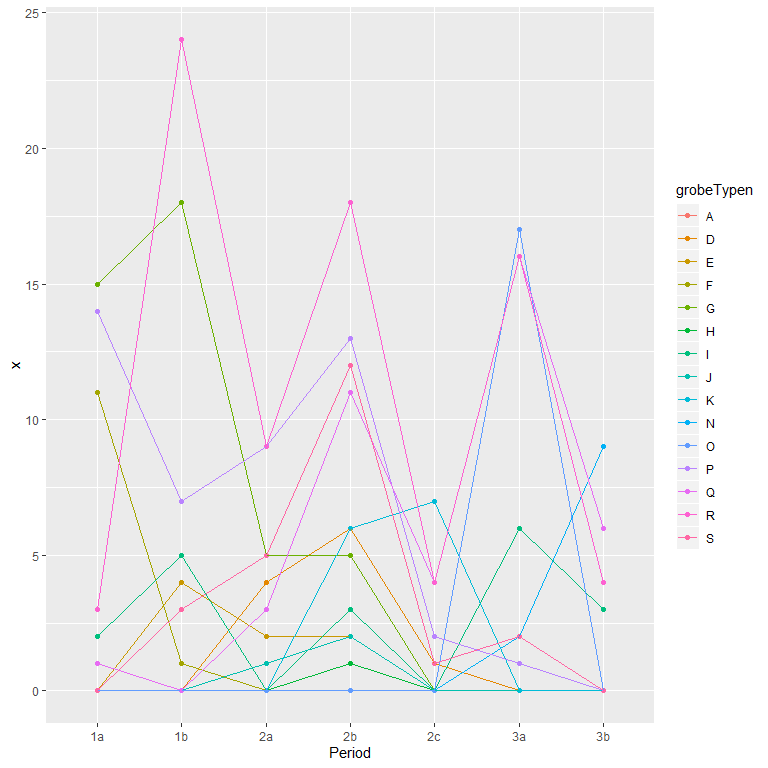
Können wir die Daten also vllt ein bisschen gruppieren?

Ich bin dafür, dass wir nur die Großbuchstaben der Typenbezeichnungen benutzen, weil ich davon ausgehe, dass das irgendwelche Übergruppen darstellen könnte. Ich arbeite dafür mit Bornh2 weiter.

Drei Schritte braucht es: 1. Ich brauche eine neue Spalte, in der die neuen Gruppentypenbezeichnungen eingetragen werden , 2. Ich muss dort die richtigen Typen eintragen – in diesem Fall kann ich einfach nur den ersten Buchstaben aus den Typenbezeichnungen behalten Wie das geht, hab ich ergoogelt, sowas hab ich vorher nicht gebraucht – und 3. die neuen Gruppen müssen zusammengerechnet werden mit der Funktion aggregate.

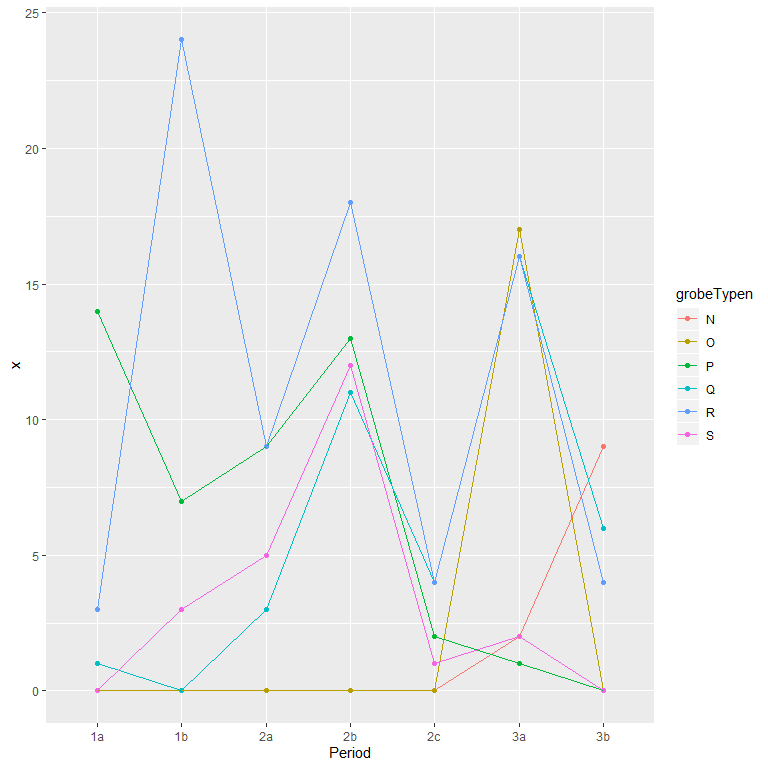
#1. neue Spalte  
Bornh2$grobeTypen <- Bornh2$Typ # hiermit erstelle ich eine neue Spalte, die genau den gleichen Inhalt hat, wie die Typ-Spalte  
  
#2. ersten Buchstaben erhalten (Buchstaben 1 bis 1 erhalten)  
# wir brauchen ein neues Paket namens "stringr". Bitte installiert es.  
library(stringr)  
  
Bornh2$grobeTypen <- str\_sub(Bornh2$Typ, 1,1) # wir nehmen die Worte in Bornh2$Typ und benutzen nur Buchstabe 1 bis 1  
  
#3. Daten wieder zusammenfassen:   
# es sind die gleichen Datensaetze mit unterschiedlichen Haeufigkeiten entstanden, die muessen noch mal zusammengefasst werden  
  
Bornh3 <- aggregate(Bornh2$Haeufigkeit, by = list(grobeTypen = Bornh2$grobeTypen, Period = Bornh2$Period), FUN = sum)

# wieso hier ein neuer Code Chunk? Reine Gewohnheit meinerseits, damit ich einen besseren Überblick behalte, trenne ich nach längeren Umwandlungen gern den Code für die Grafik nochmal ab.  
# neues Diagramm  
  
ggplot(data = Bornh3)+  
 geom\_point(aes(x = Period, y =x, color = grobeTypen))+  
 geom\_line(aes(x = Period, y = x, color = grobeTypen, group = grobeTypen))

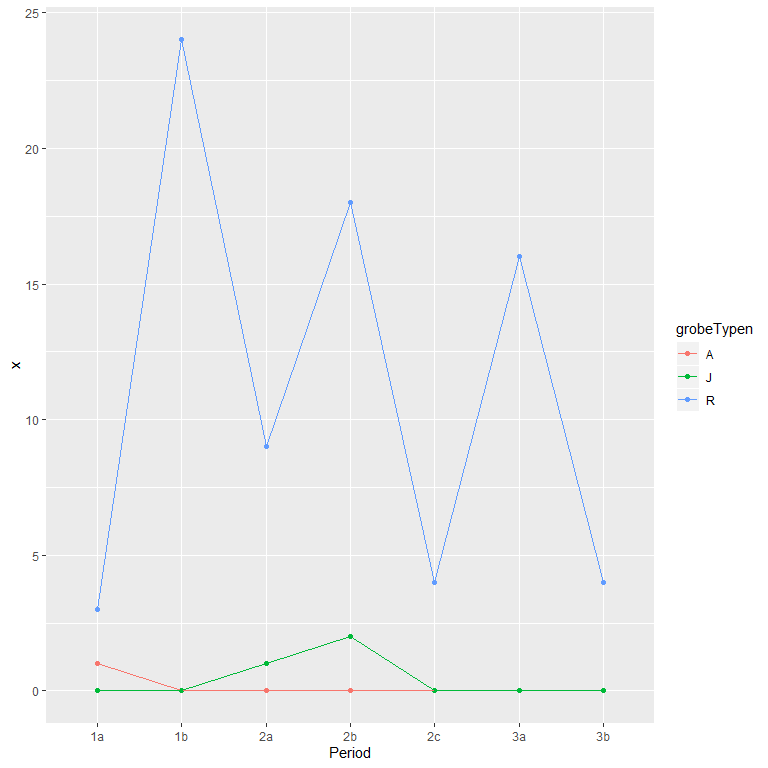
 Das sieht doch gleich viel besser aus. Auch wenn immer noch etwas viel vllt.

Ich kann den Datensatz auch noch weiter verkleinern, in dem ich mir zB nur bestimmte Typen rauspicke. Dafür gibt es unterschiedliche Möglichkeiten. Die subset-Funktion, mit der ich dann auch wieder neue Datensätze erstelle oder das Filtern:

Bornh3%>%  
 filter(grobeTypen > "M")%>% # nur die groben Typen, die "größer als M" sind, also nach M im Alphabet kommen  
 ggplot()+  
 geom\_point(aes(x = Period, y =x, color = grobeTypen))+  
 geom\_line(aes(x = Period, y = x, color = grobeTypen, group = grobeTypen))

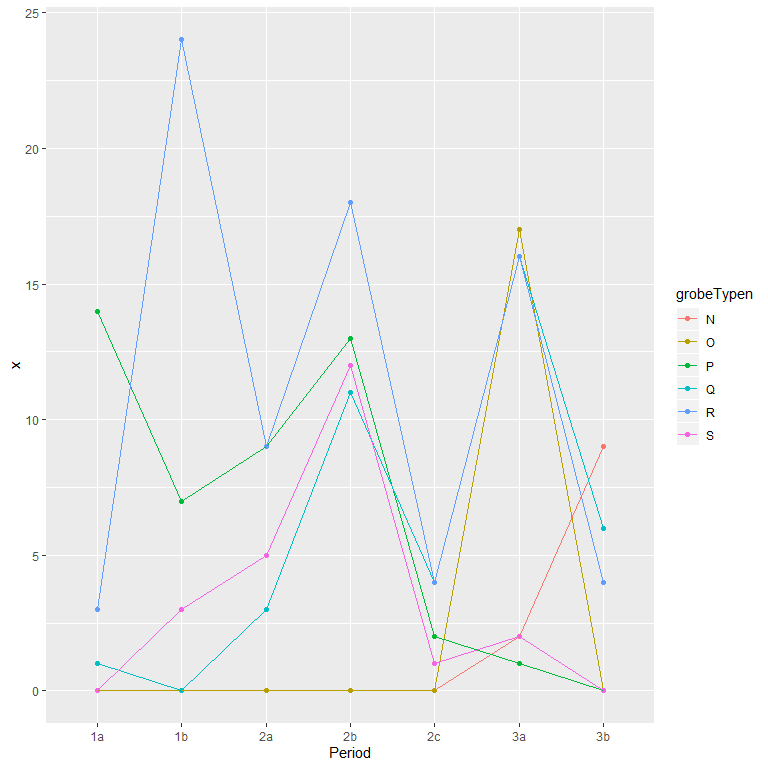


#oder  
Bornh3%>%  
 filter(grobeTypen == "A" | grobeTypen== "J" | grobeTypen == "R")%>% #bitte nur die Zeilen, wo grobe Typen A, J oder R ist. Das Zeichen für ODER ist ein senkrechter Strich und findet sich links unten auf der deutschen Tastatur.  
 ggplot()+  
 geom\_point(aes(x = Period, y =x, color = grobeTypen))+  
 geom\_line(aes(x = Period, y = x, color = grobeTypen, group = grobeTypen))

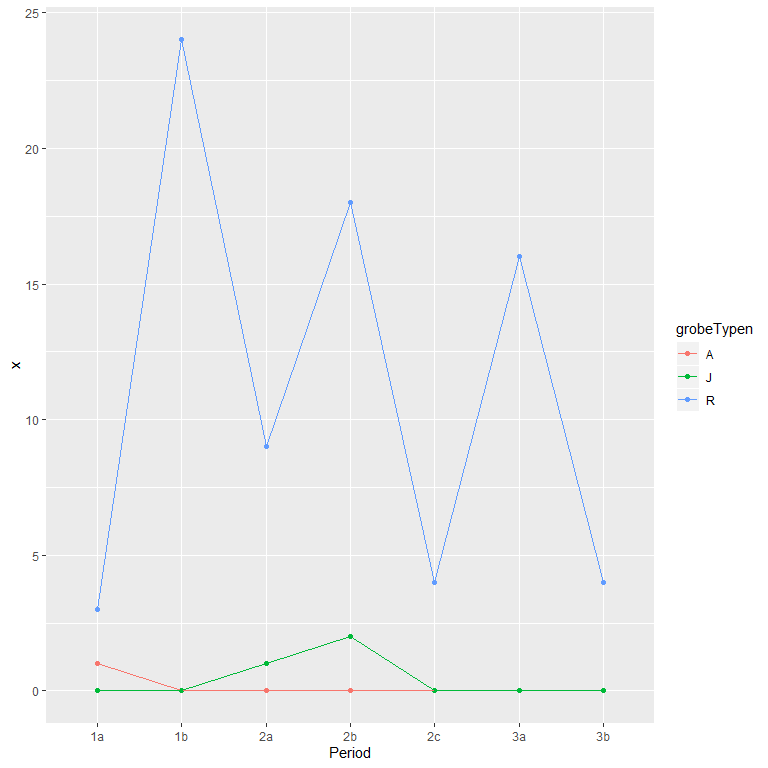


Das gleiche mit subset:

Bornh4 <- subset(Bornh3, Bornh3$grobeTypen > "M") #ich nehme eine Auswahl von Bornh3 und zwar da, wo Bornh3$grobe Typen größer als M ist und weise diesem Datensatz die Variable Bornh4 zu  
  
ggplot(data = Bornh4)+  
 geom\_point(aes(x = Period, y =x, color = grobeTypen))+  
 geom\_line(aes(x = Period, y = x, color = grobeTypen, group = grobeTypen))



Bornh5 <- subset(Bornh3, Bornh3$grobeTypen == "A" | grobeTypen== "J" | grobeTypen == "R")  
  
ggplot(data = Bornh5)+  
 geom\_point(aes(x = Period, y =x, color = grobeTypen))+  
 geom\_line(aes(x = Period, y = x, color = grobeTypen, group = grobeTypen))

 Ob ich subset nehme oder filter, liegt ähnlich wie bei der Entscheidung oben zwischen tidy verse und old school R v.a. daran, ob ich mit dem reduzierten Datensatz noch häufiger arbeiten werde. Wenn ja, dann weise ich ihm lieber eine Variable zu, weil die Chance dann auch nicht ganz schlecht steht, dass ich noch weiß, was das für eine Variable ist (Bornh4 ist ein ganz schlechter Name. Sinnvoll wäre Bornh\_M oder so gewesen). Wenn ich das aber nur ein-zweimal für eine Visualisierung mache, dann nutze ich filter.

Welche Auswahlen jedoch archäologisch relevant ist, entscheidet ihr als Expert\*innen im Feld!

Jetzt reicht es aber auch langsam mit den Liniendiagrammen, machen wir doch mal was mit Dichte.

## Dichtediagramme!

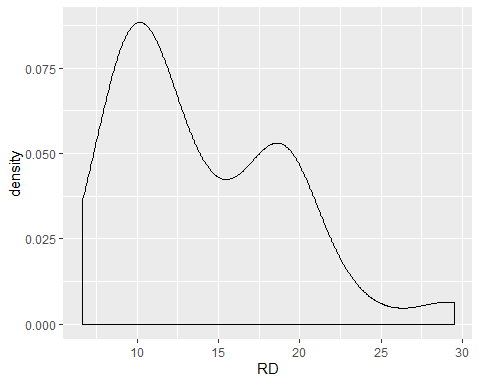
Wie die funktionieren, habt ihr hoffentlich gelesen.

Bei Dichtediagrammen wird die x-Achse wieder metrisch. Das geht also NICHT mit den Periodenangaben, mit denen wir die ganze Zeit eben rumgespielt haben.

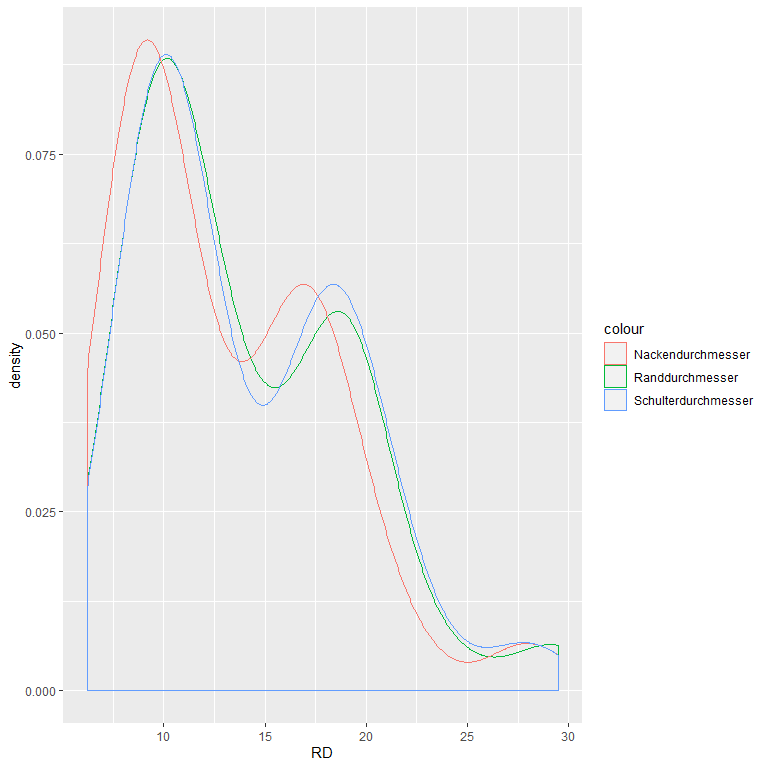
Wir könne stattdessen wieder mit den Bronzezeitlichen Tassen arbeiten.

Die Funktion in R heißt density.

ggplot(data = BACups)+  
 geom\_density(aes(x = RD))

 Das war doch ganz einfach. Aber jetzt möchte ich gern die unterschiedlichen Messungen an den Tassen im Vergleich sehen.

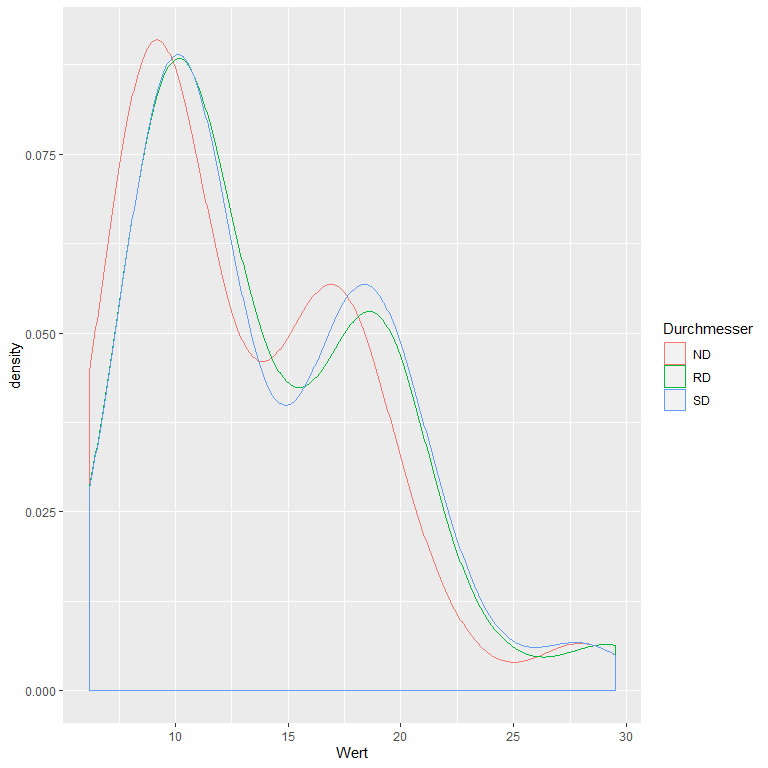
# Ich kann ganz einfach den geom\_density-Befehl mehrmals rufen  
  
ggplot(data = BACups)+  
 geom\_density(aes(x = RD, col ="Randdurchmesser"))+  
 geom\_density(aes(x = ND, col = "Nackendurchmesser"))+  
 geom\_density(aes(x = SD, col = "Schulterdurchmesser"))

 Das funktioniert zwar, aber es ist nicht gerade “elegant”.

Elegant wär es, erst die Daten so umzuformen, dass sich in ggplot dann mit möglichst wenig Befehlen meine Grafik darstellen lassen kann.

Wir brauchen also wieder die Umformungen mit tidyr. Der Schritt ist relativ einfach. Ich nehm die Spalten, die mich interessieren sortiere die Werte dieser Spalten neu, so dass ich eine Spalte habe, in der steht die ehemalige Überschrift der Spalte und einen Spalte, in der der dazugehörige Wert steht. Das ist wieder der gather-Befehl.

BACups%>%  
 gather(key = "Durchmesser", value = "Wert", "RD", "ND","SD") %>%  
 ggplot()+  
 geom\_density(aes(x = Wert, col= Durchmesser))

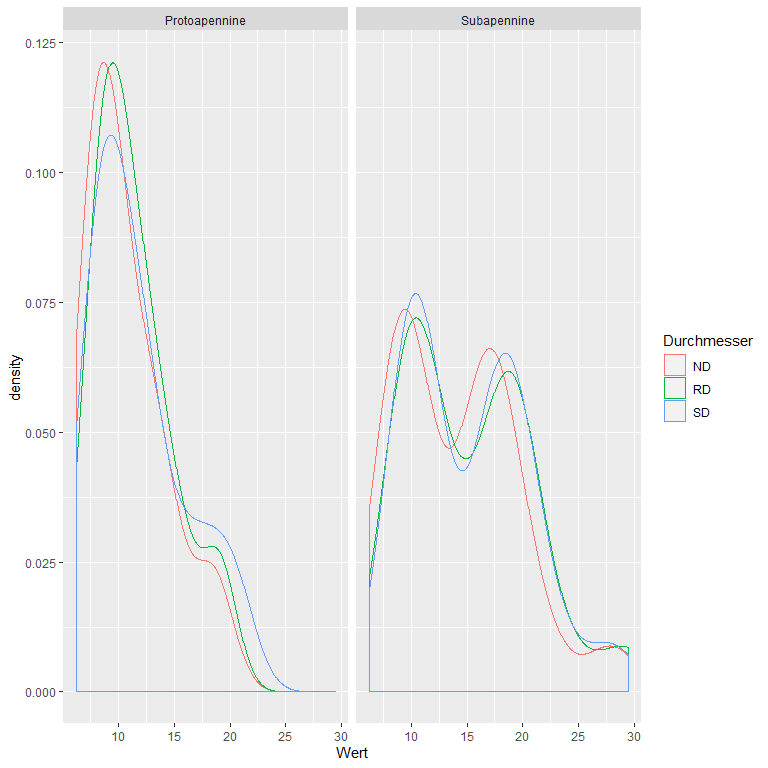


# genauso viele Zeilen Code aber schicker.

## Facettierung!

Jetzt wird es nochmal richtig cool. Der Dichteplot eben, den nochmal nach unterschiedlichen Phasen anzulegen, das wär gut oder?

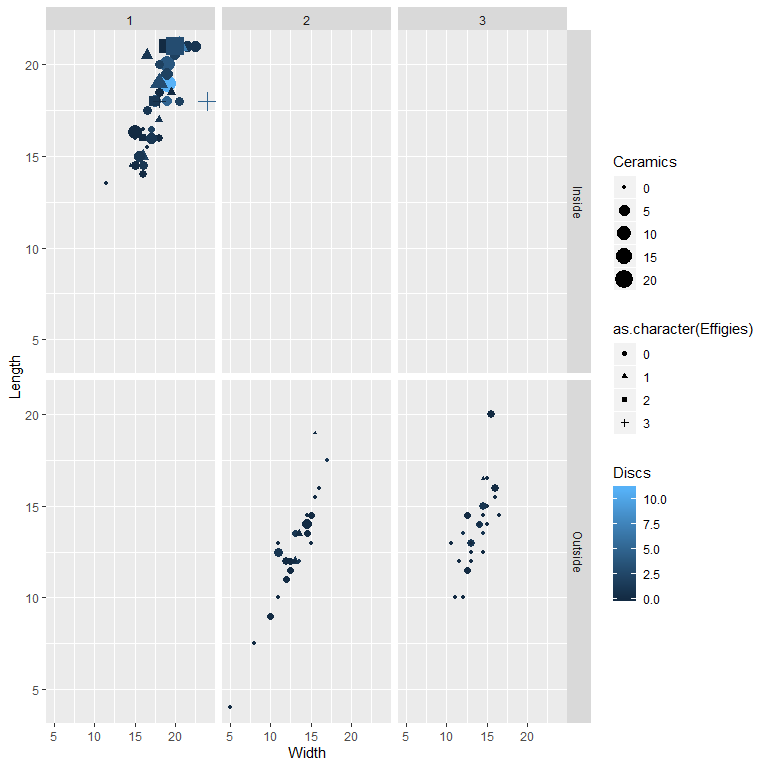
BACups%>%  
 gather(key = "Durchmesser", value = "Wert", "RD", "ND","SD") %>%  
 ggplot()+  
 geom\_density(aes(x = Wert, col= Durchmesser))+  
 facet\_grid(.~Phase)

 Nur eine einzige Zeile Code mehr und schaut es euch an: Interessantes Ergebnis oder? Die ganzen hohen Durchmesser-Werte kommen fast alle aus der subappeninen Phase. Interessant!

Noch ein anderes Bsp fürs Facettieren. Nehmen wir doch mal den Snodgrass-Datensatz mit den Häusern. Da gibt es zwei nominale Attribute, die man bei der “Facettierung” geenüber stellen kann.

Achja. Und ich benutz mal alle Variablen eines Streudiagramms, die mir einfallen… Versucht mal durchzublicken.

data("Snodgrass")  
  
ggplot(data = Snodgrass)+  
 geom\_point(aes(x = Width, y = Length, col= Discs , shape = as.character(Effigies), size = Ceramics))+  
 facet\_grid(Inside~Segment)



# Was passiert hier alles?

1. Daten umformen 15min

## eigene Daten einladen

## Last comments

GGplot hat noch viel viel mehr Möglichkeiten. Um einen Überblick zu bekommen, empfehle ich den Blogpost hier zu lesen, der vorführt, 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 R und 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 deutsche Wikibook zu R: <https://de.wikibooks.org/wiki/GNU_R> und
* das englische R-Cookbook: <http://www.cookbook-r.com/>
* die 6 häufigsten Fehler: <https://bookdown.org/chesterismay/rbasics/6-errors.html>

## weitere spannende Dinge in R:

* Rmarkdown: Text und Code in einem Dokument dazu:
* The Definitive Guide: <https://bookdown.org/yihui/rmarkdown/>
* offizielle website: <https://rmarkdown.rstudio.com/index.html>
* sp, sf, spatstat, maptools: Pakete für räumliche Analysen
* rrtools: Projekt als R-Paket abspeichern (Text, Code und Daten in einem)

## Hilfestellungen 15min

1. Hilfe-Funktion in R
2. Ben Marwicks page
3. Data Science Buch
4. YARRR
5. google-tips