R- a tool for statistical analysis

A modern introduction using tidyverse

Brian Williams

Epidemiology and Global Health Unit,
Department of Public Health and Medicine,
Umeå University

November 26, 2019

Preliminaries

Brian Williams <bjw649@gmail.com>

Installing R

Download and install R from the R Homepage (http://www.r-project.org/).

Choose your computer, then operating system. Use the 'base' install for Windows.

Use default values throughout the install.

While you are at the web site, you will see links to 'CRAN' - go there and browse around and see what documentation is available. You'll find lots of examples, tutorials etc.

Installing Rstudio

1. Go to the Rstudio download page (http://www.rstudio.com/products/rstudio/download/).
2. When you get to the page, look for your computer/operating system under the heading 'Installers for ALL platforms'. Click on yours.
3. You can choose a 'mirror' location for a download of the installer .exe file.
4. When the file is finished downloading. Click on it.
5. On a Windows platform you will be asked whether it's OK for the program to make changes to your computer. Click 'Yes'.
6. The installer opens up - close other applications, click 'Next'.
7. Accept the default settings - click 'Next' twice more then click 'Finish'.

An RStudio icon should now be available on your Desktop or Start menu (or equivalent - depending on your operating system). You can open RStudio by double-clicking on the icon.

The program will normally associate with files with extensions (.r, .R, .Rmd, .Rnw) where the case of the letters does not matter. Some software, (EndNote, for example), may also like to use one or more of those extensions, in which case you may like to choose the default?). You can change associations with file types by right clicking on a file with an extension of interest and selecting 'Properties'. The 'General' tab for Properties provides an opportunity to change the file association.

Getting the Course materials from GitHub

The course material is in the R_Course_ folder.

1. **Resources** contains R_Course_Notes.pdf, miscellaneous resources, reference material etc. (Additional material may be added from time to time.)

2. **Data** contains the (moderate sized) datasets accessed by the code used in the course. (Some larger datasets will be accessed directly via the web.)

3. **myR_Code** will contain code you are developing during the course in tutorials etc.

Now:

1. **Open RStudio**, by clicking on the icon.

2. From the File menu in RStudio, **open Lecture1.R** in the folder Resources/courseCode/. (Select [File -> Open File...].) In these notes, menu items will be indicated in square brackets with arrows as shown in the previous sentence. The file `courseCode.R` contains all of the R code used in the notes. Each Lecture and Tutorial is clearly identified.

3. Open Lecture1.R. (Double click it).

4. Save this file in your myCode folder as Lecture1.R [File-> Save As ....]

5. Use mouse/menu to **set the working directory** to the file location: [Session -> Set Working Directory -> To Source File Location]. You will see in the Console Pane (bottom left of the screen, see Figure 1.1) that the R function for this is `setwd()`. **This is an important step!** Without it, R won't be able to find the data files, which have all been put in the Data folder which is found relative to the location of your Code folder.

The myCode folder is for your use. I'll refer you to it from time to time. You can try code out in this folder and because of the file structure we have here, you'll be able to access data in the Data folder in exactly the same way as we will do in class from the Code folder. If it get's too cluttered, create a Workspace folder in the R_Course folder. It will also be most convenient for you to put the new code for your assignments and the final exam in new folders called something like 'myAssignments' and 'myExamfolders' in the R_Course folder, because then you'll be able to access the Data folder in the same way.

Now we're ready for the class!

Preface

0.1 Original Preface by Brian Williams

Brian Williams <bjw649@gmail.com>

This set of notes is intended as an introduction to the use of R for Public Health Students. These notes are themselves an example of what can be achieved using one of the R User interfaces (RStudio) making use of the R package `knitr`.

This is *not* a course on statistics - the expectation is that students will already have undertaken statistics courses (or be about to do so). The emphasis here is on using a consistent subset of tools for preparing data for analysis and taking a preliminary look at it using tabulation of aggregates and visualization. The course will largely work with Hadley Wickham's 'tidyverse' packages.

The course is limited to 13 Lectures of around one to two hours duration and 13 tutorials of similar length and as a consequence, there are limitations on the amount of material which can be covered. I have tried to include appropriate references for further reading wherever discussion has been limited.

Apart from the broader basics of R, I have included a view of the future of data science where it might be of interest to researchers in the Public Health.

I have introduced the idea of Markdown in earlier courses, when RStudio converted it to nice HTML documents, but now newer versions of RStudio have made conversion to MS Word documents straightforward.

To my way of thinking, the ease with which documentation of analytical processes can be maintained now, mandates that it should always be done. Doing so aids the researcher in on-going recording of his or her own thinking, methodology, experimentation etc, and it is especially valuable when there are several approaches to choose from, requiring comparisons across perhaps complex procedures. It is also an excellent way of providing fully accessible information for supervisors or collaborators. Ultimately, this on-going, easy-to-do rigorous documentation provides a ready made basis for publication of completed research projects.

That said, with the rapidly changing experimental and analytical technology (changing data bases, analytical software etc), *maintenance* of research in reproducible form is a significant issue. My own view is that given the rapid development of research in general, and the potential maintenance work-load, that we should aim for 'reproducibility' to be maintained for around five years. In some very actively developing areas, three years may be more sensible, while in other more stable areas, a longer time-frame may be manageable without undue effort. Maintaining reproducibility of software in cutting edge development will always require additional time and often may appear to be time spent for no productive outcome.

Bearing in mind that the intended audience of this course is not a group of computer scientists and some may have no experience in programming, I have taken a rather infomal approach trying to introduce examples early in the course which would motivate the audience, rather than beginning formally with an extensive set of definitions of language as one might find in a computer science text book. The 'An Introduction to R' on the CRAN web page is an excellent more formal presentation if you prefer that approach. I would like to think that these notes and that document complement one another. The draft 'R Language Definition' on the CRAN site is still more formal.

For those with no experience in R or programming, there may seem to be an overwhelming amount of information to comprehend. I can only advise you to not panic, relax, and don't worry about things you don't yet fully understand. *You cannot hope to be fully conversant in R in two weeks*! (I'm certainly not after dabbling for more than 10 years!)

One of the most important lessons in the course will be to learn how to find answers to your R questions. Modern programmers work with multiple languages, editors, protocols, operating systems etc and cannot be expected to remember function arguments, specialised syntax etc. The real skill is being able to efficiently find out how to do something - not remember how to do everything!

R too, has many thousands of functions. Google usage is fundamental. There are a number of forums on the web which are generally very friendly and supportive if you are struggling with a problem. (Nabble, Stack Overflow are a good start.) You do need to follow forum protocols - almost all require a minimal example of your problem.

There are many other sources of examples of R code, most of which you will find using Google. Don't forget the CRAN pages - there are a lot of resources there!

The notes are a 'work-in-progress'. They have largely been derived from my own work using R over the past ten years or so, and then cast into a teaching form in workshops and classes in the Epidemiology and Global Health Unit in the Department of Public Health and Medicine at Umeå University in the last couple of years.

I apologise for 'glitches' in the text (there will be many), but at the same time, I am hopeful that the notes (and classes) will provide a good basis for your venturing into R.

The notes will probably continue development in the next couple of years - look out for updates in the GitHub repo.

Brian Williams
March 2019.

0.2 2nd Preface by Kaspar Meili

Sadly, due to other commitments, Brian is no longer able to continue the development of the course files to the same extend as he used to. Luckily he agreed to let me help him maintaining the course and as a first step I uploaded the material to GitHub. We decided to publish the under the Creative Commons Attribution 4.0 International License CC BY. `https://creativecommons.org/licenses/by/4.0/`.

Contents

```
library(knitr)
set_parent("../tidyRcourseBook.rnw")
library(tidyverse)
warn <- FALSE
EVAL <- TRUE
```

Chapter 1

Lecture 1 - Introduction to R and RStudio

- and a glimpse of what's coming!

Brian Williams <bjw649@gmail.com>

1.1 Summary

In this session there will be a brief introduction to RStudio and reading, manipulating, cross-tabulating and visualizing data from dataframes using the *tidyverse* package in RStudio.

Note: In this lecture some advanced features of R will be demonstrated. **Don't worry about what appears to be complex commands**. You will be shown how to easily step through the code and see what comes from it!

RStudio is the most widely used integrated development environment (IDE) for R and provides excellent support for documenting code for reproducibility.

1.1.1 Tidyverse

Tidyverse is a suite of modern R software, largely developed by Hadley Wickham and colleagues at RStudio, which aims to reduce R's learning curve by providing a more consistent way of dealing with dataframes.

Not all the software in tidyverse is limited to dataframes, but that is its broad goal. Tidyverse is still under development, though what we will use is well tried and tested.

There are still many procedures of interest to us in 'Core' R, which currently lie outside tidyverse and we will use these as the need arises.

Check out the freely available web version of the book "R for data science" by Hadley and Garret Grolemund [**Wickham2016**]. This course is inspired by the approach in that book. Much of the material in this course is extracted and simplified from an earlier course aimed at research students in the Medical Faculty. The earlier course notes are available in the resources folder (R4researchFeb2016.pdf) and may provide expanded background on many topics. The R4researchFeb2016.pdf notes are also live-indexed and have a live table of contents, so you should be able to find things in those notes reasonably easily.

1.2 Preliminaries

Load and install R, RStudio and extract the folders from R_Course.zip (available on the Course pages on Cambro) in a convenient folder on your computer. (Extracting will normally create a new folder called R_Course).

1.3 Getting started

Note that you will follow the steps in this section at the beginning of every Lecture and Tutorial.

Assuming you have taken all the steps in the Preliminaries section:

Navigate to your R_Course folder. It should contain 3 folders, Data, myCode, Resources and files called ReadMe.pdf, Preliminaries.pdf and RCourseOutlineRev2019.pdf.

Open R_Notes.pdf, located in your Resources folder.

You will need to open the Resources folder and then the R_Notes.pdf at the beginning of each class. You will be able to follow the class presentation and use the live links in the notes for each Lecture and Tutorial. Note that the index is also live-linked so you should be able to find your way about without too much trouble. (If you are using Acrobat and Windows, [Alt <-] pressed together will take you back from the index to where you clicked on the index link.)

Now:

1. **Open RStudio**, by clicking on the icon.
2. From the File menu in RStudio, **open the folder courseCode** in the folder R_Course/Resources (Select [File -> Open File...].) In these notes, menu items will be indicated in square brackets with arrows as shown in the previous sentence. The folder `courseCode` contains all of the R code used in the notes. Each Lecture and Tutorial is clearly identified.
3. Double click on Lecture1.R and it will open in The Edit Pane of your RStudio window (see Figure 1.1).
4. Use [File -> Save As...] in the file menu to save the file in your myRcode folder.
5. Use mouse/menu to **set the working directory** to the file location: [Session -> Set Working Directory -> To Source File Location]. You will see in the Console Pane (bottom left of the screen, see Figure 1.1) that the R function for this is `setwd()`.

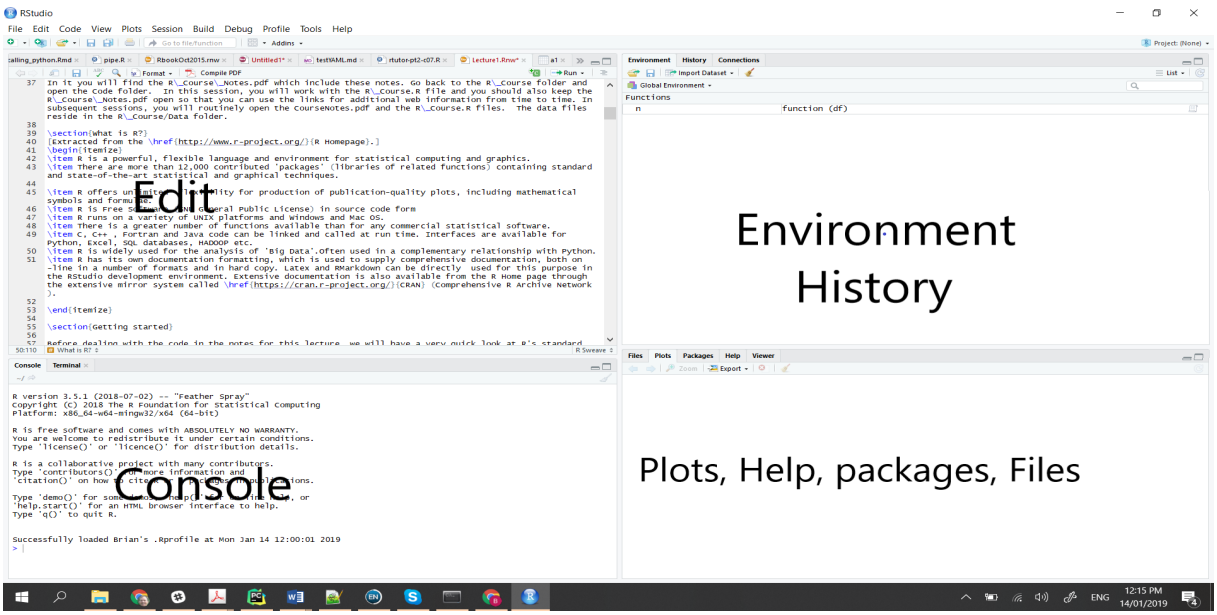


Figure 1.1: The four panes in the RStudio window

This sets your working directory to the myCode folder. **This is an important step!** Without it, R won't be able to find the data files, which have all been put in the Data folder which is found relative to the location of your myCode folder. Both the Data folder and your myCode folder are in the R_Course folder (their 'parent' folder). If you are not clear take a look at Section 27.1 in the Appendix.

You will see that the top left window pane of the RStudio interface now contains the Lecture1.R file. We refer to this pane as the **Edit pane** or 'script' pane or window (See Figure 1.1. Notice that the name of the file is shown on a 'tab' above the window. You can use this pane to edit a number of different scripts using the tabs, and from this pane you can run selected code directly, by clicking on the 'Run' button on the pane's toolbar.

If you compare the text in beige, below in these notes, with the first 'block' in the Edit pane, you will see that in the notes, each line is preceded by '##'. This is just a decoration generated by the software for these notes. The proper R statements without '##' are in your Lecture1.R file, as displayed in the edit pane of RStudio.

Beneath the Edit pane, is the **Console pane**(See Figure 1.1), in which the evaluated R commands will appear. It has a '>' symbol as a prompt, which indicates that the Console Pane is ready for your next command. You can also paste commands into this pane for execution, or edit a previous command using the up-arrow to retrieve it.

If you see a '+' prompt, it is an indication that you have entered an incomplete command and R is waiting for the rest of the command. If you can see what you have left out, enter it and press return and the command will be executed. Sometimes it will not be clear what is missing and you may have to start again - in that situation to return to the '>' prompt you must press the 'Escape' key.

To clear everything in the console window, press 'Ctrl+L'.

The top right pane (see Figure 1.1) is the **Environment/History pane** (selectable using the tabs). The environment will contain basic information about variables which the user has introduced. There are two groups. At the top is 'Data' which consists of two-dimensional arrays (dataframes look like a spreadsheet table) and underneath are 'Variables' which are all other data objects defined by the user in the session.

The history tab contains the sequence of R commands which the user has entered during the session (and earlier). You can execute these directly by selecting them with a mouse click and then clicking on the tool above, labelled 'To Console'. If you want to insert these commands in your edit window, click where you wanted the commands inserted and then click on the 'To Source ' button in the History Pane.

The bottom right pane has a variety of displays (Files, Plots, Packages and Help) indicated and accessed by the tabs.

Now before we begin -

The text in the sections of the notes in the coloured (beige?) boxes between the dashed lines:

----

are sequences of R commands (scripts). You will find an extraction of all these scripts in Lecture1.R, Tutorial2.R etc in the Resources/Course_R_Code folder. Scripts may include comments. Lines preceded with the # symbol (like the dashed line above) may be used to explain to a reader what the script is doing - or (as here) provide a way of separating or identifying lines of text. These lines preceded with the # symbol are ignored by R's interpretation (computation) of the script.

You can run the example scripts in Lecture1.R in three ways. (The easiest way is the last one!)

First: **select** (highlight) the code using your mouse.

Then **copy** (using the Windows shortcut, `Ctrl-c`). (`Command-c on a Mac`)

Now **click** in the Console window of RStudio (the window below), and **paste** the code (`Ctrl-v`). (Mac: `Command-v`)

If nothing happens, in the console, press the return/enter key.

Second:, after highlighting the code, **click** on the 'Run' button near the top right of the Edit window.

Third (best): Place the cursor on the first line and press `Ctrl-enter`. The code in that line will execute and the cursor will move down to the next line of code. Repeat pressing `Ctrl-enter` until you have stepped through as much of the code as you want to execute.

Try these three methods with the following 4 lines of script:

```
  ##Normally distributed N(0,1) variates, mean, var
  #-------------------------------------------------------------------------------
set.seed(1121)  # set a seed for pseudo-random number generation
(x <- rnorm(20))   # creates a vector, x, containing 20 N(0,1) variates
mean(x)         # computes the mean of the sample
var(x)          # computes the variance of the sample
```

```
#-------------------------------------------------------------------------
```

Now **CLICK in the edit window** away from the selection to deselect (remove the highlight)!!!!

If you don't, the highlighted code will be **LOST** as it is replaced by your next typed characters in the edit window.

It's easy to make this mistake (I do it frequently!) but if you realise you have done it, the edit menu provides an 'undo' item which will get the lost characters back for you. (`Ctrl Z` also works - and you can undo several actions by pressing `Ctrl z` repeatedly). As with all kinds of editing on computers, it's a good idea to save your edited file using (`Ctrl S`) frequently. You can save **all** the files you have opened in RStudio, using (`Ctrl Alt S`).

1.4 HELP!!

There are various ways of getting help in R.

- `??graph` searches the documentation for the word 'graph' and lists findings in the Help window (bottom right).
- `?plot` is used to find help for a *function* (the function `plot`, in this case).
- Similarly, `help(fn)` finds the documentation associated with the function named `fn`. It acts exactly the same as the previous one (the question mark form).
- `example(fn)` prints a number of examples of the use of the function in the Console window.
- Some packages have an introductory document called a 'vignette' which can be accessed with the function `vignette()`, but you should start with `browseVignettes()`!
- Try: `vignette('lmtest-intro')`.
- `data()` lists available data sets (and is used to load datasets, too).

For more complex problems go to the Help window (click on the Help tab if it is not currently selected) and click on the Home icon (the picture of a little house in the tool bar of this window) and you will see links to some of the various documentation on the CRAN web site. Otherwise on the CRAN website itself you can look at reference material on the CRAN site.

.....or simply *use Google* (often the best way for more complicated queries) If your problem is a little complicated, Google is likely to find something on the 'stackoverflow' site. Take your time looking at material on this site. Because there is a significant level of control on posting, the responses are more reliable than many other sites on the web. Look for a big green 'tick' for the best solution to a questioner's problem.

1.5 Functions

We have already used some **functions** in R. Functions are fundamental to the use of commands. Commands always use functions, though it may not always be apparent.

- Functions appear as a character string followed by parenthesis (curved brackets) containing zero or more items called **arguments** separated by commas. These argument control the way the function behaves.
- Most functions have multiple arguments, which have been given **default values**. If you don't specify an argument when you 'call' (i.e. use) the function, its default value will be used.
- We can change the defaults by specifying values for the arguments when we call the function. Arguments in all functions can be changed in this way but must conform to the requirements of the function, which may require, for example, that an argument be of a particular type, or that its length match that of another argument.

To see how a function (say) `mean()` is used, enter `help(mean)` in the Console window and look at the output in the Help pane to the right. Help for functions can also be accessed by typing e.g. `?mean` in the Console window. Try it with `?sd`

There are other examples of functions in the on-line Manual "An introduction to R" on the CRAN website.

1.6 Data frames: reading data and investigating them

In this section we'll see the power of R, but we won't try to look in detail at the function useage - that'll come later, so don't worry too much about the syntax of the commands (functions).

There are a number of functions which allow reading of data sets from other statistical software like SAS and Stata (see the package 'foreign'), but for the moment we'll deal with reading our data from a .csv file, perhaps created by spreadsheet software. We are going to focus our efforts on a single R package called tidyverse, which you will see is implemented by calling the `library` function with tidyverse as its argument. Before doing so, however, we need to install the package.

1.6.1 Installing packages

Tidyverse contains a suite of modern R packages, mostly authored by Hadley Wickham and his colleagues at RStudio. Tidyverse focuses on the preparation, analysis and vizualization of dataframes - and that is the focus of this course.

Installing tidyverse

Installing a package is easy. Go to [Tools->Install packages ...]. Type the name of the package (tidyverse) (it should be predicted) and then click on install. You will see some activity in the console window and then a message indicating that the installation is successful, followed by the Console Pane's standard prompt '>'.

When that's all done, go to the packages pane and click on the 'Packages' tab at the top. Use the window slider at the side or the search at the top to find the tidyverse package in the list. Click in the box on the left and you will see that the library command has been used in the Console window of RStudio. [The libary command will also include the directory path of all your installed packages.] Click on the blue name tidyverse and you will be taken to the help page for the package. All packages will have some kind of help file like this. The Description file may be a bit too terse, but if you click on the User guides..... link, you will usually find more digestible infomation.

1.6.2 Reading a dataframe using the tidyverse package `readr`

In `tidyverse`, the standard function for reading a .csv file is (read_csv).

```
  ##Reading and summarising data frames
  #-------------------------------------------------------------------------------
library(tidyverse)
bP        <- read_csv(file="../Data/BackPain.csv", na = c("","."," ","NA"))      # (1), (2), (3), (4)

## Parsed with column specification:
## cols(
##   .default = col_character(),
##   age = col_double(),
##   bmi = col_double(),
##   waistc = col_double(),
##   comorb = col_double(),
##   disability = col_double(),
```

```
##  height = col_double()
## )
## See spec(...)  for full column specifications.

glimpse(bP)
bP              # bP is a tibble so only 10 lines are printed (and a limited number of variables, too)
```

Notes:

1. The <- symbol assigns the output from the function on the right to the variable named on the left.
2. `read_csv()` reads a '.csv' formatted file specified in the first argument.
3. The na = argument tells the function to process missing, blank or NA as NA.
4. Note that the use of ../ in the specified file path finds the data file by going firstly to the folder of the parent of the working directory. The parent folder is the R_course folder. There it finds the Data folder in which it finds the file "BackPain.csv"!)
5. `read_csv()` assumes the data has a *header row* which is used to create column (Variable) names.
6. The output from `read_csv()` is a form of data frame object called a 'tibble' (More about this later.)
7. Have a look at the help window for `read_csv()`. (Type ?read_csv in the Console window.) You can use additional 'arguments' to specify na's, column header (variable) names etc.
8. Note if comma is used to specify decimal points in your file you can use `read_csv2()`, which assumes a semi-colon separator, or `read_tsv()` for tab separated files. If something else has been used, see the docuentation on `read_delim()`.
9. You can have a look at the data file in the Edit Pane by going to the Environment Pane (top right-hand-side) and clicking on bP - (it's at the top in a section headed 'Data'. Clicking on the little blue arrow will expand a summary in the Environment pane.)
10. You can also import data from SAS, SPSS, or Stata with your mouse! [File-> Import Dataset]. Check the package `haven` for details of the handling of different formats.
11. Specifying the name of a standard dataframe lists the entire df - not a good idea if it's large! Tibbles restrict the print.

The first thing we notice from our `glimpse()` in the Console below, is that `bP` has 34122 observations (rows) and 24 variables (columns). The columns are a mixture of 'dbl' and 'chr'. We'll talk about this more as time goes on, but for the moment we're going to convert all the 'chr' variables to what R calls *factors* - these are called categorical variables in some other statistical environments.

We'll also demonstrate the power of dplyr when using *pipes* in the next code chunk - in a single assignment we convert all the character variables to factors, remove all the NA's in the dataframe, and create a new variable for the waist/height ratio of the subjects. It looks very complex - but really its just like learning another language (Yes, I know - Swedish is very difficult!)

```
bP <- bP %>%                                        # (1)
  mutate_if(is.character, as.factor) %>%            # (2)
  filter(complete.cases(.)) %>%                     # (3)
  mutate(waistHtRatio = waistc/height)              # adds a new variable called waistHtRatio  (4)
```

Notes:

1. The %>% 'pipes' the dataframe on its left to the first argument of the function on its right (in this case on the next line).
2. The `mutate_if` function is converting those variables which are characters, to factors
3. The `complete.cases` function can selectively remove records with NA's in specified variables - in this case it is removing all rows with 1 or more NA's
4. `mutate` creates a new variable as a function of existing variables.

In general when using the complete.cases function, one should select removal only for those rows where at least one of the variables of interest is NA. Here we effectively chose ALL the variables. If you look in the environment pane, you will see that our 30,000 or so records have been reduced to 12,200! We continue for demonstration purposes! [We'll look at this more carefully later!]

1.6.3 Tabulation

Here we introduce a new package based on `tidyverse` called `janitor`. This package simplifies some of the basic tabulation processes that we might be familiar with from other statistical software.

One-way table

```
library(janitor)

##
## Attaching package:  'janitor'
## The following objects are masked from 'package:stats':
##
##     chisq.test, fisher.test
```

```
c1 <- bP %>%
      tabyl(country)
c1
```

1.6.4 Two-way table

```
c2 <- bP %>%
      tabyl(country, eduS)
c2
```

...and some people are more familiar with percentages:

```
c3 <- bP %>%
    tabyl(country, agegr) %>%
    adorn_percentages("row") %>%
    adorn_pct_formatting(digits = 2)
c3
```

The levels of factors by default are listed alphabetically, sometimes there is a sensible order so we change it:

```
 bP <- bP %>%
       mutate(eduS = fct_relevel(eduS,"No primary","Compl Primary",
                                 "Compl Sec/HS", "Compl Uni/Coll"),
                     fct_relevel(bmi4, "Underweight","Normal",
                                 "Pre-Obese", "Obese"))
#-------------------------------------------------------------------------------
```

We'll visualize the bmi4 below.

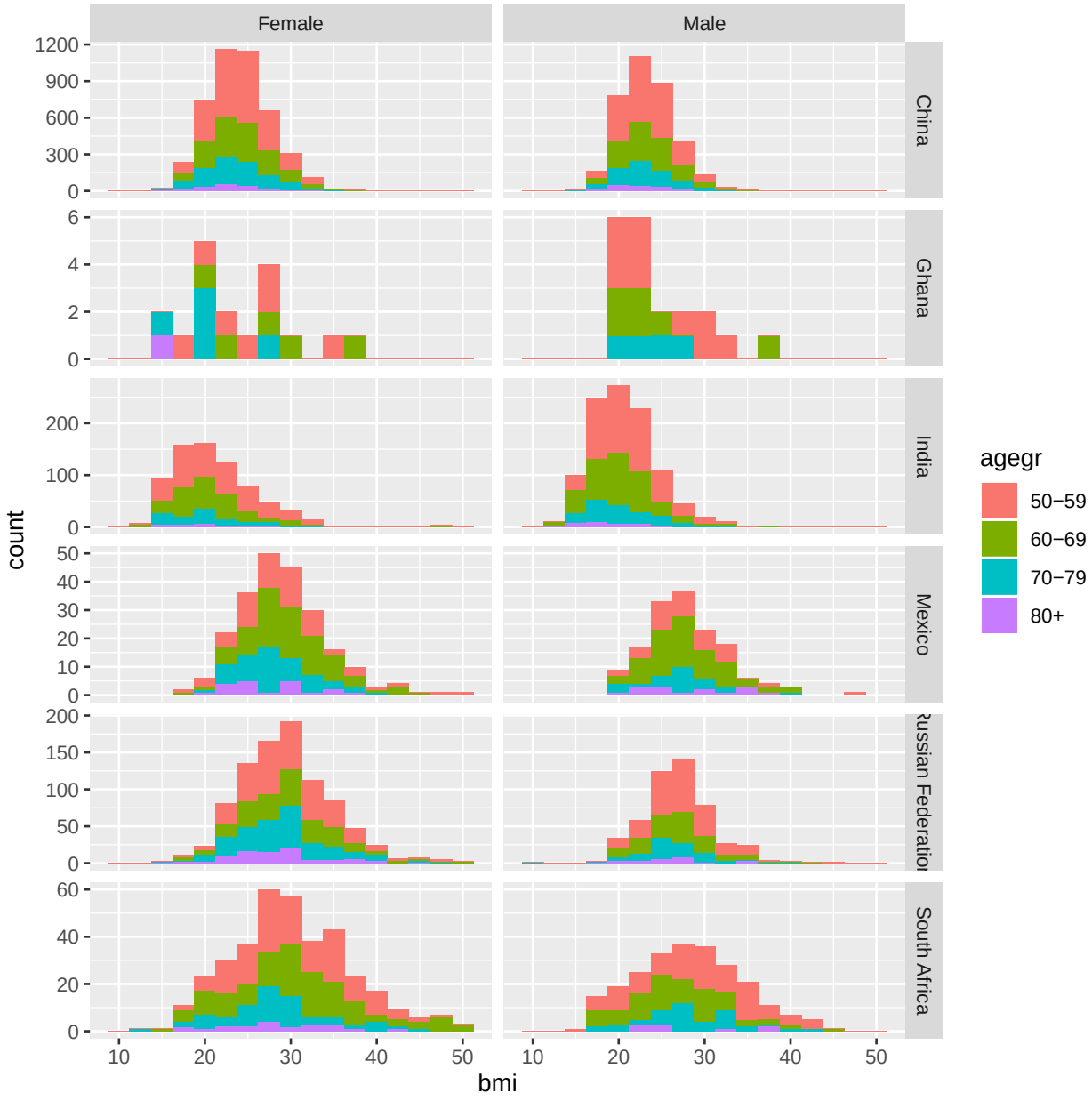
```
c3 <- bP %>%
    tabyl(country, eduS) %>%
    adorn_percentages("row") %>%
    adorn_pct_formatting(digits = 2)
c3
```

```
bySex <- group_by(bP,country,residence, sex) %>%
       summarise( meanDisability = mean(disability), sdDisability=sd(disability))
# kable(bySex)
```

1.7 Visualization

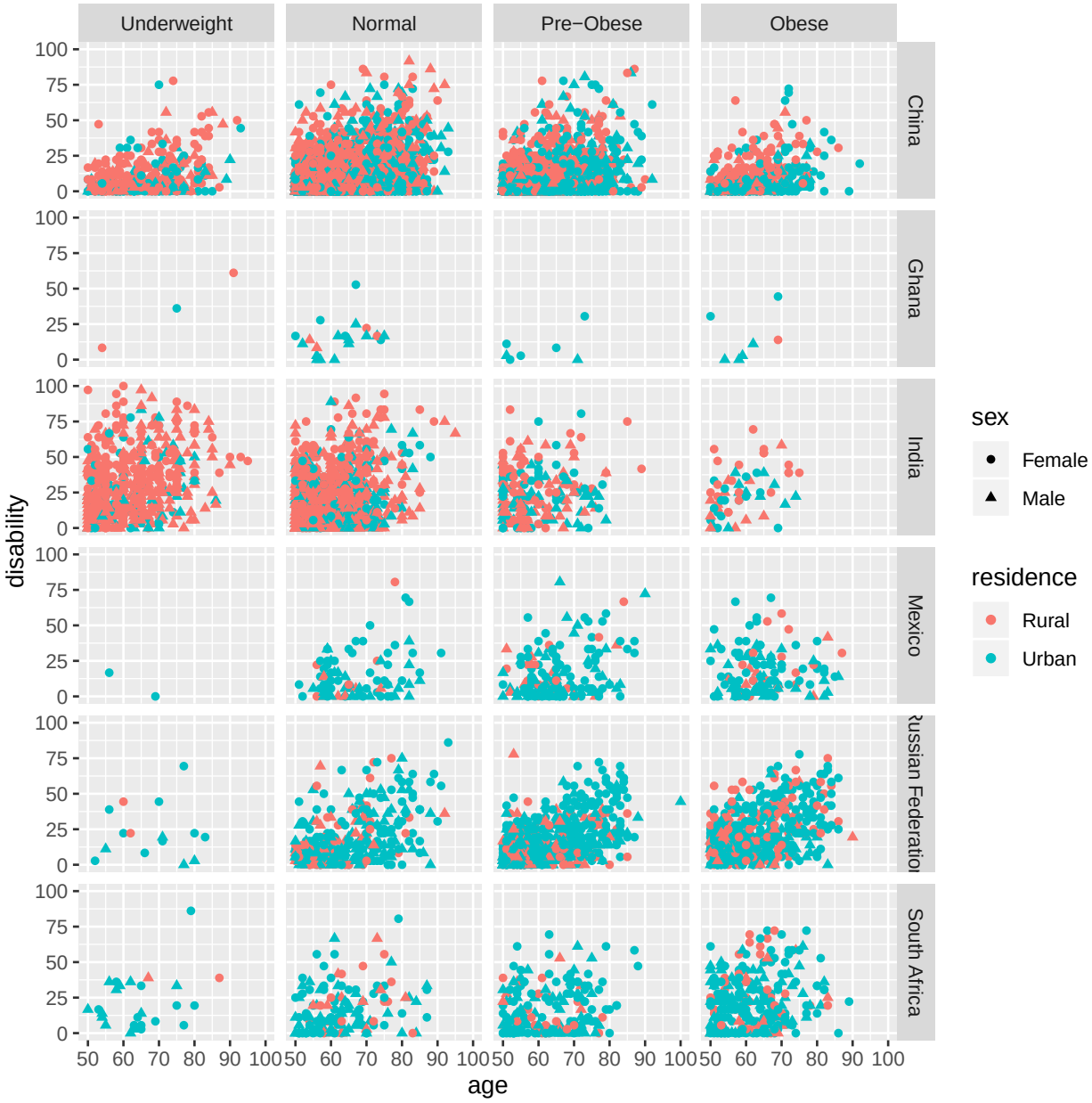
Visualization in the tidyverse uses a package developed by Hadley Wickham called `ggplot2`. This package is widely regarded in the computing industry as one of the very best. However, it preceded the tidyverse ideas of dataframes in and out of functions and so does not quite conform to the idealised model. It produces static two-dimensional plots, which in modern web-computing is 'old-hat' but Wickham has modern developments underway. `ggplot2` is based on a 'Grammar of Graphics' (hence the gg.). This allows the user to add layers of information on a single plot. It takes a little bit of learning, but there is plenty of help on the web and the results are wonderful. Here are a few examples using the backPain data. (We'll take a closer look at this code later.)

```
pp1 <- ggplot(bP, aes(x=bmi, fill=agegr)) +
  geom_histogram(binwidth = 2.5) +
  facet_grid(country ~ sex,scales="free_y")
pp1
```



Note the 'free' y-axis. Its not obvious at first glance that China has much more data than South Africa. An annotation on the plot of the total numbers would draw immediate attention to this. Note too that Ghana's data set (after removal of the NA's) is *very* small. For some unexplained reason, the Ghana data has very few height measurements even though these are necessary for comutation of BMI. Let's have a look at the data for disability vs age.

```
 bP <- bP %>% mutate(bmi4 = factor(bmi4, levels =
                                     c("Underweight","Normal", "Pre-Obese", "Obese")))
p <- ggplot(bP, aes(x = age,  y = disability ,color = residence, shape = sex)) +
  geom_point() +
  facet_grid(country~bmi4)
p
```


Underweight
Normal
Pre−Obese
Obese
China
Ghana
India
Mexico
Russian Federatio
South Africa
disability
age
sex
Female
Male
residence
Rural
Urban

Chapter 2

Tutorial 1 - Rstudio, dataframes, packages and file handing

Brian Williams <bjw649@gmail.com>

2.1 Preliminaries

Go to the Resources folder and open the courseCode.R file. Find "Tutorial1". Copy it. Open RStudio. Open a New R script.Paste Tutorial1 into the edit pane of the New file and save it in your myRcode folder as 'Tutorial1.r'.

2.2 Working Directory

In RStudio you can change to your working directory of choice using the menu as follows: [Session->Set working directory:.] or you can use the standard R command, `setwd()`. If you use the menu to set the working directory to the directory of the current file, using [Session->Set working directory->To Source File Location], you will see the `setwd()` function used in the Console Pane.

```
getwd()
```

Amongst other things, `getwd()` can be helpful in reporting results if you have a complex file system and potential confusion some months later when looking at the output. If you want to see the files in the working directory, then use the function `dir()`.

```
dir()
```

Note that even though it has no arguments you must include the parentheses when using this function.

2.3 Importing external files

```
library(tidyverse)
bP        <- read_csv(file="../Data/BackPain.csv", na = "", comment = '#', skip = 0)
             # bP is a tibble so only 10 lines are printed (and a limited number of variables, too)
select(bP,alcohol)
bP <- bP %>% mutate_if(is.character, as.factor)
select(bP,alcohol)
```

1. The argument list begins with a 'relative' file name. The '..' means go to the parent folder of the working directory then look for a folder called 'Data' and select the file called 'BackPain.csv' in that directory.
2. In R, missing data is indicated with the string 'NA' (without the quotation marks). `na.strings = ""`: This is a 'nothing entered' value for the `na.strings` parameter. This argument provides values of NA for the data frame entries which have the characters listed for this parameter, i.e. with our setting, those entries which have a 'blank' (white space), are made 'NA'. In some other datasets, a missing value might be indicated by a symbol such as '*' or '#' and this argument let's you specify that.
3. You can use the `col_names` argument in two ways. If you set `col_names = FALSE`, `read_csv` skips the first line of the data file and constructs column names with the form X1, X2, X3 etc. The other way of using `col_names` is to set it to a vector of your own choice of column names, i.e. something like: col_names = c("x", "k", "w", ...)

2.4 Cleaning up NA's - rough approach!

Now we will do a very rough cleanup of the data, by simply removing all records which have one or more missing data items (NA's). We'll begin by summing the total number of NA's, then removing the rows in which they are present, and then repeating the sum check to ensure that we have indeed removed all the NA's.

```
sum(is.na(bP))    # compute number of NA's
bP <- bP %>% drop_na()  # Use this to remove all cases (rows) containing at least one NA from
#  The non-tidyverse way  bP <- na.omit(bP)
sum(is.na(bP))     #check whether all NA's are removed
#-------------------------------------------------------------------------------
```

Notes:

- `is.na(X)` returns a vector of `TRUE` or `FALSE` for every element of X. When summed, TRUE = 1, FALSE =0.
- `omit.na()` and `tidyR::drop_na()` are very powerful and produces a 'complete cases' data frame - but be careful, you may lose a lot of data! Look at the new size of bP!
- Where logicals are included in a numerical computation, R assumes `TRUE` = 1, `FALSE` = 0, so `sum(is.na(bP))` computes the total number of NA's in the data frame.

2.4.1 Referencing data frame elements

Outside of tidyverse, the preferred way of referring to variables is with the '$' notation, e.g. **dfn$varName** - where dfn is the data frame name, and varName is the variable name. You can also refer to it using the number of the column in square brackets. If you use the name of the column, you will still correctly access it if you change the column numbering in some other operation. However, there are times when using column numbers is convenient.

We use square brackets with row, column indexes to select individual elements of the data frame. **bP[3,2]** is the 3rd element of the variable in the second column.

If the comma is kept but either index is omitted, all elements of the missing index are selected. Thus **bP[,2]** refers to all values recorded for the variable in the second column and **bP[2,]** refers to all items recorded in the second row.

In operations where a number of variables are to be accessed, it is sometimes easier to use their column number with the `bP[colNumber]` notation.

Suppose that in the back pain dataset, we carelessly assume the 11th column is named "bmi".

```
##Column selection in data frames  *******Use names!!!!!!
#-------------------------------------------------------------------------------
head(bP[11])      #print out the 1st 6 elements of bmi....OOOPS!
head(bP$bmi)       #print out the 1st 6 elements of bmi
names(bP)          # Ah - bmi is col 12!
#-------------------------------------------------------------------------------
bP %>% select(bmi)
```

2.4.2 Imports from other statistical packages

The R package 'foreign' enables reading of data stored by Minitab, S, SAS, SPSS, Stata, Systat and others. You can also export (Write) files from R in the formats of some other packages, including Stata. Objects such as data frames can be saved as ASCII (text) files to produce .csv files which can be read from almost any other program. ASCII files, however, will be much larger than binary files like .rda (see below under Save). Here's an example of reading from a Stata file. We'll save the file as a .Rdata file and use it in Tutorials later. This file contains some additional variables.

```
library(haven)
bPx <- read_dta("../Data/BackPainData_1.dta")
summary(bPx)
save(bPx, file =  "../Data/BackPainData_1.Rdata")
```

2.5 Listing and removing objects

Let's create a 'vector' object so we can see how objects are listed and removed in R and RStudio.

```
vect1 <- bP$agegr                                    #1 Created a vector object
ls()                                            #2 List all objects in WD
objects()                                                #2 List all objects in WD
remove(vect1)                                           #3 Remove object 'vect1'
ls()
```

1. The $ notation has been used to select the column of the variable `agegr`. The code has assigned the vector of the variable to 'vect1'.
2. Objects are listed by using either `ls()` or `objects()`. You must include the parentheses when using these functions. Note too, that the objects are already listed in the Environment Pane in RStudio.
3. Objects can be removed using `remove()` or its abbreviated form, `rm()`.
4. **All variables** in the Environment can be removed by clicking on the little 'brush' icon in the Environment toolbar.

2.6 Saving objects and workspaces to files, reloading

The `save()` function is straightforward.

With respect to the data we have been looking at here, we could do the following:

```
save("bP", file="BackPain.RData")
dir()
```

Now let's remove the object bP from our environment, and restore it by loading the saved file "BackPain.RData".

```
remove(bP)
ls()
```

Notice that bP is gone from the Environment pane as well as missing from the output of `ls()`.

```
load("BackPain.RData")
ls()
```

Now we have it back.

Notice that the loaded data takes the *original object name*, not the file name. If you want to rename the new object, you must firstly copy it and then remove the old object:

```
bP2 <- bP
ls()
rm(bP)
ls()
```

R is clever about this. Even though you will see both objects listed, R does not actually make a copy (which if your object is very large might cause a memory problem). When the R interpreter reaches the `rm()` function it *then* simply changes the name of the object in memory.

The `save()` function allows saving of multiple objects either named at the beginning of the command itself, or the first argument can be a list of the objects to be saved in the form of a character vector of the object names. At the conclusion of a session `save.image(".Rdata")` is automatically offered. This saves the entire workspace so that when R is next opened, the workspace can be loaded with `load(".Rdata")`. Pay attention to the directory in which this is taking place! RStudio auto-loads the .Rdata file in your Documents folder at start-up. If you load the wrong .Rdata file, you can remove all files easily by clicking on the 'clear' tool in the Environment pane. Sometimes, you will want to save a workspace or a large cleaned data frame under another name so that it is not inadvertently opened or deleted (`save(myDataFrame,file = "myFileName")`). You can actually save a list of objects in this way (`save(myDataFrame, a, b, c, file = "myFileName")`). You can also choose to save the data in ASCII (text) format (`save(myDataFrame,file = "myFileName", ASCII=TRUE)`). Any time you want to load these back into your working environment, simply `load( "myFileName")`. Try it out! There are a few functions that are important to know for saving workspace data to disk and subsequently re-loading it. You can see what data objects and functions you have created in the Environment pane. You can list them in the console pane using `objects()`. You can remove a number of objects with the `rm()` command.

There are two other related functions which can be very useful in the construction of self-contained *minimal reproducible examples*. These functions are `dput()` and `dget()`.

```
t1 <- "The cow jumped over the moon, the little dog laughed to see such fun"
dput(t1, file=t1)
t2 <- dget(t1)
t2
```

See `help(dput)` for more information.

2.7 Packages and their data

Packages can be readily installed in RStudio, using the Tools menu: [Tools-> Install Packages...] then enter the package name in the middle edit window - leave the others. You will see some activity in the Console pane as the package downloads.. Try installing the package `dplyr` - we will use it frequently in this course.

To get an overview of the functions associated with a package, for example dplyr, use the help function with a package argument: `help(package = "dplyrL")`.

On a Windows PC, installation of a package loads it into a folder in your computer's Documents folder. It will be called something like 'Documents/R/Win-library/3.1' (for Version 3.1). It can be useful to know the location of this folder if you are updating R.

Packages may contain quite large datasets, so R assumes that you will not automatically want to load all your installed datasets into memory. To *load* a particular installed package, say ggplot2, you use the command `library(ggplot2)`. (Within functions, we use `require(ggplot2)`, which returns FALSE if the package cannot be found and allows the programmer to exit gracefully!) This makes the package's functions and datasets available. RStudio lists all the installed packages in the bottom-right pane under the Packages tab. You can access a package by ticking (clicking in) the box alongside the package name in that pane as an alternative to the `library(ggplot2)` command. You will see that a number of base packages are already ticked.

2.7.1 The base data sets

If you use the command `search()`, you will see that these packages are on R's search path, which it uses when looking for data files. To find out what's available, type `data()`. You can access data sets in these packages with the command e.g. `data(Nile)`. When this command is executed, the dataset is 'soft-loaded'. Until you actually do something with Nile, it is still not loaded into memory. Until you use it, in RStudio in the top right pane with the Environment tab active, it will appear annotated in grey with <Promise>. After typing, for example, `summary(Nile)`, into the console window, the data are loaded and the dimensions of Nile appear in the Environment pane.

```
## Using the data in the base release of R
search()
library(help=datasets)
data(Nile)
summary(Nile)
plot(Nile)
```

Notice that the `library()` function can be used to obtain help about a package as shown in this chunk.

Finally, it is worth noting that sometimes there are identical function names in different packages which have been loaded and to distinguish them, we specify the package appropriate to the function, using the operator '::'. So to if we think there is any risk of ambiguity we can refer to function `fn1()` from package `packageName` as `packageName::fn1()`..

Chapter 3

Lecture 2 - Vectors, dataframes

Brian Williams <bjw649@gmail.com>

3.1 Summary

In this lecture there will be a brief discussion of some of the important terminology and components of R - objects (vectors, data frames), functions, notation conventions etc. We will then look at a self documenting language called RMarkdown, which allows incorporation of R code directly into documentationwhich can be transformed into HTML for web pages or MS Word or PDF form.

3.2 Logistics

Go to the Code folder and open the courseRcode folder. Find "Lecture2.R". Save As.. in the myRcode folder.

As before, use mouse/menu to set the working directory to the file location: [Session->Set Working Directory->To Source File Location]. **This is an important step in every class!**

3.3 Vectors

Vectors play an important role in our data analysis because the data in each of our variables *is* a vector. A vector is a list of data which are all of the same type - numbers, character strings or factors. We can use vectors to construct dataframes or we can extract vectors from dataframes. Because they have such an important role, we'll take a brief look at how they are dealt with in R.

3.3.1 Elementary construction of vectors

The function `c(some comma-separated list of object arguments)` `combines` the arguments into a single new object (if possible!). It is one of **the most widely used of all R functions** and is used very often when you want to use a group of numbers or characters as an argument in some other function. This is one you MUST remember. Its murky origins in Unix, mean that it is also sometimes referred to as *concatenate*.

If you use `c()` to combine a list of arguments which have different classes, R tries to figure out how the user might want to combine the arguments with a common type. (Remember our standard *atomic vector* must have all elements of the same class.)

Here are some examples:

```
##Automatic variable conversion (Coercion)
#--------------------------------------------------------------------------------
A <- 11
```

```
B <- 7
C1 <- A + B
a <- c(A,B,C1)  # Concatenates or binds the arguments into a vector
a
str(a)     #  Very useful standard command in R for printing the structure of an object
d <- "Victoria"
t <- c(d,'geoff', 'Lars')
t
t <- c(t,'Stig')
t
t <- c(t,A)
t
t <- c(1,2,"Eugenie")
str(t)
#-------------------------------------------------------------------------------
```

Note that R is OK with converting the integer g to a numeric to form the four element vector, a.

Elements of vectors are accessed as well a displayed, using square brackets as we shall see below.

The operator ':' is very useful for creating vectors of simple sequences and selecting sequential elements in vectors.

It is often useful to know how long vectors are; the function `length(v)` returns the length of v.

```
##vector manipulations
#-------------------------------------------------------------------------------
length(a)
ss <- 5:22      # creates a vector, ss, whose elements are the sequence 5,6... to 22
ss
ss[6]
length(ss)
# and a little bit tricky:
sss <- ss[ss > 20]   # assigns those values of ss which are greater than 8 to a
# new vector sss
sss
sum(sss)
#-------------------------------------------------------------------------------
```

We refer to commands like `sss <- ss[ss > 20]` as 'filtering'). We will see in 5 that there is a tidyverse function called 'filter' which is used for dataframes. The function `sum()` returns the summation of the elements of the specified vector.

We'll spend more time later looking at filtering and data preparation but let's jump ahead and load up a real data-set. 'Data frame' is R jargon equivalent to a 'data set' in most other statistical software.

R has a range of objects which can store data of various kinds. The simplest is the **vector** which is simply an ordered collection of data items of a single type. If you want an important example, think of it as the observations of a variable in your data set. We will do a lot of manipulation of data and much of it involves operating with vectors.

3.3.2 Addressing vector elements

The number of items in the vector is its length. The individual item in the sixth position in a vector, `aa` can be accessed by reference to `aa[6]`.

We can access a sub-vector consisting of the 3rd, 4th and 5th components using `aa[3:5]`, or `aa[c(3, 4, 5]`. The latter can be used to access any set of indices.

Using a minus '-' sign in front of an index is used to delete that element.

```
cc <- c(1, 4, 98, 17, 43, 172, 34, 67, 58, 143, 27, 72, 44, 88, 83)
cc
cc[c(3,5,8)]
ee <- c('Tom', 'Dick', 'Harry', 'Nils', 'Lars', 'Stig', 'Hans')
ee
eMharry <- ee[-3]
eMharry
```

3.4 Data frames

A data frame is a list of variables. Each list element (variable) is a vector of equal length. The elements of the list are the columns of the data frame. The types of these vectors (the data frame columns) are not required to be the same, though, of course, within the columns each element must be the same.

We have already seen that the function `read_csv()` returns a data frame. We can also construct them from equal length vectors.

Data frames can access columns using df$colName notation and they can also use matrix-style notation. These are both demonstrated in the next script.

Where possible, it is recommended that you use the df$colName notation because it is easier to follow in scripts and also if column order is changed, the named notation still works.. As you might expect, a data frame can be constructed from a group of equal length vectors.

```
people <- c('Lena', 'Solveig', 'Guinevere', 'Hans', 'Erik' )
byear <-  c(1982,      1976,      1949,      2001,   1967)
score <-  c( 22,        43,        87,        45,     60)
salary <- c(40,         47,        31,        NA,     75)
df <- data.frame(Name = people, BirthYear = byear, Score = score, Salary = salary)
```

NOTE that this is a little confusing to look at, because the variables have been laid out across the page (rows) and in the dataframe they are placed in columns.

Here are the common ways of accessing elements of the dataframe

```
df$Name[3]    # List form - 3rd row of column 'Name'
df[2,3]       # Matrix form - 2nd row of 3rd column
df[,3]        # Row number omitted, assume ALL rows, i.e. this is equivalent to df$score
df[4:5,1:3]   # Selects the subset - rows 4 and 5, columns 1  to 3
```

Here are a few of core R's functions to investigate dataframe structure.

```
#Other basic functions -a reminder
summary(df)
names(df)
names(df)[3]
dim(df)
ncol(df)
nrow(df)
str(df)
str(people)
class(df)
```

Notice that the character vector 'people' has become a factor when incorporated into a dataframe.

3.4.1 Tibbles

Tibbles are the tidyverse equivalent of a dataframe, but they avoid some of the problems - like if you name a large dataframe in the Console, you will fill the pane with numbers. A tibble will only print the first ten row and as many columns as will fit on the screen. The printout shows the variable type, too. If you really want to see all your variables, you can use `print` and set the `width` to `Inf`.

The datasets available in the core release of R are generally data.frames. Dataframes can be converted to tibbles easily as follows:

```
df <- as_tibble(df)
class(df)
```

Finally a warning - occasionally older R functions won't work with tibbles so you may need to convert your tibble back to a data.frame. You do this as follows:

```
df <- as.data.frame(df)
class(df)
```

3.5 Type conversion

Frequently, we require conversion of types as we have just done with tibbles. For example, in preparing a heading for annotation of a plot, we might want to convert a numeric value stored in a variable to a character variable to be included in a text string.

```
plotNumber <- 3
t1 <- paste('This is plot number', as.character(plotNumber), sep = " ")
t1
```

[in fact, R will coerce the numeric to a character tytpe automatically, without the use of the `as.character` function].

Similarly, we might want to explicitly convert a character variable with value "7" to a numeric type:

```
chv <- '7'
17 + chv               # Does not work
17 + as.numeric(chv)
```

There is a large suite of 'as.type()' functions which operate in similar vein for coercion where it is possible. Type '`??as.`' (without the quotation marks) into the Console and take a look at RStudio's handy little prompt window - you'll see lots of `as.something()` functions.

3.6 R Markdown?

Markdown is formatting software used for construction of web-pages and other published documents incorporating text, tables and figures. The user creates a .Md file, which contains text, tables, figures and images with special annotations to format the document. R Markdown is a special form of the Markdown document , which in addition accommodates 'chunks' of R code.

R Markdown allows you to present your document in the following ways:

1. The chunks can simply appear in the document as highlighted code,
2. The chunks can appear highlighted in the document along with their output,
3. The chunks of code can be hidden and the output from the code presented alone,

4. The chunks can be extracted from the .Rmd file to produce a new document (.R file) containing only the code.

R Markdown uses a package called `knitr` to manage the chunks and convert an R Markdown document (with extension .Rmd) to a standard Markdown document, which is then formatted using the standard Markdown software. This all takes place seamlessly in the RStudio interface by clicking a button.

R Markdown supports easy documentation of computations so that they may be independently reproduced. This assists in collaboration with colleagues, decisionmakers, examiners and readers wanting to independently check your work. It is an excellent tool for generating **Reproducible research** and has even been used to publish a number of books using the complementary package `Rbookdown`.

3.7 R Markdown resources

RStudio provides an easy natural entry into the use of R Markdown with on-line support for a Quick Reference guide and 'Cheatsheets', both available from RStudio's `Help Menu`. These documents are also included in your Resources folder.

For more advanced use, the author of the `knitr` package, Yihui Xie, maintains a web page (blog) and has written a book [33] on `knitr`. RStudio also has a web page dedicated to R Markdown.

3.8 Starting an R Markdown document

Starting an R Markdown document is easy:

1. In RStudio, **click on File-> New File-> R Markdown**. We will use the default output choice, HTML, so...
2. **click on the OK button** in the window which has opened. Now, a 'template' of your Rmd document appears in RStudio's Edit pane.
3. **Save** the file as Homework1.Rmd in your myCode folder.
4. Then **set the working directory** to that folder.[Session -> Set Working Directory -> To Source File Location]
5. Change the title near the beginning of the document to "R Homework 1".
6. Make sure the author statement has your name correctly. Change it if necessary, keeping the double quotation marks around your name.
7. Now Click [Help -> Markdown Quick Reference] and the reference will appear in the Help Pane (bottom right),
8. ...and if you Click [Help -> -> Cheatsheets -> Markdown Cheatsheet], you get a very nice .pdf summary.

Now you're all set to write your own code and explanations into your R Markdown document.

The template contains some simple R code using one of the base R data files, and produces some simple output including a plot. The evaluated R code is shown when you generate an HTML file from this document.

When you want to see what your formatted document will look like, simply **Click** the small down arrow alongside the **Knit** button on the tool bar. A menu will pop-up, offering choices of knitting to HTML, pdf or Word doc.

Click on the HTML choice. You will see a new window pop up, containing a preview of the HTML file. If you like what you see, you can open it in your default browser by clicking on the 'Open in Browser' button and there you can save it with a right-button-click.

Try it out!

3.9 Editing your RMarkdown document

There are essentially three main components in a RMarkdown file. Firstly, there is the header section (called YAML), which is begun and finished with three hyphens ('dashes'). We've already edited this and won't say more about it. The second component is your text, tables, figures images etc which includes the special Markdown character sequences to provide formatting. Thirdly, there are the R chunks which, as we have seen above can be used to insert output into the published document with or without the R code.

Formatting text is really easy. Look at the the simple management of headings, using "#'s" to produce a heading level. (It's the second item in the Quick reference in the Help Pane, and if you want to see what it will look like, it's on the second page, under `Pandoc's Markdown` of the RmarkdownCheatSheet.pdf file) You'll also see, in each of these, how to use **bold fonts**, *italic fonts* and super- and sub-scripts.

Lists and numbered lists are also easy, the former beginning with a '*' and the latter with the number followed by a stop - for example '7.' (but without the quotation marks).

Try adding a numbered list of car brands to the document - Ford, Volvo, Audi, Volkswagen.

Chunks are distinguished by a sequence of 3 back-ticks, followed by an 'r' in squiggly brackets at the top of the chunk. At the bottom of the chunk there must be three back-ticks and nothing more. The easiest way to include a new chunk is by clicking on the green Chunks button at top right of the Edit screen, when you have an Rmd file active. On clicking, a menu appears. Select the first option (R). Alternatively, pressing simultaneously, Ctl-Alt-I, will give the same result.

There are a number of options for dealing with the chunks and these can be set for individual chunks within the chunk header (i.e. in the squiggly brackets). Chunks can be given a name, too.

Options can be set globally (i.e. options to be the default for all subsequent chunks) using the function `knitr::opts_chunk$set()`. You will see an example of this in the first chunk of your .Rmd file. Chunk options are all listed in the R Markdown Reference guide which is also in your resource folder. Some common useful options listed in the reference guide are:

- include = FALSE prevents code and results from appearing in the finished file. R Markdown still runs the code in the chunk, and the results can be used by other chunks.
- echo = FALSE prevents code, but not the results from appearing in the finished file.
- message = FALSE prevents messages that are generated by code from appearing in the finished file.
- warning = FALSE prevents warnings that are generated by code from appearing in the finished.
- fig.cap = "..." adds a caption to graphical results.

In your new Rmd file, in the second chunk (cars) insert a comma followed by 'echo = FALSE' (without the quotes), after the 'r' in the header. It should look like this:

```
```\{r cars, echo=FALSE\}
```

Now, click on the 'Knit HTML' button again and ...Voila! This time the code for this item does not appear in the HTML preview.

The various options that are available are all listed in the **R Markdown Cheat Sheet**, which you can access [Help -> Cheatsheets -> R Markdown Cheat Sheet]. You will also see in the Cheatsheet menu item the **R Markdown reference Guide**, which is an expanded version. These are both neat summaries of the information you need to run R Markdown.

Now remove ', echo=FALSE' and replace 'cars' in the squiggly brackets with 'T2Kable1'. Go to the Tutorial2.R file and copy the code in the chunk headed T2Kable1 and then paste it in your .Rmd file, replacing '`summary(cars)`'. The chunk should now look like this:

```
```\{r T2Kable1\}
library(knitr)
kable(head(iris))
kable(head(iris, 1))
```
```
```

You can test it either by clicking on the 'Knit button', or to test the chunk alone, put the cursor in the chunk and press Ctl-Alt-C (all three keys simultaneously). [You can use your mouse if you prefer. Select all the code in the chunk and then click on the 'Run' button.]

Now to insert the next chunk, put the cursor below the T2Kable1 chunk. Type some comments about the T2Kable1 chunk results if you wish, and then press Ctl-Alt-I. [You can use your mouse again if you prefer, click on the green 'C' button next to the 'Run' button.] Now paste in the next chunk from Tutorial2.R, ('T2Kable2').

So, you can use this process whenever you have a chunk to include in your document.

Now, let's look at placing a figure stored in a file into our document. Perhaps you've found a friend on the web and you would like to include his picture in your document. Here's an example of how it's done:

`image: ![](../Data/friend.jpg)`

You can copy this and paste it into your .Rmd document, knit it and have a look at my friend. If you don't like the look of him, find an image (.jpg) of a friend of your own on the web. You can paste the entire web address into the parenthesis at the end of the image statement (in place of '../Data/friend.jpg').

3.10 Tidy tables using knitr

The default RMarkdown table notation is very clumsy. A simple alternative is to use the `kable()` function. Here are some examples using the base::iris data. Copy the R chunk from Tutorial2.R into a new chunk in your .Rmd document and test it.

Here we are using one of the standard data sets - its a few measureents of different species of iris (a flower).

Try some of the others below.

```
library(knitr)
kable(head(iris))
kable(head(iris, 1))
```

3.10.1 No row names

```
kable(head(iris), row.names=TRUE)
```

3.10.2 At most 3 digits

```
#create a 5 row datframe with variables with different distributions.
kable(data.frame(Rnorm = rnorm(5), Runif = runif(5), Rbeta = rbeta(5, .5, .5), RPoisson = rpois(5, 0.5))
```

The function `rnorm()` generates random deviates. The others are what you might have guessed!

3.10.3 Alignment

By default, numeric columns are right-aligned, and other columns are left-aligned.

```
kable(head(iris), align=c('l', 'c', 'r', 'l', 'r'))
```

Chapter 4

Tutorial2 - RMarkdown practice using functions, dataframes, packages

Brian Williams <bjw649@gmail.com>

4.1 Preliminaries

Go to the Resources/courseCode folder and open file "Tutorial2.R". Save it in your myRcode folder **Set the working directory to that of the source file**. Go to the Resources folder and open the file 'RmarkdownCheatSheet.pdf'.

NOTE: The R Markdown document you create in this tutorial will be part of your submission of Assignment 1.

Chapter 5

Lecture 3 - Manipulating dataframes and factors in tidyverse

Brian Williams <bjw649@gmail.com>

5.1 Objective

To develop skill in preparing tables using dplyr and reporting in RMarkdown.

5.2 Submission

During this lecture and the following tutorial you should append new documentation to your R Markdown document. At the conclusion of the following tutorial you will submit your RMarkdown document to Cambro for assessment.

5.3 Preliminaries

Go to the Resources folder and open the courseCode.R file. Find "Lecture3.R". Copy it and save it in your myRcode folder. Now in the File menu, [File-> New File -> R Markdown] delete everything in the template Markdown except the yaml at the top. Save the file as Lecture3.Rmd in your myRcode folder. Now, as we use code in the class, copy it from the Lecture3.R file into a chunk in your .Rmd file [Click ctl-Alt-I to get a new chunk] - and then you can add comments.

You'll need to load the backpain data set for later in this laboratory session.

```
bP  <- read_csv(file="../Data/BackPain.csv")
bP <- bP %>% mutate_if(is.character, as.factor)
glimpse(bP)
```

5.4 The packages dplyr and tidyr - the core of tidyverse

These two packages are an important component of the tidyverse. The package `dplyr` was written by Hadley Wickham to try to simplify, speed-up and regularize commands and their arguments for manipulating data frames. The functions in `dplyr` operate on a dataframe (their first argument) and the output is a data frame. This allows us to 'pipe' a continuous sequence of commands with each one taking a dataframe as its first argument and passing its output (also a dataframe) to the next command in the pipe. Functions in `dplyr` are designed to form simple manipulations and in order to do more complicated tasks,

the user constructs a sequence of function calls. This is like a 'grammar' of manipulation of data objects. The package is new (2014) and still under development but has been welcomed by the R community as a potentially unifying approach. There are many different ways of manipulating dataframes in R, with inconsistent organisation of arguments and outputs. In this course, we will focus on `dplyr`.

The package `tidyr` complements `dplyr` With functions that, in the words of Hadley Wickham, 'provides a bunch of tools to help tidy up your messy datasets'.

`dplyr` contains some elementary functions which we shall see later can be used within grouped structures where they become very much more powerful. HW refers to these elementary functions as 'Single table verbs'. We will also look at some functions for combining data frames, `left()`, `bind_rows()` and `bind_cols()`.

5.5 Single table verbs

We will look at the following single table verbs:

1. `filter()`
2. `slice()`
3. `arrange()`
4. `select()`
5. `rename()`
6. `distinct()` in Section 9.4)
7. `mutate()`
8. `transmutate`
9. `count`
10. `summarise()` (can also be spelt `summarize()`)
11. We won't deal with `sample_n()` or `sample_frac()` - see help if you are interested in sampling from rows of your data frame.

Of these single table verbs, `filter`, `select` and `mutate` are the core - you will use them frequently.

5.5.1 `filter()`

With `filter()` you select **rows** of a data frame. Note the double '=' signs.

```
bPf <- filter(bP, country == 'China' ,
               residence == 'Rural', sex=='Female',
              diabetes == 'yes')
bPf
```

Soon we will want to take a closer look at the data from Ghana - so we'll use the filter to get it

```
bPGhana <- filter(bP, country == 'Ghana' )
```

...and of course to get rid of those pesky NA's:

```
bP <-  filter(bP,complete.cases(bmi,waistc,age, height))    # complete data in bmi etc
summary(bP)
```

```
bPg <- filter(bP, country =="Ghana") %>%
      select(bmi)
sum(is.na(bPg))
bPg %>% tally()
```

5.5.2 slice()

Rows can be selected by position using `slice()`.

```
NbPsl <- slice(bP, 1:5)
bPsl
bPs10 <- slice(bP,seq(10,nrow(bP), by =10))  # Select every 10th observation
```

5.5.3 arrange()

The function `arrange()` is used to reorder rows.You provide a column name to control the ordering; if you want to resolve ties, add more column names. Default is ascending order.

```
bPa <- arrange(bP,waistc)
bPa   # default is ascending order
```

Descending order is done as follows:

```
bPa <- arrange(bP,desc(waistc))
head(bPa[,'waistc'],20)    #   Here's the first 20 of them
```

5.5.4 select()

Column selection can be done using column names (always preferred) or their numerical position:

```
bPse <- select(bP,country, residence, sex, height, disability, diabetes)
bPse
kable(head(bPse))
```

Here it is using a sequence of column numbers:

```
bPsNum <- select(bP, 1:4)
bPsNum
```

You can use the ':' between named columns, too.
You can delete a column by selecting it with a minus sign in front of name or number:

```
bPsNum <- select(bPsNum, -sex)
bPsNum
```

or:

```
bPsNum <- select(bPsNum, -2)
bPsNum
```

5.5.5 rename()

Renaming columns is straightforward:

```
bPsNum <- rename(bPsNum, wealthQuantile = wealthQ)  # New name = old name
head(bPsNum)
```

5.5.6 mutate()

The function `mutate()` adds new columns which are derived from old columns:

```
bPse <- mutate(bPse, heightInches = height/2.54)
head(bPse)
```

It can also be used to change the contents of an existing column.

```
bPse <- mutate(bPse, height = height/1.0)
```

and in a single command it can change/add multiple columns:

```
bP <- mutate(bP, heightM = height/100,
               wHr = waistc/height,
               sageAge = age-50)
head(select(bP, heightM, wHr, sageAge), 10)
```

5.5.7 transmute()

If you want to discard all of the old columns, use `transmute()`:

```
bPse <-transmute(bPse, htInches = height/2.54, dis100=disability/100, country=country)
head(bPse)
```

Note how `country` was kept and that the column order is the same as the arguments.

5.5.8 count()

We can easily summarise individual factors with counts of their levels using the function `count()`. Thus:

```
count(bP,wealthQ)
 bPcc <- count(bP,country, residence, wealthQ)     # Produces long form output
```

Notice that because we have not cleaned this data, there are a significant number (94) of 'blanks' in the data.

5.5.9 base::summary()

Counting multiple factors in this way is awkward because they will be presented in long form - and with multiple factors it might be very long!. There are various ways of selecting the variables of the data frame which are factors, but there are inevitably difficulties in presenting the frequency of their levels in a convenient (tabular?) fashion. The `base::summary()` function can be used after selecting only those columns which are of interest:

```
summary(select(bP, residence, sex, wealthQ, country, agegr))
```

Selecting all factors can be done using the function `select_if()`:

```
summary(select_if(bP,is.factor))
```

5.6 Manipulating factors with `tidyverse tools`

Because we'll use factor variables a lot with `group_by` in the section below, let's take a closer look now at manipulating factors (categorical variables). We'll begin here by selecting a factor variable from the backPain data frame - country, in which the order of the 'levels' (categories) of the factor is not significant (nominal).

Now, let's have a look at 'country'.

```
select(bP, country)  #
levels(bP$country)       # The levels function can't use argument select(bP, country)
str(select(bP, country) )        #structure of cc
head(as.numeric(cc), 20)  # print numeric values of first 20 elements
#-------------------------------------------------------------------------------
```

Notice that R saves storage of factors by saving them as numerics and relating the numerics to the levels.

Often coding for levels (input codes) is abbreviated (or sometimes more lengthy than we might like for display). By default, when plotting the level names are used on the plot. If the names are not suitable, we can then define a more suitable name for the level (we 're-code' it) and then the new code name will be used in the plots and at the same time R will re-name the level in the dataframe.

There is considerable confusion about the use of 'labels' with levels and `tidyverse` offers tidier (and clearer) solutions!

Basically we want to be able to do four things:

1. Coerce numeric and/or character variables to factors where it is appropriate,
2. Re-order to something more sensible. The default alphabetical order is often unsatisfactory for presentation purposes, either in a table or a plot. Use `fct_relevel`.
3. Re-name unnecessarily terse, long or meaningless names. Use `fct_recode`
4. Cut a numeric variable into named groups to create a new factor variable. Use one of:
   - `cut_interval` which makes n groups with equal range,
   - `cut_number` which makes n groups each with approximately equal numbers of observations or
   - `cut_width`, which makes groups with a specified 'bin width'

You can see examples of the cut functions here.

5.6.1 Coercing numeric and character variables to a factor

Here is how to coerce a simple numeric variable. (In this case `comorb` is coded as the number of comorbidities, 0, 1, 2 - where 2 may be 2 or more.)

```
bP %>% select(comorb)
bP <- bP %>% mutate(comorb=as.factor(comorb))
bP %>% select(comorb)
```

To change the codes of the factors, to give us meaningful labels for presentation, we use `fct_recode`:

```
bP <- bP %>% mutate(comorb = fct_recode(comorb,
                                        "None" = "0",
                                        "One" = "1",
                                        "Two or more", "2"))
bP %>% select(comorb)
```

NOTES:

- Here we have chosen to recode (rename) all the levels, but we could have re-named only one.
- The recoding definition in the `fct_recode` function is "new name" = "old name"

We have already seen the 'blanket' conversion of all character variables in a dataframe to factors:

```
bP <- mutate_if(bP, is.character, as.factor)
```

We note that this should be done with care, because there will certainly be times when converion of character variables to factors is inappropriate - for which reason, the tidyverse dataframe functions do not automatically convert character variables to factors, which is the default in older R functions.

5.6.2 Re-ordering factor levels

To re-order, you must specify the levels in your desired order as the arguments of fct_relevel following the variable name. The spelling must be exactly as in the original!

```
levels(bP$bmi4)
 bP <- bP %>%  mutate(bmi4 = fct_relevel(bmi4, "Underweight","Normal", "Pre-Obese", "Obese"))
 levels(bP$bmi4)
```

A mis-spelling results in alphabetical order of the original levels.

5.6.3 Re-coding (renaming) categories(levels) in factors

If you simply want to change the coding of individual levels, this is easily done with `fct_recode`. Here is the call to reduce the length of the Russian Federation code (and we can deal with South Africa at the same time):

```
  ## Nominal factors - assigning Labels
bP %>% count(country)
bP <- bP %>%
  mutate(country = fct_recode(country,
                              "Russian Fed" = "Russian Federation",
                              "Sth Africa" = "South Africa") )
bP %>% count(country)
```

If you look back at the preceding script you will see that there we had the level name 'Russian Federation'. We've changed the associated level name to 'Russian Fedn'.

5.6.4 Converting numerics to factors using the `cut()` functions

The function `cut()` provides a quick way of converting numeric data to grouped factors. The `forcats` package in `tidyverse` simplifies and extends these calls a little.

Here's how to create 6 levels with approximately the same number of observations in each group:

```
bP <- bP %>% mutate(height6 =  cut_number(height,
               n = 6,
               labels = c("Very Short", "Short",
               "Average", "Tall", "Very Tall", "Extremely Tall")))
summary(bP$height6)
```

We use `cut_interval` to cut with approximately equal ranges:

```
bP <- bP %>% mutate(height4 = cut_interval(height, n=4))  #,
summary(bP$height4)

bP <- bP %>% mutate(height4 = cut_interval(height, n=4,
                                labels = c("Very Short", "Short", "Average", "Tall" )))
bP %>% count(height4)
```

.... and `cut_width` let's you choose the range (but there seems to be some minor problems with it.)

```
bP <- bP %>% mutate(height4 = cut_width(height, width = 15))  #,
summary(bP$height4)

bP <- bP %>% mutate(height4 = cut_interval(height, n=4,
                                labels = c("Very Short", "Short", "Average", "Tall" )))
bP %>% count(height4)
```

The ranges chosen here seem to be inconsistent??
If you want to choose your own breaks, you must revert to the base function `cut`:

```
bP <- bP %>% mutate(height4 = cut(height, breaks = c(0,120,150,170,Inf)))
summary(bP$height4)
# Adding nice names is easiest all in one command this way:
bP <- bP %>% mutate(height4 = cut(height, breaks = c(0,120,150,170,Inf),
                  labels = c("Very Short", "Short", "Average", "Tall" )))
summary(bP$height4)
```

..and finally, here's how to combine levels. Simply recode grouped levels to the same name:

```
summary(bP$height6)
bP <- bP %>%  mutate(h6_to_4 = fct_recode(height6,
                                          "short" = "Very Short",
                                          "short" = "Short",
                                          "very tall" = "Very Tall",
                                          "very tall" = "Extremely Tall"))

summary(bP$h6_to_4)
```

The package `forcats` is also very useful for dealing wth factors. See Chapter 15 of 'R for data science'[31] for detailed examples and extensions on the use of the functions we have seen here.

5.7 group_by()

The `summarise()` function and those we have discussed above, become much more powerful when we use grouping operations with the verb/function `group_by()`.

The other verbs are affected by grouping as follows:

- Grouped `select()` is the same as ungrouped select(), excepted that grouping variables are always retained.
- Grouped `arrange()` orders first by grouping variables
- The `slice()` function extracts rows within each group.
- The `count()` function counts the number of rows with each unique value of variable, so it is particularly useful for counting the frequency of levels in factors.
- The `summarise()` function is particularly useful when applied to grouped variables, and is explained in detail below.
- The function `n()` (takes no arguments) determines the number of observations in a group.

5.7.1 Summarising groups

In summarising groups we can add columns containing the statistics (mean, sd, max, IQR etc) for every group combination of the set specified. In our bP data, we group using factor variables. Because we will now chain `dplyr` commands together, it becomes convenient to introduce the 'piping' operator, %>%. When we place the piping operator after a data frame or a `dplyr` command, the data frame or the output data frame from the command is passed to the first argument of the command after the piping operator. Another piping operator can be placed after the second `dplyr` function, passing the output data frame from the second function to the first argument of the third function and so on. It will become clearer with examples, below.

```
library(knitr)
crs1 <- group_by(bP,country,residence, sex)
bySex <- summarise(crs1, meanDisability = mean(disability), sdDisability=sd(disability))
  bySex
kable(bySex)
```

Here's a similar example using the piping operator, %>%.

- It begins, by assigning the (not-yet-computed) result of the chained operations to the new data frame crs2.
- The data frame bP is the first input to the chain, where initially it is passed to the first function, `group_by()` which forms a data frame in which bP is grouped by country, residence and sex.
- This grouped data frame is then passed to the summarise command which forms a data frame of all combinations of the groups for which means and IQR's of disability are presented.
- Finally, the resulting data frame is assigned to crs2.

```
crs2 <- bP %>%
  group_by(country,residence, sex) %>%
  summarise(Disability = mean(disability), IQR = IQR(disability))
crs2
```

Here's a further example in which we add statistics for another variable.

```
crs2 <- bP %>%
  group_by(country,residence, sex) %>%
  summarise(Disability = mean(disability), disIQR = IQR(disability), Bmi = mean(bmi))
crs2
```

Its easy to add in another grouping variable:

```
crs2 <- bP %>%
  group_by(country,residence, sex, angina) %>%
  summarise(Disability = mean(disability), disIQR = IQR(disability), Bmi = mean(bmi))
crs2
```

Or we may want to change the order in the table:

```
crs2 <- bP %>%
  group_by(angina, country,residence, sex) %>%
  summarise(Disability = mean(disability), disIQR = IQR(disability), Bmi = mean(bmi))
crs2
```

..or filter to look at only one country:

```
crs2 <- bP %>%
  filter(country== "China")%>%
  group_by(angina, residence, sex) %>%
  summarise(Disability = mean(disability), disIQR = IQR(disability), Bmi = mean(bmi))
crs2
```

When groups vary significantly in size it is prudent to always include counts of observations:

```
aa <- group_by(bP, country, sex, residence, wealthQ) %>%
  summarise( count=n(), mdis=mean(disability))
aa
```

5.8 Save a tidied form of the dataframe bP

Let's save our tidied up form of the dataframe for subsequent use. We'l leave the NA's there, because we know we can do much better generally with `complete.cases()`. We'll just set up the factor levels tidily and add the waist-height ratio as `wHR` and we'll also convert the comorbidity variable to a factor (it was read as a numeric). Underneath it, you'll see a complex bit of code that creates a function we'll use later. (Don't worry about the code - it's very advanced!)

Notice that the final line saves `bP` and the function `count_to_pct`. When the saved file is re-loaded, both become available in the Environment.

```
bP  <- read_csv(file="../Data/BackPain.csv")%>%
        mutate_if(is.character, as.factor) %>%
        mutate(bmi4  = fct_relevel(bmi4, "Underweight","Normal", "Pre-Obese", "Obese"),
            physical = fct_relevel(physical,
                                   "low phys act", "mod phys act", "high phys act"),
            works    = fct_relevel(workS,
                       "never worked", "currently not working", "currently working" ),
            eduS     = fct_relevel(eduS,
                      "No primary", "Compl Primary", "Compl Sec/HS", "Compl Uni/Coll"),
            alcohol  = fct_recode(alcohol, "Abstainers",
                              "Drinkers" ="Non-heavy/Infreq heavy/Freq heavy drinkers" ),
            country  = fct_recode(country, 'Russian Fedn.' = 'Russian Federation',
                                           'Sth Africa' = 'South Africa'),
            comorb   = as.factor(comorb),
            comorb   = fct_recode(comorb, 'None' = '0',
```

```
                                          'One'  = '1',
                                          'Two+' = '2'),
            wHR      = waistc/height)

count_to_pct <- function(data, ..., col = n) {
    grouping_vars_expr <- quos(...)
    col_expr <- enquo(col)
    data %>%
        group_by(!!! grouping_vars_expr) %>%
        mutate(pct = (!! col_expr) / sum(!! col_expr)) %>%
        mutate(pct = 100*pct) %>%
        ungroup()
}

save(bP, count_to_pct, file="../Data/BackPain.RData")
```

In all our future classes, we can simply invoke the command below and `bP` will be instantly added to our Environment, cleaned up as above..

```
load("../Data/BackPain.RData")
```

Chapter 6

Tutorial 3 - Dplyr dataframe manipulation with RMarkdown

Brian Williams <bjw649@gmail.com>

6.1 Preliminaries

Go to the Code folder and open the R_Course.R file. Find "Tutorial3".
Set the working directory to that of the source file.

```
load("../Data/BackPain.Rdata")
```

Open your .Rmd file from Lecture 3 and add to it as you use/develop code in this tutorial.
Prepare some new tables using group_by or tabyl of variables you think may bear some relationship to back-pain.

Chapter 7

Lecture 4 - Introduction to ggplot2 basics

Brian Williams <bjw649@gmail.com>

7.1 Preliminaries

Go to the courseCode folder and open Lecture4.R file. Do "Save As.." in the myRcode folder. Then [Session->Set working Directory->To Source File Location]

We'll use the backpain data to demonstrate some of the manipulations with factors and also the use of some of the standard plotting tools. So let's begin by loading that data set. If you haven't been using Ctl-Enter to step through the R code, this lecture will be a good time to try it out. Each time you click Ctl-Enter in the R code file, you execute the code in the line where the cursor lies and then the cursor moves to the next line, ready for another click.

```
load("../Data/BackPain.Rdata")
bP <- drop_na(bP)
```

Check `bP` in the Environment pane to ensure `wHR` is there. [Click on the blue arrow.] As we've mentioned before, a blanket `drop_na()` or `na.omit()` dramatically reduces the Ghana (and Mexico) data sets.

7.2 Introduction

The package `ggplot2` is a major revision of the original ggplot package(no longer available).It is described in the second edition of Hadley Wickham's book. An early draft is included in the Resources folder (ggplot2_2016-book.pdf).

There is extensive documentation and examples here.

Winston Chang has a web-site too, with many examples. If you get stuck, the forum at ggplot2@googlegroups.com, is very active and provides rapid help. Please do pay attention to their posting protocols. You must provide minimal example codes which reproduce your problem. If you Google your problem, you'll also find a lot of answers on Stack Overflow.

7.3 ggplot2 concepts

The broad idea of graphical visualization is that we take our statistics - numbers and categories and map their values to a visual framework using plotting symbols and connections(lines) and other aggregation objects, along with other attributes such as colour, shape or size. Here's how this gets defined in `ggplot2`.

- *Data*: in the form of a data frame is usually specified with the initiation of the plot space using the function `ggplot()`.

- *Geoms*: points, lines, bars etc are the types of geometric objects that are added in layers to the plot to represent the data,
- *Aesthetics*: are visual qualities specified in the geom, such as (x,y) coordinates, line colour, shapes etc . Default aesthetics can also be specified in the call to `ggplot()`.

.

We'll begin with some simple scatter plots and then look at boxplots and finally histograms.

7.3.1 How to initiate a ggplot

Firstly, call ggplot to set up your plotting framework. Usally you will specify the data set in this call, but no plot is generated at this time. There will be a `ggplot` object created which can be assigned to a name of your choice (p?). [Note that you can also specify default x,y coordinates in this call.]

```
p <- ggplot(bP )
```

Click on the blue arrow next to `p` in the Environment pane to see what's in it. Firstly you'll see the dataframe, and underneath it lots of other plotting information. If you click on the `p` itself, you'll get a summary in the Edit pane.

7.3.2 Create a scatterplot

Now we add a (geom_point) to generate a scatter plot.

```
p <- p + geom_point(aes(x = bmi, y = waistc))
```

If you click on the `p`, you'll see that a lot more information has been added to the `ggplot`. Notice the specification of the x and y coordinates as aesthetics. This could have been done in the ggplot call. To plot the object simply enter its name.

```
p
```

Let's concentrate now on the Mexican 60-69 year group.

```
bPMex <-  filter(bP, country=="Mexico", agegr=="60-69")
```

We can easily add some colour to separate urban and rural residence..

```
p <- ggplot(bPMex) +
  geom_point(aes(x = bmi, y = wHR,  colour = residence ),size=3 )
p
```

And we can distinguish males and females by plotting them with different shapes.

```
p <- ggplot(bPMex) +
  geom_point(aes(x = bmi, y = wHR,  colour = residence, shape = sex ),size=3 )
p
```

If you want to choose your own colours, you can consult the ColorChart.pdf in the Resources folder. The aesthetics can also be specified in the ggplot arguments - so this code produces the same result:

```
p <- ggplot(bPMex, aes(x = bmi, y = wHR,  colour = residence, shape = sex)) +
  geom_point(size=3)
p
```

So here the aesthetics have been set in the `ggplot` function and then we see a textttgeom_point added with a specified size. Note that because the size has not been listed as an aesthetic, it is independent of the data. Sometimes, the size of the geom might be used to reflect a data item , in which case, it would be included in the aesthetic specification.

And now let's add a line of best fit line with confidence limits on the linear regression. The `geom_smooth` requires x,y coordinates. The preceding chunk has placed default coordinates in `p`, by specifying them in the `ggplot` call.

```
p <- p   +
  geom_smooth(method=lm, size = 1)
p
```

This plot has become much too complex and we'll need to split it up to interpret it better. We'll deal with this splitting in 11.

7.4 Boxplots

Let's have a look at some Boxplots.

With boxplots, the box is constructed with the values on the y coordinate. If a factor is specified for the x-coordinate, a box is produced for each of its levels. And if another factor is specified as an aesthetic, a boxplot is produced for each of its levels!

7.4.1 How it is done

As before we begin with a call to ggplot, specifying the data set and optionally the aesthetics. Then we add a `geom_box`, including the aesthetics, if not already included in the ggplot call, and finally we enter the name of the object to plot it. So here are the two ways of doing it:

```
p <- ggplot(bP, aes(x = residence, y = bmi,  colour = angina )) +
      geom_boxplot()
p
```

...and again with the aesthetic spcification with the geom.

```
p <- ggplot(bP) +
      geom_boxplot(aes(x = residence, y = bmi,  colour = angina ))
p
```

. and a little bit more busy and colourful...

```
p <- ggplot(bP, aes(x = eduS, y = height,  colour = wealthQ )) + geom_boxplot()
p
```

It is also possible to specify the dataset in the geom, rather than in the ggplot call:

```
p <- ggplot() +
  geom_boxplot(data = bP, aes(x = eduS, y = height,  colour = wealthQ ))
p
```

Boxplots - interaction parameter

We can easily present a number of box plots conditioned on two or more factors of interest by using the interaction parameter. Here's an example:

```
p <- ggplot(data = bP, aes(x = interaction(sex, residence, backPain30), y=bmi))
p <- p + geom_boxplot(fill = 'orange')
p
```

...and you can rearrange them....

```
p <- ggplot(data = bP, aes(x = interaction(backPain30, residence, sex), y=bmi))
p <- p + geom_boxplot(fill = 'pink')
p
```

Now let's take a look at histograms.

7.5 Histograms

It is easy to generate histograms and box plots with `ggplot`.

7.5.1 How it's done

Histograms require the specification of an x aesthetic.

```
p <- ggplot(data = bP, aes(x = bmi))
p <- p + geom_histogram()
p
```

Quite ugly! Histograms default to 30 'bins'. And the default colour needs some attention! You can specify the fill colour and the outline colour, as well as the bin width.

```
p <- ggplot(data = bP, aes(x = bmi))
p <- p + geom_histogram(binwidth = 0.25, fill = 'beige', colour = 'brown')
p
```

Specifying the number of bins is just a little bit more involved - we need to firstly compute the width for the required number of bins:

```
binsize <- diff(range(bP$bmi))/60
p <- ggplot(data = bP, aes(x = bmi))
p <- p + geom_histogram(binwidth = binsize, fill = 'beige', colour = 'brown')
p
```

You can check the documentation of `diff()` and `range()` for yourselves!
Let's superimpose the men and women's bmi histograms.

```
p <- ggplot(data = bP, aes(x = bmi, fill = sex))
p <- p + geom_histogram(binwidth = binsize,
                        position = 'identity',
                        colour   = 'brown',
                        alpha    = 0.5)
p
```

Notice firstly that there is an extra aesthetic - a histogram needs only one variable. specified as 'x'. We can colour the fill however to show other attributes.

We see two new parameters. Alpha is used in plots to make the plot semi-transparent. You choose a suitable level of transparency. Go to the ggplot docs site for the documentation on geom_histogram and learn about 'position'! Not much help there. Take a look at the on-line docs.

You may not get a clear answer there, but there are lots of nice examples. Basically, the default here is to stack the bars, `position = 'identity'` leaves them in place and we use the transparency and colour differences to distinguish them.

7.6 Over-plotted data

At the start of the lecture, we produced our first plot with many points. With `ggplot`, we can make this plot a little more accessible. We do this by making the plotted points semi-transparent, using ggplot's '`alpha`' parameter, which we saw in histograms above. We'll add a line of best fit, too.

```
p <- ggplot(bP, aes(x = bmi, y = waistc))
p <- p + geom_point(alpha=0.1)
p <- p + stat_smooth(method = lm, lwd = 2)
p
```

This plot at least gives us a better idea of where the bulk of the data lie. Most respondents have bmi's between about 18 and 30 and their wHR levels run from 0.3 to around 0.65 (just roughly eye-balling the data - use dplyr to do a quick check!). The relationship between bmi and disability is quite well defined.

Chapter 8

Tutorial 4 - Practice with ggplot2

Brian Williams <bjw649@gmail.com>

8.1 Preliminaries

```
load("../Data/BackPain.Rdata")
bP <- drop_na(bP)
```

Set the working directory to that of the source file.
Continue to develop the RMarkdown file you began in Tutorial 2.

Chapter 9

Lecture 5 - More dplyr

Tabulation, joining and merging, tidyr and wide form

Brian Williams <bjw649@gmail.com>

9.1 Preliminaries

Go to the Resources folder and open the courseCode.R file. Find "Lecture5". OPen your Markdown file and add to it as we go through the code in this lecture.

```
load("../Data/BackPain.Rdata")
bP <- drop_na(bP)
```

9.2 More `dplyr` functions

We'll begin with an artificial example, in which we firstly demonstrate the function `bind_rows()`, in creating a duplicated row in our example and then we'll use the function `distinct()` to remove it.

9.3 `bind_rows()`

When downloading large data sets, it is not uncommon for duplicate records to be present. (The SAGE data set had a significant number of duplicates in the Indian household data when it was first released.)

This problem is easily fixed using dplyr's distinct command, which is demonstrated next with a 'toy' example of some household data. We'll use the data frame to merge on household ID (hhID) in another example below.

```
# Toy example

hhID <- 1:4
hData1 <- c("X1", "X2", "X3", "X4")
hData2 <- letters[5:8]
hhDf <- data.frame(hID = as.factor(hhID),
                   hD1 = hData1,
                   hD2 = hData2)
hhDf
```

You can think of the `hhDf` as a mini-example of the 'household data' from the SAGE data set.

We begin by binding a copy of the second row to the bottom of the data frame. The function `bind_rows()` is an easy way to bind rows to a data frame. It will also bind data frames, provided both have the same number of columns.

There is an equivalent function, `bind_cols()` for binding columns to the right of the data frame.

This function will also bind a dataframe, provided each has the same number of rows.

```
hhDf <- bind_rows(hhDf,hhDf[2,])    # bind_rows adds rows of equal length below the data frame
hhDf
```

9.4 distinct()

Now let's extend this sequence a little further by constructing another related dataframe (like the SAGE individual data) to demonstrate the `left_join()` function.

The function `distinct()` removes *all* additional duplicated rows in a data frame (i.e. the first row only of a duplicate set is kept).

Let's remove the row we duplicated.

```
hhDf <- distinct(hhDf)    # Removes the subsequent duplicated rows
hhDf
```

9.5 left_join(): Merging two data frames

There is a set of functions in `dplyr` for merging data frames. Here we'll just demonstrate the `left_join()` function. We'll create a second 'toy' data set (individuals) which will relate to the first toy (household) set through the household ID (hhID) variable.

```
#
ID <- 1:15
hhID <- c(1,1,1,1,1,2,2,3,3,3,4,4,4,4,4)
iData1 <- LETTERS[1:15]
iData2 <- letters[12:26]

iDf <- data.frame(id = as.factor(ID),
                  hID = as.factor(hhID),
                  iD1 = iData1,
                  iD2 = iData2)
iDf
```

Now let's see if we can create a combined data frame in which the individual data frame rows are maintained, but have added to them the variables from the household data frame with values of those variables corresponding to the household listed in the individual's data frame.

We'll firstly try a left_join using the common household ID (hhID) as the 'key', and then we'll do it without specifying the key.

```
merged <- left_join(iDf, hhDf, by="hID")
merged
merged2 <- left_join(iDf, hhDf)
merged2
```

There is much more to dplyr than we have time to deal with here. To learn more, have a look at the introductory vignette.

9.6 Tabulation

Here we introduce a new package based on `tidyverse` called `janitor`. This package simplifies some of the basic tabulation processes that we might be familiar with from other statistical software. (We saw this briefly in Lecture 1)

9.6.1 One-way table

```
library(janitor)
c1 <- bP %>%
      tabyl(country)
c1
```

9.6.2 Two-way table

```
c2 <- bP %>%
      tabyl(country, wealthQ)
c2
```

...and some people like to use percentages:

```
c3 <- bP %>%
    tabyl(country, bmi4) %>%
    adorn_percentages("row") %>%
    adorn_pct_formatting(digits = 2)
c3
```

We'll visualize the bmi4 below.

```
c3 <- bP %>%
    tabyl(country, eduS) %>%
    adorn_percentages("row") %>%
    adorn_pct_formatting(digits = 2)
c3
```

9.6.3 Three-way tables using tabyl

```
c3 <- bP %>%
    tabyl(eduS, wealthQ, country) %>%
    adorn_percentages("row") %>%
    adorn_pct_formatting(digits = 2)
c3
```

Notice that there is a problem with the Ghana table. We'll explore that in the Tutorial.
Various kinds of data counts can also be undertaken with `group_by()`, too:
You can get overall numbers directly with `dplyr` doing something like:

```
library(knitr)
bP%>%
    count(country, agegr) %>%
  spread(key = agegr, value = n) %>%
  as.data.frame() %>%
  kable()
```

....and you can do it with a lot more categorical variables, too:

```
bP %>% count(country, sex, agegr,  wealthQ) %>%
  group_by(country, sex, agegr,  wealthQ) %>%
spread(key = wealthQ, value = n)
```

...and if you want the percentages:

```
bP%>%
    count(country, agegr) %>%
    count_to_pct() %>%
  as.data.frame() %>%
  kable()
```

9.7 select_if and summarise_if

There are a group of `verb_if` verbs which apply across the columns of a dataframe. (See the cheatsheet or use help.) We have already seen mutate_if which has been useful for converting character variables to factors. We'll demonstrate the use of `select_if and summarise_if` in the next couple of chunks.

Let's summarise all the numeric variables in our backPain data frame.

```
summarise_if(bP, is.numeric,list(
             Mean   = mean,
             Median = median,
             SD     = sd,
             IQR    = IQR,
             MAX    = max,
             MIN    = min))
```

The output is difficult to read and needs some clever tidying. It can all be done in the tidyverse, but we will have to make use of the `tidyr::gather()` function to convert the summary to *long form*, which is all explained in the next section. We don't have time to deal with the functions `tidyr::separate()` and `tidyr::spread()`, but if you have messy data - you might like to check Hadley's Chapter 9 in his web version of the book "R for data science"[31].

```
# Modified from Stack Overflow #

bP_sum <- bP_sum %>% gather(stat, val) %>%                      # tidyr
  separate(stat, into = c("var", "stat"), sep = "_") %>%        # tidyr
  spread(stat, val) %>%                                         # another tidyr function
  select(var, min, q25, median, q75, max, mean, sd)             # reorder columns
bP_sum
```

Because there may be different numbers of levels, factors are also handled more easily in `long form`. Here is the code to do the job, but you'll need to read the next section to see what long form is and how the function `gather()` works. We'll take a look at code for our bP dataset at the end of the next section.

```
bPFactors <- bP %>%
  select_if(is.factor) %>%
  gather(name,value) %>%  # reshape datset
  count(name, value)      # count combinations
as.data.frame(bPFactors)  # There' a bug in this!!!!!!!!!!!!
```

I still prefer to use the core function `base::summary`!

9.8 Wide form and long form - `tidyr`

In this section we are going to consider the format in which our data is presented. The reason for this is that many statistical analyses are not well suited to dealing with data in the standard *wide* form and many require or are much easier to use in *long* form. So we would like to explore the reorganization of our data into this form.

Firstly, though we'll take a minute or so to consider the type of data we have in our dataframe.

9.8.1 Some considerations of the type of data

Data can be represented in many tabular forms and in considering these it is usual to separate the data items themselves into two groups, namely identifier variables and measured variables.

The identifier variables relate to the study design. The locations, the sex of the individuals and their age are often termed the demographic variables and aside from age, these variables are categorical in nature. If age groups are defined or age is constrained, then all of these demographic variables are categorical and furthermore, they might generally be assumed to be accurate.

Measured variables result in a spread of data with arbitrary resolution. Height, for example might be measured in centimetres with a resolution of 1 centimetre and hence we might see integers running from 80 to 200 or so. Blood pressure measurements would require two values, timed exercises might be measured to the nearest second, etc. Because another person is involved in the measurement, this data might also be considered to be generally accurate. Although resolution limits the data to discrete values, there will generally be far too many for them to be useful as categories.

There is a third group of variables in survey data; this is the data resulting from multiple choice questions - this data is clearly categorical in nature and more prone to errors.

In describing individuals, we can use the identifier variables and other socio-economic factors about which questions have been asked.

In some circumstances, the identifier variables and socio-economic factors could provide an unambiguous description of a subject and in a true database, the primary key variable alone does exactly that. In our data frame, with anonymous data, how

In broad terms we can think of the identifier variables and socio-economic variables as condition or cause or environment and measured variables as some outcome of these conditions - but these definitions are not clear because many of the causal relationships are circular. Fred smokes because he is depressed, but he is depressed that he can't stop smoking.

This little discussion leads to the fact that it depends on our objectives as to how we group and present data which is correlated in some way. Are we interested in showing differences between countries - differences in disability between wealth groups etc etc. The choice of objective and the chosen visualizations are left to the presenter.

Let's return to the primary subject here: long and wide form.

9.8.2 Converting wide form to long form

The 'standard' data frame like the Back Pain one we are using, essentially comprises rows which are 'observations' of all data pertaining to one surveyed person. This is called *wide form*. It is *tidy* in Hadley Wickham's terms because it has all the observations on one subject in a single row and there is only one variable in each column.

It turns out that there is an equivalent form of organising the data in terms of identifiers but with the 'measured' variables all in one column This form of data presentation is called *long form*.

In long form, all variables designated as 'measured' are collected into two columns, one in which the column entries identify the variable name of the measurement and in the second column the adjacent entry has the measured value. Thus if we have four measured values, their columns will be removed and replaced by columns called something like 'measurement type' and 'measurement value' and there will be four times as many rows, since we now require a row for each of the measurements for each individual.

You will have noticed that in addition to our 'cutting' height to produce a binned variable height4, bmi has already been binned to bmi4, where the bins conform to standard definitions of body condition. We may use bmi4 as one of a number of factors influencing disability, or we might be interested in the relationship between bmi and disability as they are influenced by some other factors.

Let's have a look at this reorganization. We need to do a bit of tidying up first. Looking ahead, for nice plots, we'll need to scale the waist-height ratio variable `wHR` by multiplying it by 100.

We can do easily using the `tidyr` package, which is included in `tidyverse`.

The function in the `tidyr` package which converts wide form to long form is called `gather()`.

The calling arguments for `gather()` are straightforward. Firstly, specify the wide form dataframe, Then specify a name for the 'key' variable and another name for the column with the values associated with the keys. Next, specify the columns to be gathered into the key and value columns. Here is the specification of the columns to be 'gathered' from the documentation:

> A selection of columns. If empty, all variables are selected. You can supply bare variable names, select all variables between x and z with x:z, exclude y with -y. For more options, see the dplyr::select() documentation. See also the section on selection rules below.

We'll choose disability, bmi and waistHtRatio as the measured variables.

[I was confused for a while by the '-y' option which only applies if you are specifying a sequence of variables (x:z) and want to omit one of the sequence. See the help pages on `gather()` for complete definition of arguments and examples.]

There is one other limitation in the selection process and that is that the measured variables must all be of the same type because in their single column in the long data frame, only one type will be allowed.

```
bPgather <- bP %>%
             mutate( wHR=100*wHR) %>%                              # Multiply wHR by 100
             gather(key = variable, value = value, disability, bmi, wHR)      # Convert to long form
```

The dataframe `bPgather` is large so take a quick look at it by clicking on its name in the enviroment pane.

Now let's see what ggplot can do with a long form dataframe!

```
library(ggplot2)
p <- ggplot(bPgather, aes(x = interaction(country,sex), y = value, colour=variable))
p <- p + geom_boxplot()
p
```

```
library(ggplot2)
p <- ggplot(bPgather, aes(x = country, y = value, colour=variable))
p <- p + geom_boxplot()
p
```

```
library(ggplot2)
p <- ggplot(bPgather, aes(x = eduS, y = value, colour=variable))
p <- p + geom_boxplot()
p
```

```
library(ggplot2)
p <- ggplot(bPgather, aes(x = wealthQ, y = value, colour=variable))
p <- p + geom_boxplot()
p
```

```
library(ggplot2)
p <- ggplot(bPgather, aes(x = physical, y = value, colour=variable))
p <- p + geom_boxplot()
p
```

Chapter 10

Tutorial 5 - Practice with the dplyr family

Brian Williams <bjw649@gmail.com>

10.1 Logistics

Go to the Code folder and open the R_Course.R file. Find "Tutorial5".

Set the working directory to that of the source file.

You will use the WHO SAGE back pain data set in this tutorial.

10.2 Another look at Ghana's data and NA's

Let's load the data and take a quick look at the Ghana data as we drop the NA's. Remember we had trouble with the tabulation of ghana's educational status vs. wealth status.

```
library(tidyverse)
load("../Data/BackPain.Rdata")
bP %>% filter(country == "Ghana") %>% summary()
bP <- bP %>%
             drop_na()
bP %>% filter(country == "Ghana") %>% summary()
```

OK - there are some big problems here. We now have only 38 subjects in the Ghana data. Let's try it again, without dropping the NA's and then using complete cases.

```
library(tidyverse)
load("../Data/BackPain.Rdata")
bP <- bP %>%
  filter(complete.cases(eduS, wealthQ)) %>%
  filter(country == "Ghana")
summary(bP)
```

Much more data, but the distribution of wealth is surprising - we'll take a look at that later, too.

```
library(janitor)
c2 <- bP %>%
    tabyl(eduS, wealthQ) %>%
    adorn_percentages("row") %>%
    adorn_pct_formatting(digits = 2)
c2
```

OK. Now let's do a comparison of the wealth distributions in the 6 SAGE countries.

```
load("../Data/BackPain.Rdata")
by_country <- group_by(bP, country)
sumry <- summarise(by_country,
                   waistcM = mean(waistc, na.rm=TRUE),
                   waistcSD = sd(waistc, na.rm=TRUE),
                   heightM  = mean(height, na.rm=TRUE),
                   heightSD = sd(height, na.rm=TRUE),
                   bmiM     = mean(bmi, na.rm=TRUE),
                   bmiSD    = sd(bmi, na.rm=TRUE))
sumry
```

```
library(dplyr)
bmisum <- group_by(bP, country)
bmiM <- summarise(bmisum,bmiM = mean(bmi, na.rm=TRUE))
bmiM
```

Chapter 11

Lecture 6 - More ggplot2, facets, time series

Brian Williams <bjw649@gmail.com>

11.1 Summary

This session extends the explanations of the use of the ggplot2 package for graphical presentation of data facets.

11.2 Logistics

Go to the Code folder and open the courseCode.R file. Find "Lecture6.R". Set the working directory to that of the source file. Take notes in your RMarkdown file (Due for submission on Wednesday!)

It is also a good time to **install** the `WDI` package and also **install** the package `gridExtra`.

```
bP<- read_csv(file="../Data/BackPain.csv", na = c("","  ","NA"), col_types = cols())  %>%
 drop_na()
```

11.3 Introduction

We have seen in Section 9.8.2 that `tidyr` has a neat function for converting so called 'wide' forms of data frames (like the back pain data we have been looking at) into 'long' form. This will lead us nicely into `ggplot2`, which likes (but does not require) long form!

In this session we will have a look at ggplot's impressive capacity to do multi-variable conditioned plots called 'facets'. We shall see that these plots provide very good capacity to look at data in multi-conditioned groups.

Before we begin, however, let's see how to easily put two or more plots together on the page.

11.4 Multiple ggplots in a window

Sometimes you may want to put different types of `ggplot`'s in separate panels in a single window. It seems to be a bit less common in ggplot, because of the capacity to do over-plotting and facet plotting which we'll see below.

The procedure is straightforward but you firstly need to have installed the package `gridExtra`.

```
library(gridExtra)

p1 <- ggplot(data = bP, aes(x = bmi, fill = sex))
binsize <- diff(range(bP$bmi))/60
p1 <- p1 + geom_histogram(binwidth = binsize,
                          position = 'identity',
                          colour   = 'brown',
                          alpha    = 0.5)

p2 <- ggplot(data = bP, aes(x = sex, y = bmi))
p2 <- p2 + geom_boxplot(fill = 'red')

grid.arrange(p1,p2,ncol = 2)
```

11.5 Using facets in ggplot

Facets are a useful way of presenting multiple correlated factors (without the total confusion arising by putting all the information on a single plot.)

Let's plot some histograms of the numbers for sex and age group of the respondents for all the SAGE countries.

```
pp1 <- ggplot(bP, aes(x=bmi, fill=agegr)) +
  geom_histogram(binwidth = 1.) +
  facet_grid(country ~ sex,scales="free_y")
pp1
```

Note the 'free' y-axis. Its not obvious at first glance that China has much more data than South Africa. An annotation on the plot of the total numbers would draw immediate attention to this.

We took a look at that in the last Tutorial.

Many of Ghana's `NA`'s are in the height variable so let's use `complete.cases()` to clear the rows which have `NA`'s in the data we are looking at - but not remove them if there are NA's in other variables in those rows.

```
load("../Data/BackPain.Rdata")
bP%>%group_by(country)%>%
  tally()
bP <- filter(bP, complete.cases(
                 residence, disability, sex, age, country, bmi4))

bP%>%group_by(country)%>%
  tally()
```

Now try the facet plot again

```
pp1 <- ggplot(bP, aes(x=bmi, fill=agegr)) +
  geom_histogram(binwidth = 1.) +
  facet_grid(country ~ sex,scales="free_y")
pp1
```

Here is another more complicated example, looking at the data for disability vs age.

```
p <- ggplot(bP, aes(x = age,  y = disability, color = residence, shape = sex))
p <- p + geom_point()  + stat_smooth(method = lm, se = FALSE)
p <- p + facet_grid(country~bmi4)
p
```

Notice that Russia and Mexico still have limited data in the Underweight category - and perhaps we should abbreviate the Russian Federation's name even more! Trend lines are consistent everywhere except the very small dataset for underweight Mexicans and Russians.

Note that we've specified that we don't want the standard error plotted. In this case it's so wide that the scales of the plot are reduced to accommodate it and we lose too much resolution!

We do not have a lot of underweight Russians and Mexicans - how many? There are many ways to compute these numbers - the chunks below is one - can you think of another (better?) one?

```
bbR <- bP %>%
  filter(country == "Russian Fedn." ) %>%
  count(bmi4) %>%
 rename( nRussia = n)      #Missing code (incuding preceding %>%)
bbR
```

```
bbM <- bP %>%
  filter( country == "Mexico")  %>%
  count(bmi4)%>%
  rename(nMexico = n)     #Missing code (incuding preceding %>%)
bbM
bbb <- left_join(bbR,bbM)  #default key!!
bbb
```

In the tutorial, try out the facet plot for yourselves, using something like: `x = bmi, y = disability, color = sex, shape = smoke` in a facet_grid country $\sim$ wealthQ. Copy and paste from above to begin with - then you'll have to use the `complete.cases()` function to pick up the `NA`'s in the newly selected variables.

11.5.1 Using long form with facets

In Section 9.8.2 we saw that we could arrange our data in 'long' form using the `tidyr` package, `gather`, and that `ggplot` could then be easily used to produce a group of box-plots, three for each country. The code to gather our tidied up data frame and plot box-plots with `ggplot` is given in the chunk below.

```
# Now gather the reduced version of bP on 3 measured variables
library(tidyr)
bPg <- gather(bP, variable, value = value, age, disability, bmi)
dim(bPg)      # We're expecting 3* bP's rows
str(bPg)      # Much better

levels(bPg$variable)

p <- ggplot(na.omit(bPg), aes(x = country, y = value, colour=variable))
p + geom_boxplot()
```

This chunk also demonstrates how to adjust the range on the y-axis.

```
p <- p + geom_boxplot() + ylim(0,90)
p
p <- p + facet_grid(residence~sex)
p
```

These plots contain too much information for the average viewer, but are presented to show the kind of plots which can be constructed with minimal effort, using ggplot.

11.6 Time series using gather

An important application of `gather` in `ggplot` is in multivariate time-series plots. Since our standard BackPain data is cross-sectional, we will need to find another data-set to demonstrate how `gather` and `ggplot` work together to produce these multivariate plots. Let's have a look at getting some of the World Bank Data on World Development Indicators.

11.6.1 Getting World Development Indicators (World Bank)

Take a quick look at the description file of the WDI package and you will see that its latest data is April, 2018.

The World Bank's indicators help page has descriptions of how the codes are derived and a link to a spreadsheet listing the available time series and their codes. This spreadsheet (WDI_CETS.xls) is in the Resources folder. Have a look at it. In the spreadsheet, you can look up (for example) *Gross national income per capita in current International dollars* and you will see that the WDI code for this item is 'NY.GNP.PCAP.PP.CD'.

Let's also have a look at mobile phone ownership. How do these data relate to public health? *Mobile cellular subscriptions (per 100 people)* has the code 'IT.CEL.SETS.P2'.

'Health expenditure, total (% of GDP)' is indicated by 'SH.XPD.TOTL.ZS'.

The code chunk below shows how to download these three items. Have a look at the help documents for definitive information about the arguments of the function `WDI()`.

Let's begin by taking a look at the 'indicators'.

```
library(WDI)
WDIsearch(string='gnp', field='indicator', cache=NULL)
```

These indicators are used to identify the data you are seeking to access.

```
WDIsearch(string='mobile', field='name', cache=NULL)
```

Item 24 in this list shows us that the indicator "NY.GNP.PCAP.PP.CD" is the code for access to "GNI per capita, PPP (current international $)" .

```
SAGE_PPP <- WDI(country = c("CN", "GH", "IN", "MX", "RU", "ZA") ,
                indicator = 'NY.GNP.PCAP.PP.CD',
    start = 1985, end = 2017, extra = FALSE, cache = NULL)   #GNP
otherC <- c("AU", "DE","IR", "SE", "VN" )
SAGE_MPH <-  WDI(country = c("CN", "GH", "IN", "MX", "RU", "ZA") ,
                 indicator = 'IT.CEL.SETS.P2',
    start = 1985, end = 2017, extra = FALSE, cache = NULL)      #Mob Phones
SAGE_HPG <-  WDI(country = c("CN", "GH", "IN", "MX", "RU", "ZA") ,
                 indicator = 'SH.XPD.CHEX.PP.CD',          # Code not in WDI_CETS.xls !?
    start = 1985, end = 2017, extra = FALSE, cache = NULL)
str(SAGE_PPP)
head(SAGE_PPP,40)
names(SAGE_PPP)[3] <- "PPP"
names(SAGE_MPH)[3] <- "MPH"
names(SAGE_HPG)[3] <- "HPG"
SAGE_HPG <- SAGE_HPG %>% mutate(HPG = HPG * 10.)  # NB (Easier to see in the plot)  !!!!!!
sum(is.na(SAGE_MPH))
sum(is.na(SAGE_HPG))
```

11.6.2 Joining the data

Now we need to join the three data frames into a single data frame. This will happen fairly painlessly, since we chose the same number of years of data and the same number of countries in each of the three downloads. The function `left_join` will join all columns with the same names (in our case, year, iso2c and country) and add the new columns. We repeat the process with the third data frame, joining it with the result of the first join.

```
df <- left_join(SAGE_PPP, SAGE_MPH, sort = FALSE)
df <- left_join(df, SAGE_HPG, sort = FALSE)
df$country <- factor(df$country)
#!!!!!! Abbreviating RF and SA improves the plot a little
df <- df %>% mutate(country = factor(country, labels = c('China', 'Ghana',
                                'India', 'Mexico', 'Russian Fed.', 'Sth Africa')))
head(df)
head(SAGE_PPP)
head(SAGE_MPH)
head(SAGE_HPG)

#-------------------------------------------------------------------------------



jj <- 0
dd <- rep(0,6)      # create a 6 element vector, each element is 0

# Normalize per capita income for plotting (divide each year's per capita income
#      data by the max of the country's per capita income data)

for (i in levels(df$country)){        # loop through the levels of country
  jj <- jj + 1                        # jj will take values 1-6 as we cycle through the loop
  dd[jj] <- max(df$PPP[df$country == i], na.rm=T)   # Find the maximum PPP for country i
                                                    # Store in vector dd
  df$PPP[df$country == i] <- df$PPP[df$country==i]/dd[jj]*100. # divide PPP for country i
}
df <- select(df, -iso2c)    # Cleaning up - don't need this
```

At this stage you should do a quick check to ensure that things have worked out the way you want. It can be a little bit tricky if the data are reordered in the merging process, so we specified `sort = FALSE`, so that would not happen. (Try it yourself with the default [`TRUE`] setting for sort. Then look at the head of each of the data frames). We've normalized the per capita incomes so that we'll have better scales for our plots of the three variables.

11.6.3 Gathering the data frame

Now we're ready to gather the data frames and produce some interesting results.

```
tsg <- gather(df, variable, value, PPP, MPH, HPG)      # !!!!!!!!Note removal of -ve's

head(tsg,50)
```

Notes

- Here we have gathered so that year and country are held constant as we put the three measured variables, "PPP","MPH" and "HPG" on separate rows of the long form data frame. No names have

been specified for the long form so we have a new factor variable called 'variable', with levels "PPP","MPH"and "HPG" and their values appear in the new numeric variable called 'value'.

11.6.4 Construct a new data frame for the annotation in each facet

```
tdf <- data.frame( country = levels(df$country),                    #1.
                   x1 = rep(2000, 6),
                   y1 = rep(Inf, 6),                                #2.
                   text1 = paste("PPP (max) Normalized to 100 = ", as.character(dd)))
tdf$text1 <- as.character(tdf$text1)                                #3.
tdf
```

Notes

1. In order to place separate (different) text in each of the facets, we need to set up an extra data frame containing: the level(s) of the factor(s) in each of the facets. This will identify the facet for insertion of the text. If two factors (factor1, factor2) are used then this data frame will have factor1*factor2 rows and the first column will have the levels of factor1 repeated length(factor2) times [(rep(levels(factor1),length(factor2)))] and the second column will have the levels of factor2 repeated in the rows of each of the blocks in the column of the factor1 levels [rep(levels(factor2), each = length(factor(1))].

   Our example above simply requires a column with the levels of our single factor (country). The next two columns will be the the (x,y) locations of the texts in the coordinates of (factor1, factor2) (i.e. corresponding to the factor1, factor2 facet). The last column is the text to go into each of the facets.

2. Inf sets the y coordinate to the upper limit of the y-axis.

3. Make sure this hasn't been coerced to a factor!

11.6.5 Facet plot with individual annotations

Annotating individual facets is a little bit tricky, as you saw above, but once the text data frame is set up the rest is reasonably straight-forward.

```
p <- ggplot(data = tsg,
            aes(x = year, y = value, colour = variable))            #1.
p <- p + geom_line() + facet_grid(country ~ ., scale = "free_y")    #2.
p + geom_text(data = tdf, aes(x = x1, y = Inf, label = text1),      #3.
              vjust=2, inherit.aes = FALSE)
```

Notes

1. Here's the set up of the ggplot object - we'll get a plot of the variables (PPP, MPH and PVT) in different colours.

2. The plot will be different coloured lines for each of the three variables and there will be a single column of 6 facets, one for each country and the 'y' scale in each will be free to find its own values.

3. ...and finally the annotation using our extra data frame. In order to assume there is no confusion with our earlier aesthetics, we set `inherit.aes = FALSE`. The parameter `vjust = 2` brings the text down from the upper limit of the y axis. I can't find any documentation on vjust > 1 on the web, but this works nicely! Here is a nice 'cheat-sheet' for making little adjustments to tidy up plots - it includes values of vjust > 1.

11.7 Annotating faceted scatter plots with best fit equation

This example takes our last annotation example a couple of steps further. We will see that we have a two column array of facets, that the assembly of the supplementary data frame is handled by `ddply` and a separate function has been created for dealing with the presentation of the lines of best fit. The code you see here is a modification of that used in Winston Chang's book (which does not work for facet plots). I think my code is mostly from this example on stackoverflow.com.

We'll also show how to send the plot directly to a file.

We don't have time to talk in detail about the presentation of mathematical formulae using `ggplot`, but as you might by now expect, the tools are all there. For the most part ggplot, in its text geoms, uses formats from base R's `plotmath` with the option `parse = TRUE`. Try `?plotmath` and `demo(plotmath)`.

```
lm_eqn = function(bb){
  m = lm(wHR ~ bmi, bb);# Note: the function here has not been generalised
                        # with arguments for different variable names
  eq <- substitute(italic(y) == a + b %.% italic(x)*","~~italic(r)^2~"="~r2,
                   list(a = format(coef(m)[1], digits = 2),
                        b = format(coef(m)[2], digits = 2),
                        r2 = format(summary(m)$r.squared, digits = 3)))
  as.character(as.expression(eq));
}
```

In the next chunk, we begin by grabbing plyr from the library and then setting our graphical output to be saved in .png format in a file called `"plot9.png"`. We set the width and height in mm and the resolution in the computer standard graphics unit, dots per inch. Similar functions are available for .bmp, .jpeg, and .tiff devices. Note that at the conclusion of the printing of our graphic we must specify `dev.off()` in order to close the file and return subsequent graphic output to the screen.

```
library(plyr)
png("plot8.png", width = 2480, height = 1240, res = 120)
pp4 <- ggplot(bP)
pp4 <- pp4 + aes(x = bmi, y = wHR, shape = residence, colour = agegr, group = 1)
pp4 <- pp4 + geom_point()
pp4 <- pp4 + stat_smooth(method = lm,level = 0.99) # assumes formula = y ~ x
pp4 <- pp4 + facet_grid(country ~ sex)
pp4 <- pp4 + labs(title = "Compare bmi vs waistHtRaio")
pp4 <- pp4 + ylab("waistHtRatio*100")
pp4 <- pp4 + xlab("BMI")
eqnDF <- ddply(.data = bP, .(country,sex), lm_eqn)                              #1.
pp4 <- pp4 + geom_text(data = eqnDF,aes(x = 40,y = 180,label = V1),
                       parse = TRUE,inherit.aes = FALSE)
print(pp4)
dev.off()
```

This `ddply` command constructs a data frame consisting of all the combinations of the levels of `country` and `sex` and passes the corresponding subset of `bb` to the function `lm_eqn`, where a best fit line is constructed and a character string of the equation of the best fit line is added to that row of the new data frame. When the combinations of the levels in each row have been passed to `lm_eqn` and the text string added to the data frame, the latter is assigned to `eqnDF`.

A further search on leads to an easier way to do this, but firstly requires that you install the package `ggpmisc`.

```
library(ggpmisc)
png("plot8.png", width = 2480, height = 1240, res = 120)
```

```
my.formula <- y ~ x
p <- ggplot(data = bP, aes(x = bmi, y = wHR, shape = residence, colour = agegr, group = 1)) +
  geom_smooth(method = "lm", se=FALSE) + ylim(0,1.5) +
  stat_poly_eq(formula = my.formula,
               aes(label = paste(..eq.label.., ..rr.label.., sep = "~~~",x=30, y=1.4), y=1.4),
               parse = TRUE) +
  geom_point()+ facet_grid(country~sex)
p
dev.off()
```

... and a bit more information here.

At first glance the fits are good, but look at the r^2 values and then remind yourself of the locations of the most dense part of the data set and then think about how well defined the slopes are in those dense regions. Even though the high bmi readings support the slopes, there is so little data there (comparatively), its impact on the r^2 is small.

The package `ggpmisc` is well worth looking at if you are trying to do complex things with `ggplot` - in addition to this help with equation annotation, it can annotate peaks and troughs in your data, deal with time series, annotate plots with tables such as ANOVA etc.

Chapter 12

Tutorial 6 - Practice with tidyr and dplyr extensions

Brian Williams <bjw649@gmail.com>

12.1 Preliminaries

Go to the Code folder and open the courseCode.R file. Find "Tutorial6".
Set the working directory to that of the source file. Load your Rmarkdown file.

```
load("../Data/BackPain.Rdata")
bP <- drop_na(bP)
```

Add some plots from the lecture to your Markdown file.

Chapter 13

Lecture 7 - Elementary statistics and Regression

Brian Williams <bjw649@gmail.com>

13.1 Preliminaries

Install the `car` package. This is a very useful package written by John Fox. The name is from the title of his book "An R companion to applied regression". Although the code does not use `tidyverse`, I think it's a good book for people to use to learn how to use R for regression analysis (but I'm not a statistician!).

Load bP if you've not done so already, and go to the courseCode folder and open the Lecture7.R file. Save it in your myRcode folder and set the working directory to that of the source file.

```
# Initial read of back pain data set
load("../Data/Backpain.Rdata")
```

13.2 Objective

To undertake some basic data maniplation and statistical analysis and prepare a report on this work using RMarkdown.

BE WARNED: This data set will not produce neat regressions of the kind you will see in most textbooks, unless we choose directly related data like bmi and waistHeightRatio.

13.3 Submission

There are no submission requirements for this class, but the document prepared here and in Tutorial 7 will form the basis of a document for submission in Tutorial 8. It is essential that you create and develop an RMarkdown document in this class and this afternoon's tutorial to take to tomorrow's sessions. Begin a new R Markdown document with title "R Homework 2" and make sure your name is correct. Save the document as your_name_H2.rmd.

13.4 Introduction

R is known for its many statistical packages and some have been tried and tested for many years. The core statistical modeling is described in detail in R-Intro.pdf, Chapter 11, "Statistical models in R". This Lecture provides a brief introduction to this core modeling. There's also documentation for lm here. You need to

have a good look at this if you want to do anything that is at all unusual. (Google is still your best friend for examples etc).

We'll begin by taking a look at a few basic statistics using the base functions from R. Firstly a correlation matrix between two numeric (continuous) variables:

13.5 Correlation matrix and tests of numeric variables

13.5.1 Normality of numeric data

ggplot::qq_geom)

It's easy to visualize normality of data with ggplot. You can plot histograms, as we have done before or you can do a qqplot.

```
p <- ggplot(bP, aes(sample=age)) +
 stat_qq() + stat_qq_line()
p
```

Age is of course, not normally distributed. We can also use a couple of core functions to compute statistical indicators of normality and comparisons of distributions. These do the classical Kolmogorov-Smirnov (`ks.test()`) and Shapiro-Wilks (`shapiro.test()`) tests.

Here's the function `shapiro.test()` applied to a sliced section of the data (It doesn't work if n > 3000):

```
shapiro.test(bP$bmi[1:2500])
```

13.5.2 Correlation of numeric data

The base function is called `cor()` and is simply called with the required vectors or dataframe. You can choose the method between "pearson" (default), "kendall" or "spearman" .

```
bP %>% select(bmi, waistc) %>%
  cor(use = "complete.obs")   #removes NA's where necessary
```

This is easily extended to more variables:

```
bP %>% select(bmi, waistc, age) %>%
  cor(use = "complete.obs")   #removes NA's where necessary
```

...and if you are enthusiastic, you can look at all the numerics:

```
bP %>% select_if(is.numeric) %>% cor(use = "complete.obs")   #removes NA's where necessary
```

```
bP %>% select(bmi, age, disability) %>%
  cor(use = "complete.obs")   #removes NA's where necessary
```

We might also be interested in the correlation test statistics. These are determined by the `stats::cor.test()` function. `stats::cor.test()` requires specification of two vectors, and a dataframe cannot be specified, so...

```
  cor.test(bP$age, bP$disability, use = "complete.obs")
```

There's only a weak correlation between age and disability (as defined in the survey and by WHO's algorithm). It's a little difficult to show on a plot because the data has a discrete structure. Jitter and alpha help a little but there are still clearly evident bands in the age data.

```
ALPHA <- 0.1
TITLE <- paste("Jittered plot of disability vs age with alpha = ", as.character(ALPHA))
bP %>%
  select(age, disability) %>%
  ggplot(aes(x=age, y = disability)) +
  geom_jitter(alpha = ALPHA) +
  xlim(50, 100) +
  geom_smooth(method = "lm", se = FALSE) +
  ggtitle(TITLE)
```

13.6 Contingency tables and χ^2 test

13.6.1 Tables

We have already seen some tabulation using the tabyl function - here's a little more before we look at the function `chisq.test()`.

```
library(janitor)
bP %>%
    tabyl(agegr, comorb) %>%
    adorn_title()
```

(Note that the use of (adorn_title())) adds the title of the column variables, but the output is no longer a simple dataframe.)

```
    bP %>%
    tabyl(country, agegr) %>%
    adorn_percentages("row") %>%
    adorn_pct_formatting(digits = 1, affix_sign = FALSE ) %>%
    adorn_title()
```

Three-way cross-tabulation is also sometimes of interest - it can be done with `tabyl()`:

```
tabyl(bP, sex, backPain30, agegr) %>%
  adorn_title()
```

... and remove the NA's first:

```
bP %>%
  filter(complete.cases(sex, backPain30, agegr)) %>%
  tabyl( sex, backPain30, agegr) %>%
  adorn_title()
```

13.6.2 χ^2 tests

Chi-squared tests are most easily undertaken using the function `base::chisq.test()`. The implementation is not 'tidy', but it's not difficult anyway.

```
ch_age_edu <- chisq.test(bP$agegr, bP$eduS)
ch_age_edu
```

The results indicate a strong association - perhaps we should take a closer look at the test statistics and data?

Let's take a look at the object created by `base::chisq.test()`. We'll check its class and its structure, and then figure out how we might extract individual statistics from the output:

```
class(ch_age_edu)
str(ch_age_edu)
```

The first thing we see is that the output is an object of type `htest`.

The output from the `str()` function indicates that the `htest` object is a `list` with 9 items. (Many functions have `list` outputs). The dataframes we have been working with are a special form of a list - each variable is a list item of equal length and is accessed with the '$' symbol. You can see the `str()` function reports the objects contents in summary form with a '$' symbol in front of each item - so it is accessed like we accessed variables in data frames.

For example, we can get each of the 'observed', 'expected' and 'residuals' values by naming them as follows:

```
ch_age_edu$observed
ch_age_edu$expected
ch_age_edu$residuals
```

...and, of course we can tabulate percentages as we did earlier.

```
  bP %>%
  filter(complete.cases(agegr, eduS)) %>%
  tabyl(agegr, eduS) %>%
  adorn_totals("row") %>%
  adorn_percentages() %>%
  adorn_pct_formatting(digits = 1, affix_sign = FALSE) %>%
  adorn_title()
```

Well, that is impressive! The educational levels in these countries have improved strongly in the younger age groups.

13.7 Facet plots

Facet plots also provide a useful visualization of the data:

```
ggplot(bP, aes(x= age, y = disability, colour = sex)) +
  geom_point() +
  facet_wrap(~country, nrow = 2)
```

13.8 Linear Regression

In base R, linear regression is undertaken with the core function `stats::lm()`. The model is described using a *formula*, beginning with a specification of the dependent variable, followed by '∼' and then the independent variables. In the examples below, the independent variables are all listed with a '+' sign. There are many options for the definition of the formula, allowing for example, easy implementation of forward or backward step-wise multiple regression. There is documentation on the web, but I have also included a nice summary called *formulaNotation.pdf* in the Resources folder.

```
RegModel.1 <- lm(disability~age+bmi, data=bP)
RegModel.1
```

The function `lm()` simply returns an `lm` object. To see more detail of the results of the computation you can use the `summary()` function (which knows how to deal with an argument which is an `lm` object).

```
summary(RegModel.1, warning = warn)
```

We'll see in the next section that the function `stats::anova()` also knows how to deal with an argument which is an `lm` object.

```
LinearModel.2 <- lm(disability ~ age + bmi + eduS + physical + residence +
  wealthQ + comorb + country, data = bP)
summary(LinearModel.2)
```

13.8.1 Linear Model Options

.... and here's one with a subset expression.

```
LinearModel.3 <- lm(disability ~ age + bmi + eduS + physical + residence +
  wealthQ + comorb + country, data=bP, subset = sex == "Male")
summary(LinearModel.3)
```

13.9 Generalized Linear Models

The `glm()` function is a very powerful function with many useful arguments. If GLM is your analysis, you will need to have a good look at the documentation. `?glm`. I note that the first example on the help page is from the classic text book ([9]) by my former colleague, Annette Dobson. (There is R code available for all the examples in the latest (4th) edition of that book.)

```
GLM.6 <- glm(disability ~ age + backPain30 + sex, family = gaussian(identity),
   data = bP)
summary(GLM.6)
```

...and logistic regression:

The GLM gives us the logistic regression option. This is expressed as the binomial distribution with a log odds link.

```
GLM.8 <- glm(backPain30 ~ age + bmi + sex + arthritis + wealthQ,
  family = binomial(logit), data = bP)
summary(GLM.8)
```

Try plotting the `glm` object.

In the next lecture we'll take a look at some diagnostics of potential problems in regression.

Chapter 14

Tutorial 7 - Introduction to regression

Brian Williams <bjw649@gmail.com>

14.1 Preliminaries

Go to the Code folder and open the courseData.R file. Find "Tutorial7".

Set the working directory to that of the source file. Last chance to add to your R Markdown file. Submission tomorrow!

14.2 Exercise

In your data folder you will find a file called nhgh.rda. This is a saved file comprising a data frame of diabetes patient data from the NHANES data set. Load the data set and try regressing bmi against waist circumference. Run the standard diagnostics, by using `plot()` on the `lm` object. Summarise the regression.

Create a scatter plot with a line of best fit.

You might like to explore some other relationships in this data set. Have a look at the examples of multiple regression in the Help page for `lm`.

```
load("../Data/nhgh.rda")
```

Chapter 15

Lecture 8 - Anova and Regression diagnostics

Brian Williams <bjw649@gmail.com>

15.1 Preliminaries

Install the `car` package. This is a very useful package written by John Fox. The name is from the title of his book "An R companion to applied regression".

Load bP if you've not done so already, and go to the courseCode folder and open the Lecture8.R file. Save as - to the myRcode folder. Set the working directory to that of the source file.

```
# Initial read of back pain data set
load("../Data/Backpain.Rdata")
```

@

15.2 ANOVA

We'll continue with some of the models we set up in the last Lecture. Let's begin by setting them up again.

```
RegModel.1 <- lm(disability~age+bmi, data=bP)
RegModel.1
```

```
LinearModel.3 <- lm(disability ~ age + bmi + eduS + physical + residence +
  wealthQ + comorb + country, data=bP, subset = sex == "Male")
summary(LinearModel.3)
```

ANOVA analysis is applied directly to `lm()` objects. Have a look at the structure of these objects. They contain *lots* of information about your model which can be used for analysing and presenting the outcomes of your regression in many ways.

```
str(LinearModel.3)
```

Here's the one-way ANOVA as presented by the core function `stats::anova`.

```
anova(RegModel.1)
```

And here's John Fox's version, `Anova()`, from the `car` package.

```
library(car)
Anova(LinearModel.3, type="II")
```

Have a look at the documentation of the `car` version by typing `help(Anova)` or `?Anova` in the Console pane.

15.2.1 Compare two models

If we have two `lm()` objects, we can use the base R function `anova()`, from the `stats` package. It's different to `Anova()`. Check it out: `?anova`.

Firstly, we run a version of LinearModel.3 without the country variable.

```
LinearModel.5 <- lm(disability ~ age + bmi + eduS + physical + residence +
  wealthQ + comorb, data = bP, subset  = sex == "Male")
summary(LinearModel.5)
```

Now do the comparison:

```
anova(LinearModel.3, LinearModel.5)
```

15.3 Model diagnostics: checking assumptions

15.3.1 Subset the data

Just to keep things a little more convenient for the diagnostic analysis, we'll take a country sub-set. In the previous lecture, we sub-setted the China data - so we'll simply usethe same code here:

```
China <- filter(bP,  country == "China")%>%
         select( -country)%>%
         drop_na()
China
```

Now fit a model to the China data.

```
LinearModel.1 <- lm(  bmi  ~ age + sex + residence + wealthQ + wHR,
                    data = China)
summary(LinearModel.1)
```

The function `lm()` also has provision for sub-setting a dataframe in its arguments.

15.3.2 Add the observation statistics

Its easier to do the diagnostic checking if you add the observation statistics to the data frame.

```
library(car)
fitsLM1 <- fitted(LinearModel.1)
residLM1  <- residuals(LinearModel.1)
df <- data.frame(fitsLM1,residLM1)
China<- bind_cols(China, df)
 head(as.data.frame(China))
```

The fitted values and the residuals are added to the China data frame using the `dplyr` function `bind_cols()`.

15.3.3 Normal probability plots

We'd like to know that our residuals are normally distributed, because otherwise we can't be sure of the quality of our fitting process (which may in a non-normal case weight outliers too heavily), but perhaps more importantly many common tests of regression results *assume* normality. So if the residuals are not normal, then you cannot reliably perform many of the basic hypothesis tests.

```
 p <- China %>% ggplot(aes(sample=residLM1))
p <- p + stat_qq() + stat_qq_line()
p
```

Ouch! Does not appear to be normally distributed!

15.3.4 Scatter plot of fitted bmi vs observed bmi

If we do a scatter plot of fitted bmi vs observations, we'll see clearly how the distortion in the qqplot has come about.

```
ggplot(China) + geom_point(aes(x = bmi, y = bmi), colour = "red") + geom_point(aes(x = bmi, y = fitsLM1)
```

The red dots are the observed bmi plotted against itself. The green dots are the fitted bmi against the observations.

15.3.5 Scatter plot of fitted values and residuals

```
p <- ggplot(China, aes(x=fitsLM1, y = residLM1)) + geom_point(alpha = 0.1, colour = "brown")
p
```

It looks like the variance might be increasing (heteroscedastic).
John Fox has spent some time preparing a scatterplot wth rolling sd.

```
library(car)
China %>% scatterplot(residLM1~fitsLM1, reg.line = lm, smooth = TRUE, spread =TRUE,
            id.method = 'mahal, id.n = 2', span = 0.5, data = .)
```

The function `scatterplot` is also from the package `car`. Its description from the documentation page (`?scatterplot`) is:

> "Makes enhanced scatter plots, with box plots in the margins, a non-parametric regression smooth, smoothed conditional spread, outlier identification, and a regression line; sp is an abbreviation for scatter plot."

This plot is particularly useful for checking heteroscedasticity . We can see that there are problems with this one!

15.3.6 Histogram of residuals

```
p <- ggplot(China, aes(x=residLM1)) +geom_histogram(binwidth = 0.5, colour = "brown", fill = "beige" )
p
```

15.3.7 Variance-inflation factors

The `car` function `vif` computes variance inflation functions for linear models and GLM's.

```
vif(LinearModel.1)
```

```
car::qqPlot(LinearModel.1, simulate=TRUE, id.method="y", id.n=2)
```

15.3.8 base::plot of an `lm` object

Finally, we can simply plot an `lm` object and get a sequence of standard diagnostic plots, similar to those we have just seen. The last of them plots the Cook's distance and leverage, giving indications of the observations which have most affected the regression - and perhaps should be considered as outliers?

```
plot(LinearModel.1)
```

Chapter 16

Tutorial 8 - Regression 2

Brian Williams <bjw649@gmail.com>

At the beginning of the session

Go to the Code folder and open the courseCode.R file. Find "Tutorial8".
Set the working directory to that of the source file.

```
load("../Data/BackPain.csv")
```

16.1 Assignment due for submission today.

File stucture testing.

Chapter 17

Lecture 9- Mapping in ggplot2

Brian Williams <bjw649@gmail.com>

17.1 Summary

This session continues with data manipulation, introducing the wide and long form and mapping in the ggplot2 package.

17.2 Logistics

Go to the Code folder and open the courseCode.R file. Find "Lecture9". Set the working directory to that of the source file.

Before the lab

I'd like you all to download shape files and .Rdata files for a country in which you have an interest. Find the site `gadm.org`. On the home page you will see a prominent line:

> You can download the data by country or for the whole whole world.

In this line 'country' is a link. Click on it. Choose your country and download, firstly, a *shape file*. .The shape file will come as a zipped file, incorporating the ISO3 country name of the country you chose. Secondly, (going back to the previous page), download a .Rdata (selected from the drop-down menu 'File format'). Place both files in your Code folder. Extract all from the zip shape file - it should create a folder in the Code folder with the same name as your zip, but without the '.zip' extension.

You will need a number of specialised mapping packages. Some of these contain large datasets. It will help if you install any of these which are not already installed before coming to Tutorial 8.

1. maptools
2. rgdal
3. rgeos
4. maps
5. sp
6. spam
7. fields

At the beginning of the session

Go to the Code folder and open the R_Course.R file. Find Lecture9". **Set the working directory to that of the source file**.

17.3 Mapping

The main theme in this lecture will be using ggplot to prepare maps incorporating a number of different representations of data. There are a number of useful sources for help on mapping including the CRAN Task View: Analysis of Spatial Data.The recent paper by Kahle and Wickham (2013)[17] is also a neat introduction.

One of the significant issues in using maps is that country borders change and many of the maps available on the web are not up-to-date. If borders change in countries which are of particular interest with regard to your data, care must be taken to find map data with accurate border information. There can be political problems if you try to publish articles with maps which do not have current borders.

The CRAN Task View: Analysis of Spatial Data is the best overview of what's available in R for mapping purposes. There are lot's of packages doing things you may never have heard of if you are not a professional map-maker! There are many who say that R is poised to overtake the traditional GIS software for mapping analysis - others suggest that the focus of traditional GIS is mapping capabilities, while R's strength is statistical analysis of spatial data - with some mapping added! This said, there are a number of less flexible but very straightforward mapping systems that you may like to experiment with before diving into the murky waters of `ggmap` and all the others! Have a look for example, at the package `rworldmap`'s vignette and you will see that there are a number of reasonably straightforward approaches to getting a number of interesting maps on global scale. The package does provide quite recent useful maps of country boundaries. Unfortunately, though, (at the time of writing) this package does not allow you to map within country data.

Most significant mapping datasets these days use a storage format called a 'shapefile', which is in fact a folder containing a collection of files which store various mapping information and optionally other data related to the locations being mapped. For our purposes there are two ways of reading shapefiles, one using the package `rgdal` and the other using the package `maptools`. Both will be demonstrated in this lab. The former (`rgdal`) deals effectively with coordinate systems, which are important in maps drawn for world scales, while `maptools` is very adept at dealing with maps at country scales. If you would like to read a comparson of these two approaches, see here and to read more about the shapefiles themselves, see here.

Perhaps the most authoritative reference on mapping in R is the book by Roger Bivand et al. A recent introductory book, by Brunsdon and Chen, focusses mostly at country level with US examples and looks probably more readable for a beginner than Bivand's book. Its Amazon page is here. Dorman's "Learning R for geospatial analysis", is a nice introductory level to the basics of geospatial analysis and how to do it in R. (I've only browsed Bivand's first edition and Dorman's book and I haven't seen Brunsdon's book.)

17.3.1 Country scale maps from GADM

At world scale, care must be taken about the location of country boundaries since many of the map data freely available on the web are not current - especially in war zones! Some governments will be especially sensitive and if you intend to publish maps, you need to be careful about the current accuracy of your maps.

You also need to be aware of the need to have an appropriate coordinate projection.

You will have noticed that when you unzipped your download of the shape file from GADM, it was in fact a number of files. Keep them all in the same directory, because a shape file is not a single file - all are necessary for the shape file to work properly!

The GADM files are of administrative regions at various scales, coarsest being 0. Almost all countries have at least level 2 and many have level 3.

Winston Chang's book has a little bit of information on using maps in ggplot. Here's his code to access a shapefile using the maptools package. (Remember that this package does not automatically sort out an appropriate projection, though at country scale its default is probably OK.)

I've got the shapefile folder for Australia in the Data folder. Notice that a shapefile is actually a folder with a number of files, including one with extension .shp, which is the layer argument in the function `readOGR()`. If you've put a shapefile folder from a country of your own choice in the Data folder, edit the code to agree with the three letter ISO Code name for your chosen country in the argument of `readOGR()` and then run the block of code below.

```
library(ggplot2)
library(rgdal)
library(sp)
SF <- readOGR(dsn = "../Data/gadm36_SWE_shp/gadm36_SWE_2.shp", layer ="gadm36_SWE_2")  #1
#SF <- readRDS("../Data/gadm36_SWE_2_sp.rds")                    # 1.
class(SF)                                                       # 1.
head(SF@data)                                                   # 2.
#plot(SF)                                                        # 3.
cMap2 <- fortify(SF)                                            # 4.
head(cMap2)                                                     # 5.
myMap <- ggplot(cMap2, aes(x=long, y= lat, group = group))+ geom_path()
#myMap <- myMap + coord_equal()  +ylim(-45,-8) +xlim(110,155)
myMap <- myMap + coord_map("mercator")
#myMap = myMap + coord_map("ortho", orientation = c(-34, 151, 0))
myMap
```

Notes

1. The function `readOGR` (from the `rgdal` package) reads a shape file and converts it to a `SpatialPolygonsDataFrame` object. This is not a 'simple' data frame but contains many layers of data in 'slots'. It is a class of object that is fundamental to mapping in R - but is too complex for us to deal with here. If you enter `str(SF)` you will get an overwhelming screen full of information!

2. Here you see some of the upper layer items stored in the `SpatialPolygonsDataFrame` object.

3. Yes - `base::plot` knows how to deal with a `SpatialPolygonsDataFrame` object. You won't have a lot of flexibility with this approach, though.

4. The `fortify` function is from ggplot2. Its purpose is 'to convert a generic R object into a data frame useful for plotting'. The official documentation for `fortify()` is very terse. You will need to do a web search if you have trouble with the function calls.

5. Having 'fortified' the `SpatialPolygonsDataFrame` object, we now have a standard data frame that we can access with the usual functions in `ggplot2`. `head` shows us that we've not converted a lot of data in this process!

.

17.3.2 Plot of Burden of Health (back pain disability) using `rworldmap`

In the next code chunk, we'll make use of the data in the package `rworldmap` to produce a world map plotted with Burden of Health measured as years lived with disability per 100000 (YLD) on a coloured scale for each country. In addition, we highlight the Sage countries by giving them a magenta coloured border and annotating them with their names.

```
# This is a simple plot of the Burden of Health resulting from Back Pain
# The units are YLD - years lived with disability per 100,000.
# The borders of the Sage countries have been included in RED
# and their names annotated.
# The BofH plot covers most countries. Greenland has been set equal to Iceland
# South Sudan has been set equal to Sudan
# Note that merging has been done on country names so some smaller countries with
# Inconsistent names in the two dataframes may not have had their BofH included
# (Better to use ISO-3 names for uniqueness and consistency)

library(rworldmap)
library(ggplot2)
library(scales)
library(dplyr)

data(countriesLow)                                                    #1.
plot(countriesLow)
class(countriesLow)                                                   #1.
head(countriesLow@data)
wmap_df <- fortify(countriesLow)  # Converts  to a data.frame
head(wmap_df)
names(wmap_df)[6] <- "NAME"
#-----------------------------------------------------------------------------
bp50_69 <- read.csv("../Data/YLD_50-69.csv",header=TRUE,stringsAsFactors=FALSE)   #2
head(bp50_69)    # Just country name and YLD

# merge the dataframes
map.df <- left_join(wmap_df, bp50_69, by = "NAME")              #3.
map.df <- left_join(wmap_df, bp50_69)              #3.
map.df
map2 <- arrange(map.df,order) # sort to original polygon order

breaks <- c(1500, 2000, 2500, 3000, 3500, 4000, 4500)  # For legend

theme_opts <- list(theme(panel.grid.minor = element_blank(),
                         panel.background = element_blank(),
                         plot.background = element_rect(fill="lightblue"),
                         panel.border = element_blank(),
                         axis.line = element_blank(),
                         axis.title.x = element_blank(),
                         axis.title.y = element_blank(),
                         plot.title = element_text(size=22)))            #4.

pp <- ggplot(data = map2, aes(x=long, y=lat,group=group, fill=YLDs)) +
#pp <- ggplot(data = map2, aes(x=long, y=lat, fill=YLDs)) +
    scale_fill_gradientn(colours = rainbow(7),na.value="white",
                         breaks = breaks, labels = format(breaks))+
    coord_fixed() +
    geom_polygon(colour="white")
pp                                                                    #5.
#-----------------------------------------------------------------------------
# Now we add the SAGE country outlines

SageCountry <- c("China", "Ghana","India", "Mexico", "Russia", "South Africa")
```

```
# strings for names on map
labelLong <-   c(   100.,   -10.,    66.,     -120.,     90.,         20.)
# location for country name on map
labelLat  <-   c(   37.,     0.,     18.,      20.,      60.,        -40.)
# location for country name on map

# Now construct a data.frame for ggplot
cL  <- data.frame( long = labelLong, lat= labelLat,                      #6.
                   txt = SageCountry,stringsAsFactors=FALSE )

# First we plot the SAGE country borders
for ( i in 1:length(SageCountry)){                                       #7.
  bb <-  map2[map2$NAME== SageCountry[i],]  # grab the Sage country borders
  pp <- pp + geom_path(data = bb,aes(x=long,y=lat), colour = "magenta",size=1)
 rm(bb)  #
}

# Now do the annotation of country names
  pp <- pp + annotate("text", x = cL$long, y = cL$lat,                   #8.
                      colour= "black",
                      size  = 4,
                      label = cL$txt)

# Include a title for the plot
tt1 <- "Years lived with disability (YLDs) per 100,000 - Group age 50-69 "
 pp <- pp +
  ggtitle(tt1)+
  theme(plot.title = element_text(size = rel(2), colour = "blue"))
pp                                                                       #9.
```

Notes

1. `countriesLow` is a low resolution shape file of the world's country boundaries.
2. Data were sourced from IHME in Seattle, USA.
3. Merge the two data frames using the country names as the key for merging.
4. Sets up an essentially blank panel for the plot.
5. Initial plot.
6. A separate data frame is constructed for the SAGE country name annotation.
7. We add one layer at a time in a loop for the country borders.
8. Add the SAGE country name annotations.
9. Add title and print to screen.

17.3.3 Clean background - Chang again

Winston Chang's code for a clean background is given in the next chunk. If you run it and then add it to the pp ggplot object, you'll get your map without the axes and with the title back to its default font.

```
theme_clean <- function(base_size = 12) {
  require(grid) # Needed for unit() function
  theme_grey(base_size) %+replace%
    theme(
      axis.title        = element_blank(),
      axis.text         = element_blank(),
      panel.background  = element_blank(),
      panel.grid        = element_blank(),
      axis.ticks.length = unit(0, "cm"),
      panel.spacing     = unit(0, "lines"),
      plot.margin       = unit(c(0, 0, 0, 0), "lines"),
#      plot.background   = element_rect(colour = "black"),  # leave out for no box
      complete = TRUE
    )
}
# Attach it t the pp object:
pp + theme_clean()
```

17.3.4 Country plot of world, using Natural Earth data

Another source of country data at a number of scales is the website Natural Earth. Here, 110m scale data for country mapping can be obtained in shape file format. I've provided the file for you in the Cambro download and you should have placed it in your R_course/Data folder.

This time we'll not use maptools and `readShapePoly`. Instead we'll use the package rgdal and `readOGR`. This appears to be the preferred package by experienced users on the web and is R's interface to the "Geospatial Abstraction Library (GDAL). For an overview of the spatial packages available in R, have a look at this webpage.

```
Nlibrary(maps)
library(ggplot2)
library(rgdal)

# read shapefile
wmap <- readOGR(dsn = "../Data/ne_110m_admin_0_countries",
                layer="ne_110m_admin_0_countries")
class(wmap)    # SpatialPolygonsDataFrame
Amap <- wmap[wmap$name == "Australia",]       # Subsetting to grab Australian data
plot(Amap)
wmap_df <- fortify(wmap, regions = ISO_A2)  # convert SpatialPolygonsDataFrame to a dataframe
head(wmap_df)

base_world <- ggplot(data = wmap_df, aes(x = long, y = lat)) + coord_fixed()+

  geom_polygon(aes(group = group), fill = "wheat") +
  geom_path(aes(group = group), colour = "grey40") +
  theme(panel.background  = element_rect(fill = "lightsteelblue2", colour = "grey"),
        panel.grid.major  = element_line(colour = "grey90"),
        axis.ticks        = element_blank(),
        axis.ticks.length = unit(0, "cm"),
        axis.text.x       = element_text(size = 14, vjust = 0),
        axis.text.y       = element_text(size = 14, hjust = 0.3)) +
  labs(y="",x="")
base_world
```

17.3.5 Other learning resources

There is an example of plotting at regional scale on Hadley Wickham's github ggplot repo.

Chapter 18

Tutorial 9 - Practice mapping with ggplot2

Brian Williams <bjw649@gmail.com>

18.1 Preliminaries

Go to the Code folder and open the courseCode.R file. Find "Tutorial9".

Set the working directory to that of the source file.

18.1.1 Washington State water catchments example

The next example is part of a post by Bethany Yollin on this page, which neatly describes the use of mapping for one of my interests, which is hydrology. Have a look at the whole post, which shows how to subset the data and perform some simple calculations. A pdf of the post is also stored in the Lab 5 folder.

Public health people, too, must of course, be interested in water sources because of the attendant potential for disease transmission or habitats for vectors such as mosquitoes. Yollin has used the standard global approach using the `rgdal` package, though the `maptools` package should work equally well.

```
# load libraries
library(ggplot2)
library(sp)
library(rgdal)
library(rgeos)
library(dplyr)

# Create a string containing the location of the shapefile
localDir <- "../Data/wria"
# create a layer name for the shapefiles (text before file extension)
layerName <- "WRIA_poly"
# read data into a SpatialPolygonsDataFrame object
dataProjected <- readOGR(dsn=localDir, layer=layerName)
class(dataProjected)
```

The output shows the conversion from an ESRI shapefile to an R `SpatialPolygonsDataFrame` (from the `sp` package), which is the de facto GIS standard data object in R.

In the process of creating data frames and joining external data, there is potential for changed order of rows, so in order to maintain consistency between the non-map data and the map-data in the SpatialPolygonsDataFrame, we add a row ID to its data 'slot' (Slots are part of S4 class data objects in R which are accessed using the '@' symbol.)

Next, we use the ggplot function `fortify()` to convert the `SpatialPolygonsDataFrame` to a standard R dataframe, which is what ggplot usually plots from.

Having created a data frame containing we can then use a `left_join` function to add to it, the watershed data which is stored in the data slot of the SpatialPolygonsDataFrame.

```
# add to data a new column termed "id" composed of the rownames of data
dataProjected@data$id <- rownames(dataProjected@data)
# create a data.frame from our spatial object
watershedPoints <- fortify(dataProjected, region = "id")
head(watershedPoints)
# merge the "fortified" data with the data from our spatial object
watershedDF <- left_join(watershedPoints, dataProjected@data, by = "id")
head(watershedDF)
#Alternative: watershedDF <- merge(watershedPoints, dataProjected@data, by = "id")
# NOTE :
# : An equivalent SQL statement might look something like this:
# : SELECT *
# : FROM dataProjected@data
# : INNER JOIN watershedPoints
# : ON dataProjected@data$id = watershedPoints$id
#OR using the plyr package
# library(plyr)
# watershedDF <- join(watershedPoints, dataProjected@data, by = "id")
```

The `head()` statement has shown us that the watershedPoints dataframe contains latitude and longitude pairs in sequence which are associated with various points on boundaries on the map.

These can be readily joined with straight lines to produce the map's boundaries. The new dataframe `watershedDF` contains additional information about the watersheds contained by this boundary information.

```
N# Plotting with ggplot2
ggWatershed <- ggplot(data = watershedDF, aes(x=long, y=lat, group = group,
fill = WRIA_NM)) +
geom_polygon() +
geom_path(color = "white") +
scale_fill_hue(l = 40) +
coord_equal() +
theme(legend.position = "none", title = element_blank(),
axis.text = element_blank())
print(ggWatershed)
```

Themes are a way of changing or setting backgrounds, axes, titles etc in a standardized way. IN this case the legend has been removed, the axis text has been removed and the title has been removed.

18.2 Fast maps using `coord_quickmap` and `map_data`

`ggplot` offers a quick approximate coordinate transformation called `coord_quickmap` which is easy to use for 'small' countries. It ensures that the aspect ratio in the central 1m by 1m square of the plot is accurate. Here is an example:

```
worldMap <- ggplot(map_data("world2"), aes(long, lat, group = group)) +
  geom_polygon(fill = "white", colour = "dark green") +
  xlab(NULL) + ylab(NULL)
```

```
# worldMap + coord_quickmap()
 worldMap + coord_map("mercator")
```

Chapter 19

Lecture 10 - Intro to apply and other functions, distributions...

Brian Williams <bjw649@gmail.com>

19.1 Preliminaries

Go to the Code folder and open the courseCode folder. Find and open "Lecture10.R". Save it in the myRcode folder. Set the working directory to that of the source file. We'll use the back pain data set later in this lecture.

```
library(tidyverse)
load("../Data/BackPain.Rdata")
bP <- drop_na(bP)
```

19.2 Lists

Matrices and vectors must have all elements of the same type. A list in R, is simply a collection of data objects (some of which might themselves be lists). It might comprise for example a 4 by 4 matrix, a 20 element numeric vector and a character vector containing two elements. Lists are the most general form of vector in R. (We sometimes refer to the vectors we saw in Section 3.3 as *atomic vectors*, because they only contain atomic elements, i.e. the basic types, numeric, character etc.)

We will see that lists have a very important role for containing a collection of arbitrary pieces of data that is not in tabular form. It is easy to add extra data items to them, too. They are a widely used data object, especially within functions.

Let's construct one:

```
##
#-------------------------------------------------------------------------------
a  <- matrix(sample(1:20, 16, replace=TRUE), ncol=4)      #1. see note on sample()
b  <- rep(c(3, 17, 4, 6, 11), 4)
c1 <- c('Lena', 'Nice day!')
l1 <- list(M = a, b = b, c2 = c1)                         #2. creating the list
l1
```

1. Reminder: `sample(x,n)` returns `n` samples from a vector `x`. The parameter `replace = TRUE` maintains all elements of `x` for all sample selections. The `matrix()` function constructs a matrix object, in this case with four columns

2. The list has been created with names for each list item. The first list item is the matrix `a` and has the name `M`. Don't forget when constructing small examples like this, that it is very dangerous to use 'c' as a variable name because it becomes confused with the base function `c()`.

19.2.1 Accessing members of lists

Many (S3) data objects carry internal elements which can be accessed using a '$' symbol, as we saw for data frames in Tutorial 1, Section **??**. A data frame is in fact a special kind of list (next section). The '$' notation is a useful and informative approach because it allows the internal element to be named. Where numbers are not required for programming purposes, using names is much less prone to errors.

Lists can also access elements using the square bracket notation that we have already seen for atomic vectors.

```
class(l1$c2)                    #1. Dollar notation returns original object class
l1$c2[2]                        #2.  accessing an element
                                #    of a list member
```

1. List items are accessed by referring to them using the list name followed by a '$' followed by the list item's name. This access method returns the original object (class).

2. To refer to an element of a list item, use the [] index system we have already seen.

```
l1[2]                           #1.  Single bracket
class(l1[2])                    #2.  Single bracket returns list!!
l1[2:3]                         #3.  Single bracket can access multiple items !!
class(l1[2:3])
```

Notes

1. List items themselves can be accessed by using the [] index system we have already seen.

2. -But note that this method returns a *list*!

3. Because a list is returned, mutiple members can be indexed with a suitable index expression. Very useful for subsetting a list.

Sometimes, however, we don't want a list returned, we'd like the list item with its original type. We saw that happen with the '$' notation. This can also be done with a double square bracket notation, which can use names (in quotes) or numeric indexes:

```
class(l1[['c2']])
l1[['c2']][1]                               #1.  double bracket with name

class(l1[[1]])
l1[[1]][4,1]                                #2.  and with index
```

Note that in these two cases, subsetting with the double square brackets has resulted in extraction of a character vector and a matrix.

Adding and removing members of lists

Adding and removing list members is very easy. Once a list has been created, it is simply a matter of naming a new member. It will be added to the end of the list.

To remove a list member, it is set to 'NULL'. Once done, the indices of following members are all shifted up one.

As you might by now expect, you can use the `names()` function to either 'get' or 'set' names of list members.

```
d <- c(64, 69)
l1$d <- d                                    #1.  Add a new member
names(l1)[4] <- 'newName.d'                  #2.  Give it a name
l1
# Remove the item, using its name
l1$newName.d <- NULL
l1
```

Notes

1. Adding a list item is straightforward. Lists can also be combined using `c()`.
2. Deleting a list item simply requires assigning it a NULL value.

19.3 The apply family of functions

The `apply` family is a very important group of functions in R. They are very powerful and 'underneath' they are coded in the programming language C, making them very efficient. Where they can be used instead of loop structures (see Section 21.4), they will generally compute much faster.

19.3.1 `apply()`

The function `apply()` returns a vector or array or list of values obtained by applying a function to the row or columns of a matrix, so `apply` requires that all the variables be of the same type. In general dataframes are of mixed type. Let's subset our back pain data for that purpose. We'll look at just the numeric variables. The arguments of `apply()` are as follows:

1. Argument(1) is the name of the matrix,
2. Argument(2) = 1 indicates that the function is to be applied to the rows and argument(2) = 2 indicates that the function is to be applied to the columns.
3. Argument(3) named FUN, is the name of the function which is to be applied. User functions can be included with argument 'x' of the user function corresponding to the rows or columns specified for application.
4. Additional arguments follow the function argument in `apply`'s argument list. Preferably, they should be named arguments in the call to `apply()`.

Here we demonstrate base R's `subset()` function. (We have seen that you can do it more tidily for all the numerics using `select_if` from `tidyverse`).

```
bPNum <- subset(bP,select= c(age,bmi,waistc,disability, height)) #select most of the
                                                                 # numeric variables
pbw <-  function(x, power)     #defining our own function!!
{
```

```
x^power
}

head(bPNum)
crbP <- apply(bPNum, 2, FUN = pbw, power = 1/3)
head(crbP)
apply(bPNum, 2, range, na.rm = T)
```

We will have a much closer look at defining our own functions in Section 19.4

19.3.2 `lapply()` and `sapply()`

The functions `lapply()` and `sapply()` are more useful in that they apply a specified function to each element of lists (data frames are a sub-class of lists in which each column is a member of the list.) The `lapply()` function returns a list, while `sapply()` tries to simplify the result with regard to type.

Firstly, let's have a look at a simple useful application of `sapply`. Here's how to select all the numeric variables in a data frame. (We can do this using `tidyverse`, too!)

```
nums <- sapply(bP, is.numeric)    # -> a vector of T/F for the columns (variables)
bPnum <- bP[ , nums]
names(bPnum)
# and here are the factors
facs <- sapply(bP,is.factor)
bPfac <- bP[ , facs]
names(bPfac)
```

Here's an example which provides bmi means, listed country by country.

```
bP <- as.data.frame(bP)
countryList <- list(bmiChina = bP[bP$country == "China","bmi"],
                    bmiIndia = bP[bP$country == "India","bmi"],
                    bmiGhana = bP[bP$country == "Ghana","bmi"])
str(countryList)
lapply(countryList,mean, na.rm=T)
bP <- as_tibble(bP)
```

Note the requirement to set the tibble to a dataframe in this example! And don't forget to convert it back or you could get into trouble later!

Keep in mind that for data frames, we can do this using the dplyr package and that will generally be our preference. The function `lapply()` is more general in that it applies to lists and when we are trying to access data which is not in the form of a data frame, lists will often be the natural form of the data.

Here is the dplyr equivalent:

```
library(dplyr)
bmisum <- group_by(bP, country)
bmiM <- summarise(bmisum, bmiMean = mean(bmi, na.rm=TRUE), bmiSD = sd(bmi))
bmiM
```

19.3.3 `tapply()`

The function `tapply(vector, factor, function)` creates tables of some function (third argument) of variables in a vector (first argument) 'conditioned' on a factor variable (second argument).

```
waistc   <- tapply(bP$waistc,bP$country,mean, na.rm = TRUE)
waistcSD <- tapply(bP$waistc,bP$country,sd,  na.rm = TRUE)
height   <- tapply(bP$height,bP$country,mean,  na.rm = TRUE)
heightSD <- tapply(bP$height,bP$country,sd,  na.rm = TRUE)
bmi      <- tapply(bP$bmi,bP$country,mean,  na.rm = TRUE)
bmiSD    <- tapply(bP$bmi,bP$country,sd,  na.rm = TRUE)
cbind(waistc, waistcSD, height, heightSD, bmi, bmiSD)  # make them into a table
```

19.4 R functions - writing your own

All programming languages provide some kind of 'function' to enable the modularization of complex tasks. R is unusual in that functions are objects. To define a function, we assign an **argument** list and a **body** to a name for the function object:

```
f3 <- function(x = 0, y = 0){
  x*3*y
  }
x <- 7
y <- 2
f3(x,y)
```

Here we have created a function object called f3, which has two arguments, x and y. (We will discuss the '= 0' below). The body of this function, consisting of R expressions, is enclosed in curly brackets, although in this case, with a single expression, the curly braces could have been omitted.

So, functions have a defined task accomplished using three components:

- an input list of **arguments**, called the 'formals',
- a 'body' containing R expressions and
- an 'environment'.

All functions also have a defined return value (output).

The components of a function can be accessed using the functions `formals()`,`body()` and `environment()`.

```
f3
ff3 <- formals(f3)
ff3
str(ff3)
af3 <- args(f3)
af3
str(af3)
body(f3)
environment(f3)
```

The 'input' argument list is what makes functions different from scripts. Functions are a means to control computations so that inside the function, *the only data objects which can be assigned new values* are objects created within the function and then returned from it. This provides the tight control which is necessary in complex software. **Copies** of the values of the variables in the calling argument list are passed from the calling environment to the function. Even if the argument is redefined within the function (by removing NA's for example), it is **local to the function** and the new value is not returned to the calling environment. It **is** possible, by using the special **super assignment operator**, «-, or the `assign()` function to modify external variables ('globals'). The values of globals and variables in the calling environment

can be accessed from within a function, but can only be changed using «-, or `assign()`. In general, using global assignment or accessing globals from within functions is dangerous and should not be done.

The body of a function is a sequence of R commands. The body begins with an opening curly bracket and ends with a closing curly bracket. (Sometimes very short functions comprising a single command are created - these don't require curly brackets). Here's an example:

```
singleLine <- function(x)  mean(c(x+6, x^2))
singleLine(6)
```

The output of a function, in the absence of any specified return statement, is the result of the last computation. Only a single object may be returned, but it may be a list which has been constructed in the function to include a number of objects wanted outside the function. The output is 'returned' to the calling environment, i.e. it becomes accessible in the part of your program where the function was called. (To complicate matters, this could be within another function!)

This control of your data through input and output objects means that your function cannot inadvertently modify any other data objects and provides good data security. And it requires you to plan and define more carefully how you will make changes to your data.

R is a little unusual, in that its functions are themselves objects. This means that in defining a function, the user assigns it to an object. Furthermore, as we have seen with other objects, to print it, simply type in its name. Functions, being objects, conveniently act in the same way.

Let's begin with a simple example, which compares two vectors. In the calls to the function, the two arguments, a and b are given the values a1 and b1, by the assumption (from the function definition) that the first argument corresponds to a, the second, b. The following three functions produce the same output.

```
# Simple functions showing alternative output mechanisms
 gg1 <- function(a, b){
   (a == b)                                    # Last evaluation is returned
 }

 gg2 <- function(a, b){
   return(a == b)                             # Explicit statement of return
 }
 gg3 <- function(a, b){
   d <- (a == b)
   return(d)                                 #  Another way
 }

a1 <- c(1, 1, 2)
b1 <- c(1, 1, 1)

gg1(a1, b1)
gg2(a1, b1)
gg3(a1, b1)
```

Notice how the functions are listed in the Environment pane after they have been 'run'. 'Running' the definition of a function (we say that we 'source' the function) produces no output, but we must do so before the function can be called.

In order to make the use of arguments more flexible, default values can be assigned to arguments so that it is not necessary to specify a value for them when the function is used ('called'). If an argument is not specified in the function calling statement, it assumes the default value.

Argument names can also be used to more clearly specify them, but as we saw above, it is not necessary to refer to the argument by name when specifying a value, provided that the order of the arguments is strictly adhered to. If calling statements use arguments in sequences which are different from the function

definition, then argument names *must* be used for all arguments after (and including) the first out-of-sequence argument.

R also offers a '...' argument, which is for 'extra' arguments. It can be used for combining a list of unknown number of objects (as in the `list()` function, for example), or as in the case of plot(x, y, ...), it allows additional named parameters to be passed to `par()`. We will see below, how you can make use of this construct in your own functions.

Everything in R is an object and every object knows how to print itself to your screen when you type its name, so you can get some basic information about many functions by simply typing in their name - usually help is a better way!

Let's continue with an example to demonstrate how default arguments work.

```
# Use of default arguments in a function, showing alternative modes of calling
gg4 <- function(a = c(1, 1, 1), b = c(1, 1, 1), f = c(1, 1, 1)){
  # All arguments have a default value
   d <- ( a == b & a == f)
   return(d)
 }
a1 <- c(1, 1, 2)
b1 <- c(1, 1, 2)
f1 <- c(1, 1, 2)
gg4(a = a1, b1 ,f1)                                    #1.
gg4(a1, b1)                                            #2.
gg4()                                                  #3.
gg4(f = f1)                                            #4.
gg4(a1, f = f1)                                        #5.
```

Notes

1. a is named, all arguments are specified.
2. The f argument has been omitted, so takes the default value.
3. All arguments use the defaults.
4. a, b use defaults.
5. Because b has been omitted, it takes the default value and the name of f must be specified (otherwise it would be confused with b.)

Let's have a look at the '...' argument. Most people call this the 'dots' argument - technically it's an 'ellipsis'.

If you check the documentation of the function `lapply` (which we looked at in Lecture 5, Chapter **??**) you will see that its arguments are `lapply(X, FUN, ...)`. The function function `lapply` applies some function (in the function definition it is given the 'dummy' name FUN) to some object 'X'. When you call function `lapply`, you specify your own function, for example `mean`. Now in using `mean`, it is possible that you would like to specify some additional parameters - for example `na.rm = TRUE`. This is where the dots are useful. If you call `lapply` with any arguments other than the object, X and a function name, the arguments are assumed to be associated with your function. It is strongly advisable that you name or 'tag' any arguments added to the call by way of the dots. It is a good idea, in general, to name arguments for clarity of coding. In other functions with dots, it may be that the function itself has a number of default arguments in addition to a FUN argument. When the function is called, it looks at the calling argument list and first checks to see whether the arguments are in the definition of the called function. If they are not, then they are passed to FUN where they will be used if possible - otherwise a message is returned telling you of 'unused arguments'.

```
# Here's a function which applies functions to a vector, using ...
myVecFunc <- function(vec,FUN,...){
  FUN(vec,...)
}

x <- sample(1:50,20)
x[7] <- NA
x
myVecFunc(x,sum)                          #  sum returns NA because there is an NA
myVecFunc(x,sum,na.omit = TRUE)           #  sum returns NA - na.omit not arg
myVecFunc(x,sum,na.rm = TRUE)             #  sum removes NA, computes sum
myVecFunc(x,sort)                         #  default sort removes NA
myVecFunc(x,sort,na.last=TRUE)            #  argument puts NA last in sorted list
```

In R, a function can return any object. If more than one object is to be returned it is done in the form of a list. This example is a little tricky. Spend some time making sure you understand how it works.

```
## Here's an example making use of default arguments and showing the effect of
## different ways of calling the function
gg5 <- function(a = c(1, 1, 1), b = c(1, 1, 1), f = c(1, 1, 1),text="Comparisons"){
  # All arguments have a default value
   d <- (a == b )
   e <- (a == f)
   h <- (b == f)
   LL <- list(text = text, aEqb =d, aEqf = e, bEqf = h)
   return(LL)
 }
a1 <- c(1, 1, 2)
b1 <- c(1, 1, 2)
f1 <- c(1, 1, 2)
mm <- gg5(a1,f=c(1,2,1))
mm
```

You can use your own functions in base functions which allow functions as arguments.

```
## User defined functions as arguments of basic functions

# Here is a user defined function
myF1 <- function(x, y=1){
  cat("User entered x = ", x, ", y = ", y, "\n")

  mean(x)*y
}
bb <- cbind(sample(1:33,10,replace = T),1:10)
apply(bb, 2, myF1)          # user function with default value of y
head(bb)
apply(bb, 2, myF1, y=2)      # User function with y = 2
```

Note the use (for example purposes) of the `print()` function and the more flexible `cat()` function. Both are very useful when printed output from functions is needed.

Click here to go to the CRAN web page for more information and examples on writing your own functions.

19.5 Accessing and using your functions and packages and their data

Having created your own useful function, you will want to save it as a file for later use. Conventionally, files with R code are given the extension .r or .R. It is usually convenient to give the file the same (meaningful) name as the function.

Suppose you have an R function called birthdays, which does something clever with birthdays and it is stored in a file called birthdays.r. Subsequently, when you want to use the function, firstly check whether its in your working directory (type dir() to get a listing of files in your working directory). If it's there, you simply use the command `source('birthdays.r')` to load your function into your session's environment, so that it is ready for use. It won't be shown in your script window, so if you want to look at it, you'll have to open it. If it's in another directory, you'll have to change your working directory to that one to source it, or include the path to it in the source command. Alternatively, if it's not too large, copy it into the working directory and source it there as above.

```
## Sourcing your own functions
dir()
source('../Resources/birthdays.r')
birthdays("Lisa")
```

If you create lots of related useful functions to which you want to have ready access, you may want to create a package - which is a special documented container for such 'libraries' of functions. The CRAN web-site has information about creating packages and the standards which should be applied to that activity.

Chapter 20

Tutorial 10 - Misc functions

Brian Williams <bjw649@gmail.com>

20.1 Preliminaries

Go to the Code folder and open the courseCode.R file. Find "Tutorial10".
Set the working directory to that of the source file.

Chapter 21

Lecture 11- Control structures

Brian Williams <bjw649@gmail.com>

21.1 Summary

21.2 Logistics

Open RStudio. Open courseCode.R. Find "Lecture11".

21.3 Control structures

Computational investigations almost always involve repetition of many related calculations. The process is often more formally referred to as **iteration**. The repetitions are usually controlled by structures which are broadly known as 'loops'. Within these structures there is often a need to assess data by some logic and perhaps perform different computations on different forms of data. This assessment is undertaken in "if" structures and in modern computer languages, we see things like:

"if.. (some logical condition).. then ..(some computation).. else.. (another computation).. end if"

Hadley Wickham's "R for Data Science" has a particularly good chapter on iteration here. In the next couple of sections we will look at the looping structures and if structures which are available in R.

21.4 Loops

Loops are a fundamental component of all computer programming languages. They allow straightforward repetition of computations. We shall see that there are a couple of ways of controlling the way the repetition is undertaken. Like many other programming languages, R has `for` loops, `while` loops and `repeat` loops. The key words `break` and `next` play an important role in looping structures, especially with `repeat`, which has no pre-defined exit! Have a look at R Language definition - looping

Because R is fundamentally 'vectorized' in its structure, its internal code for vector operations is extremely efficient and while programmed loops are sometimes necessary, it is important to always be aware of the much greater efficiency which is potentially available with vector operations, if the same result can be achieved that way.

However, let's begin with the `for` loop.

21.4.1 For loops

The syntax of a `for` loop is as follows:

```
for(name in range) body
```

The example below is typical, but the use of the ':' operator is possibly dangerous if the range is defined with variables.

```
##For loops
#-------------------------------------------------------------------------------
# Throw 2 dice 10000 times
n <- 100000
s1 <- Sys.time()
#result <- numeric(n)                                                    #1.
for (i in 1:n) {                                                        #2.
  result[i] <- sum(sample(1:6,2,replace=T))                             #3.
}
diff <- Sys.time() - s1
diff
df <- data.frame(result = result)
ggplot(df) + geom_bar(aes(x = result), fill = "light blue")
```

Notes

1. The vector `result` must be initialized before its first use.
2. This is a typical numeric example in which the block surrounded by {}'s will be executed 100000 times, unless there is a control (e.g. a `break` or a `return` - later in this session) inside the loop preventing completion. The loop index 'i' is often useful to refer to elements of a vector as is done here.
3. `sample(1:6,2,replace=T)` returns a two element vector of sampling from 1:6, these two values are then summed.

Here's the preferred way of writing the same code, if for some reason, variables are to be used to define the sequence limits:

```
# Throw 2 dice 100000 times
n <- 100000
s1 <- Sys.time()
result <- numeric(n)                                                    #1.
for (i in seq_along(1:n)) {           ###See R for Data Science!!                    #2
  result[i] <- sum(sample(1:6,2,replace=T))                             #3.
}
diff = Sys.time()-s1
diff
hist(result)
df <- data.frame(result = result)
ggplot(df) + geom_bar(aes(x = result), fill = "light blue", bins = 11)
```

Here's another example, this time of a loop through a string sequence. (But we do have much better ways of getting this result, of course!)

```
#   looping through a string sequence

bP <- read.table(file="../Data/BackPain.csv",sep=",",header=TRUE)
bP <- na.omit(bP)     #Remove all the rows with NA's
str(bP)
countries <- levels(bP$country)
```

```
dfprint <- data.frame(Country = countries, RespondentNumbers= integer(6))        #1.
for (ic in seq(countries)){                                                      #2.
  dfprint[ic,2]<- nrow(bP[bP$country == countries[ic],])                         #3.
}
# Now lets add the total numbers to the data frame. A little trickier because
# it's a new level in our country factors
totalNumbers <- nrow(bP)
dfprint$Country <- factor(dfprint$Country, levels = c(levels(dfprint$Country),
                                                      "Total"))                  #4.
extraRow <- data.frame(Country = as.factor("Total"),RespondentNumbers = totalNumbers)
dfprint <- bind_rows(dfprint,extraRow)                                           #5.
print(dfprint)                                                                   #6.

#--------------------------------------------------------------------------------
```

Notes

1. The new data frame is initialized with the names of the SAGE countries and an empty integer vector.
2. Here we see a loop on a sequence of factor levels (a vector of character strings).
3. Now we get the length of columns in which only rows including the ic'th country are counted.
4. Here we add a new row with 'Factor' Total. Firstly, we add the additional factor level - otherwise we'll get an error when we try to add the new row.
5. Now we can add the new row using `bind_rows()`.
6. Finally, we can print out our newly assembled data frame. (But still having R coerce!)

21.4.2 While loops

```
##While loops
# In This example we start after a miraculous sequence of three twelves (what's the
#  probability?)
# Our little script will tell us how many throws it takes to return an average
# within eps of the
# theoretical mean of 7.

nThrows     <- 3                                                                 #1.
totalThrows <- 36                                                                #2.
plotVec     <- c(NULL)                                                           #3.
delta       <- 6.                                                                #4.
set.seed(8237)  #This ensures that your pseudo-random sampling is the same as mine!
eps         <- 0.001  # Arbitrarily chosen tolerance
while (abs(delta) > eps) {
   nThrows          <- nThrows + 1                                               #5.
   totalThrows      <- totalThrows + sum(sample(1:6,2,replace=T))                #6.
   meanThrows       <- totalThrows/nThrows
   delta            <- (meanThrows - 7)
   if(nThrows%%100 == 0) cat("After ",nThrows, " throws,  delta = ", delta, "\n")
   plotVec[nThrows] <- delta                                                     #8.
}
```

```
df <- data.frame(plotVec = plotVec, nthrows = 1:nThrows)
ggplot(df)+geom_point(aes(x = nthrows, y = plotVec),
                      colour= "dark blue", size = 1) +
  xlab("No. of throws") +
  ylab("delta")

cat("No. of throws to convergence: ", nThrows)
if(delta==0)
{
  cat("Converged exactly to true mean (7).[eps = ", eps)
} else {
  options(digits = 8)
  cat("Converged to ",delta, "from true mean (7). eps = ", eps )
  options(digits = 7)
}

#-------------------------------------------------------------------------------
```

Notes

1. Set the number of throws so far to our initial 3.
2. The cumulative total so far is 3 times 12.
3. Initialize a null vector in which we'll save the plotting data.
4. Set delta large enough that it won't fail the while condition when we first enter the loop!
5. When using the while statement we have to construct our own loop counter.
6. Each time through the loop the new throw is added to our original score of 36 from the first three throws.
7. With long loops this is a neat way of printing out progress at regular intervals.
8. Add the new data to the vector for plotting.

21.4.3 Repeat loops

```
##Repeat loops
#-------------------------------------------------------------------------------
#Generate an unusual sequence
kk <- 1
ii <- 1
jj <- 4
loops <- 0
repeat{                                                                  #1.
  loops <- loops + 1
  kk <- kk + ii
  if (kk%%2 == 0 | kk%%5 == 0) next                                      #2.
  kk <- kk + jj
  cat(c('loops, kk', loops, kk), sep = c("     ","    ",'\n' ))
  if(kk > 50)break                                                       #3.
}

#-------------------------------------------------------------------------------
```

Notes

1. Start of a block of repeated code.
2. The key word `next` forces the start of a new loop. If the total so far is divisible by 2 or 5, skip the addition of jj and the printing (cat) and loop again.
3. `break` finishes the repeat loop and moves execution to the command (if any) following the }.

21.5 `If` structures in R

We have already seen how logical expressions can be used in indexes of vectors. the same expressions apply in `if` structures. Here's a very simple example:

```
x <- 1
if (x == 1){
  print(x)
}
```

Run it then change x to some other number and run it again. So the syntax is an `if` followed by a conditional (logical) expression in parenthesis, followed by some other code in a block delimited by {}. To keep things tidy, use the layout above, with the opening { immediately following the `if` condition and finish with a new line for the closing }. The other stuff (as many commands as you like) goes in between.

Usually we have to provide an alternative outcome if our condition is not `TRUE`.

```
x <- 1
if (x == 1)
  {
    print("Disaster")
  } else
  {
    print(x)
  }
```

Run it then change x to some other number and run it again. NOTE: it is critical that the closing '}' of the first block is followed on the same line by the `else` . If you put the `else` on the next line, you'll get an error! Some programmers prefer the following style - suit yourself, but this one is preferred by Paul Johnson!

```
x <- 1
if (x == 1) {
  print("Disaster")
} else {
  print(x)
}
```

Then for the utimate in choices, we have the `else if`, which allows as many options as we might need and can finish with an `else` to clean up those we haven't menioned!

```
x <- 1
if (x == 1) {
  print("Disaster")
} else if (x > 10){
   print(x)
} else if (x < 10) {
  print("Have we got a logical problem here?")
}
```

. and we fix the little logical problem - so....

```
x <- 10
if (x == 1) {
  print("Disaster")
} else if (x > 10){
   print(x)
} else if (x < 10) {
  print("Have we got a logical problem here?")
} else {
  print(" x is exactly equal to 10!")
}
```

The `if` control statements can be nested as deeply as you need, but doing so can present some tricky problems in logic. It may be desirable to sketch the decision tree structure with a few simple examples on a piece of paper before you begin - and test, test, test, when you have written the code!!

21.5.1 `switch()`

Sometimes, you will have a number of different `else if` conditions and it becomes clumsy with lots of squiggly brackets everywhere. For these situations, there is a helpful function called `switch()` - check it out with `help(switch)`. The extract below is from the R Language definition.

> Technically speaking, switch is just another function, but its semantics are close to those of control structures of other programming languages.
>
> The syntax is:
>
> `switch (statement, list)`
>
> where the elements of `list` may be named. First, `statement` is evaluated and the result, value, obtained. If value is a number between 1 and the length of `list` then the corresponding element of `list` is evaluated and the result returned. If value is too large or too small NULL is returned.

```
x <- 2
switch(x, 2+2, mean(1:10), rnorm(5))
switch(2, 2+2, mean(1:10), rnorm(5))
switch(6, 2+2, mean(1:10), rnorm(5))
```

If you want to report a default value, enclose the `switch` in an `if else` :

```
 x <- 3
if (x < 4 ){
  switch(x, 2+2, mean(1:10), rnorm(5))
} else {
  print('Default value for switch')
}
```

Test it with values of x. You can also use characters rather than numbers in `switch()` - have a look at the R Language definition - switch.

Chapter 22

Tutorial 11 - Practice with programming

Brian Williams <bjw649@gmail.com>

22.1 Preliminaries

Go to the Code folder and open the courseCode.R file. Find "Tutorial11".
Set the working directory to that of the source file.

Chapter 23

Lecture 12- Times and dates, Debugging

Brian Williams <bjw649@gmail.com>

23.1 Preliminaries

Go to the Code folder and open the courseCode.R file. Find "Lecture12.R". Save it in myRcode and set the working directory to that of the source file.

23.2 Dates

Using dates in any environment (place, computer program) is tricky. The order of days, months etc. varies customarily from place to place, and various platforms use different starting dates for computation of dates and times using numerical representations. Leap years need attention, but that's not too difficult.

Time however is much more complicated than that. We have to deal with timezones, daylight saving adjustments (leading to days with 23 hours or 25 hours) and there are even minutes with an extra second (a 'leap second') occasionally, resulting from adjustments for the slowing of the earth's rotation. We don't have time to deal with all the complications here, but we'll consider a few examples of conversions from string formats to and from Date objects.

There are functions for manipulating dates in the base package, but the `lubridate` package 'masks' (overrules) some of these and is a little easier for string conversion and it also has lots of other tools for more complex date and time tasks.

Firstly, however, try typing `?base::Date()` directly into the console window. You'll see a number of links appear in the Help window. Type `help(date)` and you will see some straightforward documentation in the Help window (bottom right). Be aware that `lubridate::date` has precedence over `base::date`, when the `lubridate` package has been loaded. `help(as.Date)` gives Help information about converting character dates to objects of class Date, when using the `base` package. We are going to look at using the `base` package first (because you will often see it on the web) and then we'll look at the easier conversion of strings to dates in the `lubridate` package.

```
##Standard Dates (from the base )
#-------------------------------------------------------------------------------
date()   # Character string for date - not so good for manipulation
str(date())
z <- Sys.time() # Standard POSIX time and date class - time stored as seconds from ?
z
startP1 <- as.POSIXct("1970-01-01 1:0:01")   # allowing for 1 hour ahead of
# Greenwich (UTC)
startP1
```

```
as.numeric(startP1)
str(z)
z <- z+ 86400
z
#-------------------------------------------------------------------------------
```

Let's have a slightly closer look at how date formats are dealt with. It's easiest to begin by looking at a few examples and you can fairly quickly pick out the patterns. Run the following code a few lines at a time and then have a look at the explanation below.

```
mbd <- "9.05.1945"                      # Perhaps this came from a .csv file
as.Date(mbd, format = "%d.%m.%Y")
jbd <- "09/14/51"                       # This one from some other source
as.Date(jbd, format = "%m/%d/%y")
tbd <- "5-June-1978"                    # ...and a third one
tbD <- as.Date(tbd, format ="%e-%B-%Y")
str(tbD)

# And converting a Date back to some other character format:
strftime(tbD, format = "%d.%m.%Y" )
as.numeric(tbD)                       # the number is the number of days after....?

# ...and R's default 'origin'
as.numeric(as.Date("1-1-1970", format ="%e-%m-%Y"))

# Today's Date:
Sys.Date()
strftime(Sys.Date(), format = "%e-%B-%Y")   # specify a format
```

The `as.Date()` function converts the text input (first argument) using a specified format string (a named argument, the second in these examples). We see that the separators are all repeated in the format string, and that apparently special characters are used to indicate the form of the time period. 'Y' is used for a four-digit reference to the year, 'y' for a two digit year. We see 'B' for the unabbreviated name of the month, and 'd' is used for a two digit representation of the day of the month, but 'e' is used if the leading zero is to be replaced with a 'space'.

If you enter `help(strptime)` or `?strptime`, you will see all the possibilities, including representation of time of day. You will also see how to use either `strptime()` to convert character representations into objects off class "POSIXlt" and `strftime()` to convert either Date objects or "POSIXlt" objects back to a specified character format.

23.3 The package `lubridate`

If you are dealing wth time series in any depth at all, this package is a very helpful one. Dates and times are very tricky! This package simplifies the format specifications a little - though you are likely at some time to have to return to the standard repesentations.

23.3.1 Converting character strings to dates

The package **lubridate** uses helper functions to convert strings to dates. These functions 'parse' your string, once you have specified the order of day, month year - and you do this by calling a function named with the corresponding sequence of "d", "m", "y". Here are some examples.

```
library(lubridate)
dmy("8-May-1945")                                    # It's VE day.
dmy("8:May:1945")
dmy("8/May/1945")
dmy("8/05-1945")
dmy("8-5:1945")
VE_day <- mdy("5-8:1945")
mdy("5-8:1945")
mdy("5-8:1945")
ymd("1945/May/8")
ymd("1945/05-08")
ymd(19450508)
```

Note the last one. You can enter an entirely numeric character string without quotation marks.

Adding time is straightforward - simply add an underscore and then one or more of the letters "h", "m", "s" as appropriate.

```
dmy_hm("16/02/2017 11:00")                           #(straight from one of my Excel spreadsheets)
dmy_hm("14/July/1977 21:45")                         # Does anyone know who was born at that time?
datetime <- dmy_hm("14/July/1977 21:45")
str(datetime)
```

Notice that the object returned by these `lubridate` fuctions is an international standard POSIXct date-time.

From these POSIXct objects, you can also extract the year, month or day (`yday` is day of the year, `mday` is day of the month, `wday` is the day of the week [Sunday is 1], numerically)

```
wday(today())
mday(today())
wday(datetime)                                       # What day of the week was she born?
month(VE_day)
```

Unfortunately, we don't have time to explore `lubridate` further, but there is a nice vignette available:

```
vignette("lubridate")
```

...and there is a chapter on `lubridate` in "R for Data Science" [**Wickham2016**].

23.4 Debugging

Wickham in "Advanced R" [**Wickham2015**] writes:

> Debugging is the art and science of fixing unexpected problems in your code.

and in Wikipedia, we have...

> Debugging is the process of finding and resolving bugs or defects that prevent correct operation of computer software or a system.

Debugging code can take almost as long as planning and writing the code in the first place. Bugs take many forms. For example, some result from simple typographical errors, others from small errors in logic in translating a mathematical concept into computer code -and yet others come from unplanned circumstances in the use of the code.

Some, like the last type, may have immediately obvious approaches to a solution of the problem, while others may take a painstaking, line-by-line checking of the code. In the past, common practice was to print results as variables changed (sometimes line-by-line) but modern software generally has, as part of the development environment, some kind of debugging facility. RStudio is no exception and the recent versions have shown substantial improvement in this capability.

We'll begin by taking a brief look at conditions (errors, warnings and messages) and then we'll work through some simple debugging exercises in the RStudio development environment.

23.4.1 Anticipated problems - using condition functions

Not all of the problems encountered in running programs are unexpected and the prudent programmer works very hard to anticipate possible problems in advance so that suitable circumvention can be arranged *before* problems occur.

There are three immediately useful functions for dealing with these anticipated problems.

- `stop()` terminates execution and allows presentation of a message to the user.
- `warning()` generates a warning message to the user when an unlikely and possibly dangerous, but not illegal, activity has been undertaken.
- `message()` is used for informative messages - not necessarily indicating potential problems.

The advantage of `warning()` and `message()` over `print()` or `cat()` for this purpose is that `warning()` and `message()` can be easily 'turned off' with the functions `suppressMessages()` and `suppressWarnings()` respectively.

The leading object arguments in `warning()` and `message()` are coerced to character and pasted together without separators, but blanks can be included at the end of character strings.

```
x <- 6
warning("Jenny","Stewart",x < 10)
warning("Jenny ","Stewart ",x < 10)
message(x)
```

23.4.2 Debugging with RStudio

We shall see that RStudio, while not as advanced as some programming development environments, offers a range of very useful methods for debugging.

There is a nice introduction to debugging in RStudio at this link.

Example 1

This example is a very simple script which defines a function 'discrim' that checks whether the discriminant in a quadratic function is > 0. It returns TRUE or FALSE accordingly.

```
# Computing discriminant for solution of quadratic equation
discrim = function(x) {
  A = x[1]
  B = x[2]
  C = x[2]
  tf <- B^2 > 4*A*C
    return(tf)
}
```

In the next chunk, four test values for the coefficients A, B, C are loaded into the columns of a matrix, X and then `apply()` is used to test the discriminant for the data in each column

```
library(knitr)
# Four example coefficients
x1 <- c(3,4,1)      #a = 3, b = 4,    c = 1
x2 <- c(1,2,1)
x3 <- c(1.1,4,3)
x4 <- c(1,2,2)

X <- as.matrix(cbind(x1,x2,x3,x4))  # Load the test values into the columns of X

dd <- apply(X,2,discrim)            # test the four example sets
xt <- t(X)                          # transpose the matrix
colnames(xt) <- c("A", "B", "C")    # Name the matrix columns
xt
df1 <- data.frame(xt, Discrmt_gt_0 = as.logical(dd)) # Add a test result column
kable(df1, align = 'lccr', digits = 1)
```

The function `apply` is used to run discrim over each of the four columns, resulting in four logical values. In order to tabulate the result we have to transpose the matrix so that the results can be added as a new column. (We could not add them as a row prior to transposition, because they are of different type (logical) to the other elements in the columns (numeric).) We change the default names of the columns of the transposed matrix to A,B and C and then add on the column with the results, creating a data frame in the process. The result, however, is clearly incorrect. The x1 vector, for example, gives $4^2 > 4 * 3 * 1$ which is clearly TRUE.

Let's put the `discrim` function in a separate file.

Copy the discrim function from the chunk above starting with the comment "Computing discriminant....." as first line and finishing with the '}', into a new script file (.R file).

Now save the file as `discrim.R`.

Now with the discrim.R tab active in the Edit pane, the Source tool at the right hand end of your tool bar in the Script pane will be visible. CLick on it. This will run the file, and you will see the function in the function list in the Environment Pane.

Now we know the code is not right. Somehow discrim is producing the wrong answer. So we'll make the script stop inside the function during the computation so that we can check what is happening. This is done by adding a 'break point' to one of the lines in the function.

Click to the left of the number 3 at the left hand side of Line 3 (the line with $A = x[1]$). A filled red dot indicating a break point should appear. You are now ready to debug when the code enters the function discrim and stops prior to executing this line.

Now put the cursor on the line which assigns dd to the output from the `apply` function and press, [Ctrl-Enter]

Wow! We have a new pane called 'Traceback and the Console window has a different prompt and four new tool buttons including a red 'Stop' button, there is new Data and Values in the Environment pane and in the Script window our breakpoint line is highlighted in yellow and there is a green arrow pointing to it.

In the Console Pane, clicking on the Tool named 'Next' steps through the code in discrim.R - you can go to the return statement and then you will have the value of tf. The problem is obvious if you look at the values of A, B and C.

In the Script window, select the right hand side of line 5,i.e. $B^2 > 4 * A * C$. Copy and paste it into the Console window - press 'Return'. The calculation is carried out. This pane is in debug mode and can be used to explore relationships in your code using the variables' current values. These values are shown in the Environment pane.

Until now, most of the Debug menu has been 'greyed' out and has been unusable. No more. Now you can use it to proceed stepwise through your code, watching how the variables change as you go. You can take short cuts by selecting 'Finish Function/Loop' and if you are finished you can select 'Stop Debugging' (or click on the red 'Stop' button in the Console.) The buttons in the console window do similar things.

The problem with our code is evident already of course, so we need not proceed further.

Experiment with the debugger on some small scripts and you will soon see how valuable it can be. In more complicated code, it can be very difficult to uncover logical errors and some can be very subtle. With a good debugger, like this one, all things are possible!

Chapter 24

Tutorial 12 - Practice L12 material

Brian Williams <bjw649@gmail.com>

24.1 Preliminaries

To be advised in class.

Chapter 25

Lecture 13 - Reproducible research,Packages - the future?

Brian Williams <bjw649@gmail.com>

25.1 Reproducible research

The ideas of reproducible research (RR) have been around for a long time(Knuth, 1988, Gentleman,2004) [**Knuth1984**, **Gentleman2004**], but there has been a great deal of interest recently because of some notable failures in reproducibility. There have been a number of high-level meetings of government funding agencies, national science bodies and high-impact journal publishers. Take a look at the following links:

- Nature (Jan 2016)
- US National Institute of Health (Oct 2015)
- PLOS Biology (Jan 2016)
- J.Biomed Inf (2016)
- Science (Jun 2015)

The philosophy of reproducible research is entirely in line with the notions in science of repeatability of experiments but there is still discussion about exactly what constitutes 'reproducible research' and the problem is compounded when we take the broad view of research and include experimentation, collection of data etc rather than our rather narrow focus on the process of data analysis. We *will* focus on reproducible computations however, since that is the business at hand.

Before going any further we should at least turn to Knuth's view of *Literate Programming*. He says [**Knuth1984**]:

> "Let us change our traditional attitude to the constuction of computer programs: Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do."

The thrust of Knuth's discussion is that we write a document explaining our analysis and present the code to demonstrate it within the explanation. The code is said to be 'weaved' into the explanation. So let's for our purposes propose a definition.

> *Reproducible research is a form of electronic documentation incorporating both explanation and actual computational code (together with its associated data and computational results) so that a third party can themselves access data and code and with the help of the explanation, repeat the computations and verify that the results so obtained match those in the document.*

Despite the 'hype' I think it is fair to say that there has not been as much reproducible research published in the bio-medical literature as one might have liked (but there is reason to hope that this situation may change in the next few years). There have been many publications in other areas of computation [see for example Xin Li's collection], described by their authors as reproducible research, and it is probable that a very large fraction of them do constitute descriptions of analytical and statistical processes which, if carefully undertaken as prescribed, will reproduce the reported results.

However...

There are others, which would once have done so, but there have been historical changes in either the underpinning data-set or dependent software libraries, which result in three possible unsatisfactory outcomes.Here is another site offering a number of articles of 'reproducible research'. MD Anderson Cancer Center.

- The process cannot be completed as described.
- The process can only be completed with some modification, introducing some doubt as to the validity of the reproducibility.
- The results differ from those reported.

25.2 For how long should it be reproducible?

Maintenance of reproducibility in electronic documentation of RR over time is a significant issue. In rapidly developing technologies, such as genomic analysis, database formats change as well as underpinning software. The Bioconductor site has probably the most comprehensive testing routine of any similar repository - but still after some time, you can expect difficulties. One significant aid is to incorporate RR in a package and submit it to Bioconductor where it will be tested daily for compatibility of supporting packages and databases.

One could argue, too, that in areas of rapid advancement, there is no point in maintaining long-term reproducibility (Not an argument that I would *entirely* agree with.)

We need to consider the context. For some of these, you may require support from computer sciene specialists. Here are the three most likely scenarios with my recommended approaches (in square brackets) from the list above and a minimum maintenance period.

1. Reporting to senior colleagues (research supervisors), [4,5], 1 year
2. Collaborating with others (possibly remotely)[5], 3 years
3. Preparing a document for publication to a wider audience (Journal, Book etc)[5,6], 5-10 years, depending on the rapidity of developments in the research environment.

25.3 What kinds of documentation have been presented?

Modern documentation of reproducible research comes largely in two forms. The first seeks to demonstrate that the results in a published document can be reproduced by a third party with the information - code, data, secondary explanations that is supplied. In the second, the authors provide software and data to reproduce the paper in its entirety. I will refer to code compendiums[**Gentleman2004**] (code, associated packages and data) that allows reproduction of the paper including code and results of evaluation of the code as *literate code*.

Here's a list of possible approaches

1. A careful explanation, step by step of the analysis is provided along with a data set. No code for the analysis. (Probably no longer acceptable as RR.)
2. A collection of code and data files along with an explanation of how they may be used to reproduce the results shown in the original publication.

3. A collection of files including perhaps code, data, external images and literate code are presented along with a *Makefile*, allowing reproduction of the results and original document in an operating environment supporting Makefile useage (Unix).

4. A zipped collection of folders containing code, data and possibly literate code for production of a paper.

5. The previous two can be a *GitHub repo*, from which the files can be extracted, or from which a package can be installed using the package `devtools` see Tutorial 13, Chapter 26.

6. An R package, possibly on the Bioconductor site for on-going automated testing.

25.4 How should we do it?

firstly, we need a platform on which the sharing of code and data can easily take place. We'll talk a little more about Git/GitHub in the Tutorial, 26, but it certainly provides a framework allowing this kind of collaborative activity.

The next thing to consider is the software you plan to use for your literate code. In R there are two main choices, both strongly supported by the RStudio environment. The first uses the package `knitr` to allow the inclusion of chunks of R code in LaTeX documents (these will have the extension `.Rnw` and the other option is to use R Markdown (using the R package `rmarkdown`) - this also allows chunks of R code in a Markdown document (extension `.Rmd`).

Markdown can be learnt in 20 minutes, as we have seen. LaTeX is far, far more complicated but can produce typeset documents. RStudio now through the `rmarkdown` package, gives you the option of converting your document to MS Word at the click of a button (as well as `.pdf` or HTML forms.) I strongly recommend that you continue with R Markdown for the moment and if you have a 'nerdy' side you may want to tackle LaTeX later.

LaTeX is still widely used in the sciences and especially in the mathematics communities. There are two highly recommended references. The first to get you started is free, Oetiker's *'The not so short introduction to LaTeX'*, https://tobi.oetiker.ch/lshort/lshort.pdf and secondly, the classic 'LaTeX Companion' by Mittelebach et al [22]. Be aware that there is a long steepish learning curve.

25.5 Packages for Medical/Public Health research

Finding the right selection of packages is of fundamental importance when using R and fortunately the R community has recognised the problem of having a plethora of packages and the obvious need for guidance. The CRAN Task Views (CTV's) have been set up for that purpose and provide a very useful overview of a number of subject areas and are a must-see for the beginning R user/researcher. If you look at the web page you will see that it's possible to install *all* the packages for a particular task view with a couple of simple commands. Do bear in mind that if you have a really huge library, it can become unwieldy in operations like "Check for Package Updates"!

25.6 Medical - Bioconductor

CTV's on Genetics, Phylogenetics, MedicalImaging, ClinicalTrials, ExperimentalDesign, Pharmacokinetics, Psychomtrics

Genetic analysis and experimentation as scientific endeavour has exponentially increased in the last 20 years or so, much of the development resting on statistical analysis of very large data sets. Biological systems are not mechanistic and are subject to 'random' variability which must be captured in experimental analysis with some kind of statistical characterisation. Modern experimental technology in this field often uses experimental arrays with more than a million elements. Data from a time sequence on an array of this size can rapidly outstrip computer memory. The data from the experiments must then often be compared with that in Genomic databases, where much larger datasets are encountered. Management of data on this

scale requires considerable expertise in computer science and research in this area is usually undertaken by multi-disciplinary teams.

Not surprisingly, the research community has established a resource centre for genetic statistics in R. It resides at Bioconductor.org.

The Bioconductor web site is a *'must-see'* site for anyone contemplating genomic analysis.

Here's the 'about' paragraph you'll find there and a bit of history:

> ## About Bioconductor
>
> Bioconductor provides tools for the analysis and comprehension of high-throughput genomic data. Bioconductor uses the R statistical programming language, and is open source and open development. It has two releases each year, 1649 software packages, and an active user community.
>
> The Bioconductor project started in 2001 and is overseen by a core team, based primarily at the Fred Hutchinson Cancer Research Center, and by other members coming from US and international institutions.

There is a wealth of data, code and learning resources on the Bioconductor site. The Bioconductor home page has links to various resources to help you install Bioconductor and its packages, learn how to use them, get support when you are in trouble, events (workshops etc), and developing your own packages . It also has a 'Twitter' column and a News' headline.

Communication within cross-disciplinary teams in genomic analysis is a key element to their success and the Bioconductor site provides great learning support across the range of disciplines [**Huber2015**].

One of the key elements of the Bioconductor site is its automated maintenance and testing of currency of packages and associated databases.

25.7 Big data analysis

CRAN Task Views: Cluster, MachineLearning, NaturalLanguageProcessing, WebTechnologies, HighPerformanceComputing,

Chapter 26

Tutorial 13 - Git, GitHub and R packages

Brian Williams <bjw649@gmail.com>

26.1 Preliminaries

Go to the courseCode folder and open Tutorial13.R. Save As in the myRcode folder.

Set the working directory to that of the source file.

In this Tutorial we are going to use Git and the associated GitHub.

We need to install GitHub. Doing so simultaneously installs Git and also (for Windows environments) the command-line utility Git Bash.

Go to the GitHub website. You will need to sign up (No cost!). Then follow the instructions to set up your account. This site gives you detailed help for the setup of Git - but you can probably do without it for the moment.

In order to create a very minimal package, we'll use two packages: `devtools` and `roxygen2`. Install these from CRAN.

26.2 Introduction

In Lecture 12 Chapter **??** we saw that one of the recommended methods for collaborating to produce reproducible research is to use GitHub and that a preferred way of presenting maintainable reproducible research is to create a package for data, documentation and the literate code. In this Tutorial, we'll introduce Git and GitHub and also create a minimal package. Be aware that this is only enough to get you started on these two topics. For much more information on Git you can freely download the book *Pro Git*[5] by Chacon and Straub from here, and for a brief account in relation to package development, Hadley Wickham's book, *R packages*[30] is also freely available and has a nice introductory chapter here: http://r-pkgs.had.co.nz/git.html on Git and GitHub. [The whole book is accessible at http://r-pkgs.had.co.nz/.]

26.3 Git Basics

So, what are Git and GitHub?

Git lives on your computer and GitHub lives in the cloud. The two interact very nicely to give you good version control and backup which is independent of your computer. (One more independent backup and you should be "safe".)

GitHub repositories are free. Unless you pay for a private repository, GitHub repositories are publicly available. The GitHub user community strives to develop freely available software, built collaboratively. We haven't time to go into the details here, but GitHub is an excellent book-keeper of parallel software development where a number of contributors may experiment with different approaches using a procedure called `forking`.

Is it a good thing to make your ideas so freely available? Remember the famous quotation from Sir Isaac Newton:

> 'if I have seen further it is by standing on the shoulders of giants'.

One could argue on this basis that sharing knowledge is the essence of the best science.

Git is a very efficient version control system which backs up programs under development and allows re-creation of the software at any stage of its development. Essentially, after the initial creation of files in a Git *repository (repo)*, Git incrementally stores all the changes you have made to the original document. It can re-apply these changes to the original document in sequence up to whatever stage you might choose, thus giving you the capacity to restore your program at any stage of its development.

Being local (entirely on your own computer) you may wish to develop some PhD ideas on your own and then when your package is complete, with clearly defined ownership and authorship, you can push it to GitHub for the rest of the world to see and admire!

26.3.1 Git operations

In Git we undertake a sequence of commands and a file goes through different states as these commands are executed.

Firstly, however, we need to tell Git to track the progress of the file. The Git command is `add` and we have done this already in RStudio when we `added` the two code files. Once a file has been added to Git, we recognise the file as being in a number of states:

1. A new file or a changed file is in the Git *working directory* - we call this a *modified file*.
2. A *modified file* is marked to be committed - it has become a *staged file* and is held in the *staging area*.
3. The *staged files* are *committed* (i.e. a snapshot is taken of the system, links are created to staged files and copies of all the staged files are added to the Git repository (database) which then contains *committed files*.

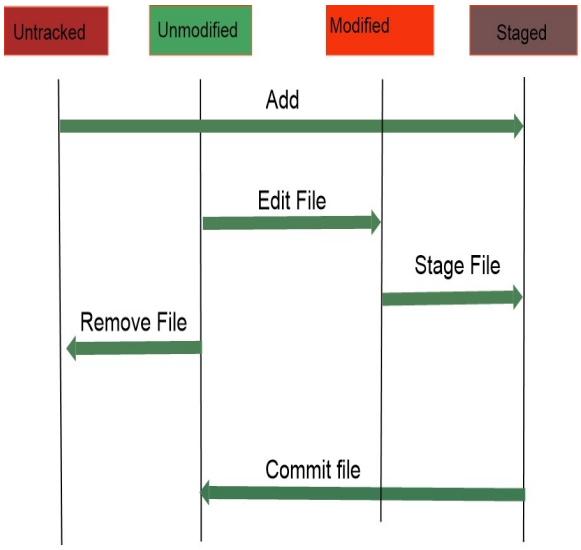


Figure 26.1: GitHub processing.

When a repository is opened, Git copies files to the working directory from the database and these may be modified, then staged and then committed.

26.3.2 GitHub

GitHub allows you to store in the cloud and you can work locally using Git. R packages can be developed using Git and Hadley Wickham's `devtools` package. GitHub allows collaborators to work in a controlled way with a shared repo.

Typically, you work locally on your computer in an R project, using Git and then when you are ready to share your package with the world, you `push` the entire project to GitHub. There, anybody can access it and install the package directly from GitHub using the command:

`devtools::install_github("GitHubUserName/packageName")`.

R packages can be developed using Git and GitHub and Hadley Wickham's `devtools` package has the capacity to convert a GitHub repo directly into a fully fledged R package.

Git and GitHub useage is integrated into `Projects` in RStudio.

So let's do it! Firstly, we'll create a RStudio project, but we will get a couple of files ready first. Here's the word-finding script we looked at in Tutorial 10.

```
   # read in the words from the file, into a vector of mode character

   txt <- scan("../Data/tf.txt","")
   tl <- length(txt)
   cat("The text file was read and there are ", tl, " words.")
   wl <- list()      # Create a list called wl with no elements
   for (i in 1:tl) {
      wrd <- txt[i]  # i-th word in input file
      wl[[wrd]] <- c(wl[[wrd]],i)
   }
print(wl)
```

Here it is in a functional form:

```
findWords1 <- function(fn = "tf.txt") {
   # read in the words from the file, into a vector of mode character
  pfn <- paste("../Data/", fn, sep = "")
  cat("pfn:  ",pfn)
   txt <- scan(pfn,"")
   tl <- length(txt)
   cat("The text file was read and there are ", tl, " words.")
   wl <- list()      # Create a list called wl with no elements
   for (i in 1:tl) {
      wrd <- txt[i]  # i-th word in input file
      wl[[wrd]] <- c(wl[[wrd]],i)
   }
   return(wl)
}
```

Let's run it to see if it still works! (and remind ourselves of what it does.)

```
findWords1("tf.txt")
```

All is well. Let's now get another of Matloff's word finders:

```
findWords2 <- function(fn = "tf.txt") {
   # read in the words from the file, into a vector of mode character
  pfn <- paste("../Data/", fn, sep = "")
  cat("pfn:  ",pfn)
   txt <- scan(pfn,"")
```

```
   words <- split(1:length(txt), txt)
   return(words)
}
```

Let's test this one, too.

```
findWords2("tf.txt")
```

Great. Now we have two code files and a data file for our project. The two code files (`findWords1.R` and `findWords2.R` are already in the Code folder in your R_Course folder, and the data file `tf.txt` is in the Data folder.

Go to [File->New Project] - select 'New Directory', then select 'R Package', then:

1. enter a package name of your choice - something unique!,
2. tick the box 'Create a git repository' and
3. tick 'Open in a New Session'.
4. Browse to choose a suitable sub-directory and
5. then add the two R code files.
6. click on Create Project.

Now we see a new RStudio window in which there is an extra Tab for Git in the pane in the top right corner and the File tab in the botttom right corner shows our code files. Before we go any further with this, let's take a quick look at the project folder - if you've forgotten where you put it, it will now be in the header of the Files Tab in the bottom right pane of RStudio's window. Open Windows Explorer and find the project folder. This is the one you named in Step 1 of the numbered process above and we will also refer to it as Git's *Working Directory*.

If you can't see a file called `.gitignore`, in the Working Directory, go to the *View* menu and tick the box, 'Hidden items'. Now, our interest will focus on the following folders and files, which have been automatically generated for our package development:

- .gitignore - see below.
- R - a folder containing the source code
- DESCRIPTION - see below
- NAMESPACE - see below

This list should not be seen as exclusive, other important folders you might expect to see in packages include:

- data
- tests
- vignette
- demo

.. they are all very important and improve the quality of your package....but we won't deal with these here. (See Hadley Wickham's book [30] or website: http://r-pkgs.had.co.nz.)

Now, let's get back to those files in the Working Directory. They are standard in a Git working directory and have been set up by RStudio for us.

26.3.3 The `.gitignore` file

Git will comment on files in the working directory which have not been added. Since some of our compiling and other processes will generate files which we don't want Git to deal with, we can name them in a file called `.gitignore`. RStudio sets this up for us with some standard intitial entries.

26.3.4 The R folder in the Git Working directory

Git can be used for any computer language or developing document. RStudio gave us an R folder on the assumption that our source documents would be R code (or literate R Code). We will add some special comments to these files soon, in order to provide 'nice' documentation when `help(package= "packageName")` is used.

Package documentation for the R file

This is done using the `roxygen2` package. Basically, all that is required from us is to have the library loaded and then modify the function with special comments and a few key words. Here's the listing for a modified version of findWords2() with the documentation added. [Our package function requires the user to specify the path to the data file.]

```
#' findwords2: elegant improvement on findwords1
#'
#' @param pfn - a text file
#'
#' @return A list of the words and their sequence number in the file
#' @export

findWords2 <- function(pfn = NULL) {
  # read in the words from the file, into a vector of mode character
  #Matloff p126 (Modified)
  cat("pfn:  ",pfn)
  txt <- scan(pfn,"")
  words <- split(1:length(txt), txt)
  return(words)
}
```

The `roxygen2` comments begin with the # symbol, followed by a single quotation mark, '. this is followed by a **single space** and then a special `roxygen2` label beginning with the ampersand symbol, @. It will help to take a look at the help page that this generates. You need to install the package for this. Here's how:

```
devtools::install_github("bjw649/testBW1")
```

We'll get back to this later, but now we can have a look at what the effect of our strange comments are when the package `roxygen2` deals with them.

```
library(testBW1)
help(findWords2)
```

If we seek more general help for the package, with `help(package = "packageName")` the DESCRIPTION folder will be accessed.

26.3.5 The DESCRIPTION folder

This is *really* important, no matter what you intend doing with your package. If it is just for personal use, two years' after you last used it, you'll need to look at the description to remember what it was all about!

```
Package: testBW1
Type: Package
Title: Test Package Including Two Word-count Functions
Version: 0.1.0
Authors@R: person("Brian", "Williams", email = "bjw649@gmail.com",
           role = c("aut", "cre"))
Description: This is a test example incorporating two simple functions which do
    word counts.
License: GPL-3
LazyData: TRUE
RoxygenNote: 5.0.1
```

This uses a different format again, but it is fairly easy to see what is going on if you look at the help file at the same time:

```
help(package="testBW1")
```

If on the help page you click on its link to DESCRIPTION file, you will see more of the detail we have included. For more information about how to format the DESCRIPTION folder take a look at the Hadley Wickham page: http://r-pkgs.had.co.nz/description.html.

26.3.6 The NAMESPACE folder

This is not important for a personal package, but important if you want to submit the package to CRAN or Bioconductor. It is more complicated than we have time to discuss once again see the book [30] and webpage http://r-pkgs.had.co.nz/namespace.html.

26.4 Creating the package

In the first instance we can create the package directly from our project in RStudio. With `devtools` installed its very simple.

1. We need to ensure that we have opened the project in RStudio, that we have set the directory to the folder containing the .Rproj file and that `devtools` is loaded.
2. Once that's all in place and our files have all been prepared as described above, we go to the Build tab in the top right pane of RStudio and check to see whether everything has been committed.
3. If not we can click on the boxes to stage modified files and then click on commit to store all the changes in the Git database.
4. Next we click on the Build tab in the same pane and then on the 'Build and Reload' button beneath it.
5. All being well, our package is created (or updated) and the new package is loaded.

Time to test, but note that if you have the `testBW1` package loaded along with your own new package, you will need to use the :: notation to ensure that you are testing your own functions and not those in the `testBW1` package.

26.5 Pushing our local repo to GitHub

To push this package to our GitHub site we need to have set up a repo on GitHub with the same name as our package/project. It is straightforward. If you do a fresh login to GitHub, you'll immediately be

offered the option of creating a new repo. Do so making sure to use the same name as your RStudio project/package.

Follow the instructions.

Go to the repo and note its URL.

Then, back to RStudio, where You will see that the Git tab offers a 'Push' button. Before you push it, you'll need to tell Git where you want to push it to. SO...

1. Check that you are still in the project directory.
2. Go to the Tools menu in RStudio and select Shell.
3. Enter the following Shell command: `git remote add origin NotedRepoURL` Where NotedRepoURL is the one you noted above! It will be something like: `https://GitHub.com/UserName/packageName`.
4. Exit from the shell.

Now you can press the Push button and push it! You'll need your GitHub Username and password to complete the push. You can go back to the GitHub project site and see all your project files and folders have been stored there (securely!).

26.6 Creating our package from Github

Remember that your free GitHub site is publicly accessible and so all your friends can install your great new package. All they have to do is install and load `devtools` and then enter the following command:

```
devtools::install_github("YourGitHubUserName/YourRepoName")
```

There's much more to Git and GitHub - they are not that hard to use from RStudio and once you become accustomed to using them , you'll find they are a very useful means of backup and collaboration.

Chapter 27

Appendix - miscellaneous extra information

Brian Williams <bjw649@gmail.com>

27.1 Relative paths to data files - Lecture 1

Consider the structure we have in the R_Course folder. It corresponds to the structure shown in Figure 27.1.

Assume we are working with Lecture1.R in the myRCode folder and we have set the working directory to the myRcode folder.

Then to find a data file called myData.csv in the Data folder, we can simply find Lecture1.R's parent (the R_Course folder), and then go to the Data folder (which is also in the R_Course folder) and find the data file there.

The relative path (from Lecture1.R to a data file called myData.csv in the Data folder is then "../Data/myData.csv". The 'two-dots' notation takes us to the parent of the current working directory (R_Course folder)

The absolute paths to the files *could* be included in the code but DON'T DO IT! It is virtually impossible to use the absolute form if the file is to be accessed by people using different computers, because everybody's absolute path would be different, depending on where they unpacked their zip file with the Course materials.

If the data file is shared from the web - then a single URL works, but we don't want a classroom of people all accessing a large file at one time!

27.2 Built-in mathematical functions

R provides all of the standard mathematical functions that we are familiar with from spreadsheets, calculators etc.

```
## Standard mathematical functions in R.

2 + 4 * 5      # Note usual order of operations (multiply has precedence)
log (10)       # Natural logarithm with base e=2.7182
log10(1000)      # Common logarithm with base 10
5^3.1             # 5 raised to the  power 3.1
5/8              # Division
8%%5             # Remainder of division
sqrt (20.25)      # Square root
abs (3-7)     # Absolute value
pi                # 3.14
```

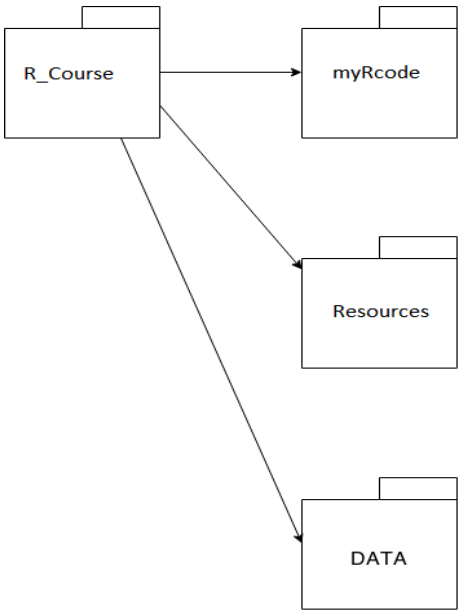


Figure 27.1: Using a common data folder.

```
exp(2)          # Exponential function i.e raising 2.7182 to power 2.
round(pi,4)  # Round pi to 4 decimal places
floor(15.9)    # Forces rounding down
ceiling(15.1)  # Forces rounding up
cos(pi/3)         # Cosine Function
sin(pi/2)         # Sine Function
tan(pi/4)        # Tangent Function
acos(0.5)/pi         # Inverse Cosine
asin(0.5)/pi*180.        # Inverse Sine
atan(0.5)/pi*180.       # Inverse Tangent
```

27.3 Probability distributions

R contains many functions for generating probability distribution data. The function names are organised consistently so that, for example, the density, distribution function, quantile function and random generation for the normal distribution are respectively `dnorm()`, `pnorm()`, `qnorm()` and `rnorm`. for the uniform distribution we would see similarly, `dunif()`, `punif()`, `qunif()` and `runif`. To see other available distributions in the Core `stats` package:

```
help(Distributions)
```

Let's have a very quick look at how these functions are used. Plot the densities:

```
x <- seq(from= -3, to = 3, by = 0.1)
y <- dnorm(x)
x2 <- seq(-3,3,.2)
y2 <- dunif(x2,-3,3)
plot(x,y, type = "l", main = "Probability density functions")
points(x2,y2, pch = 16)
```

Plot the cumulative distribution functions

```
y <- pnorm(x)
plot(x,y, type = "l", main = "Cumulative distribution functions")
x2 <- seq(-3,3,.2)
y2 <- punif(x2,-3,3)
points(x2,y2, pch = 16)
```

Plot the quantile functions

```
x <- seq(0,0.99,0.02)
y <- qnorm(x)
plot(x,y, type = "l", main = "Quantile functions")
y2 <- qunif(x)
points(x,y2, pch = 16)
```

Plot the sampling functions

```
x <- rnorm(100000)
hist(x, breaks = 30, main = "Histogram of 100000 samples using rnorm()")
x1 <- seq(0,1,1/10000)
x2 <- runif(10001)
plot(x1,x2, main = "Scatterplot of 10001 samples from runif")  # Is it really random?
```

Operators

In the segment of code above, we saw that two variables could have their values added by using the usual '+' symbol. As you might expect, R offers a number of symbols to perform such basic functions. Table 27.1, below is the list of operators found in the Language Reference Manual.

You can access a table like this easily by typing '?Syntax' (without the quotation marks) in the Console pane. The table will appear in the Help Pane (bottom right)

Constants

There are 5 types of constants, namely: logical, integer, numeric, complex, string.

In addition there are 4 special constants (which were mentioned above in the 'reserved words'.) The special constants are:

- `NULL` (an object without an assigned value),
- `NA` (a variable or object with a 'missing' value),
- `Inf` (a number whose absolute value is too large for numeric representation - typically resulting from a division by zero),,
- `NaN` ('not a number' - resulting from a division of zero by zero).

Real numeric constants are represented in the usual way:
1.3e7, 1., 1.3, 0.6, .3, 6.3e-4, 16
All of these result in a numeric constant whose type is `double`.

To assign an numeric constant value of integer type, rather than a numeric, the letter 'L' is added at the end of the integer value. For example, 178L.

String constants (sequences of characters, such as a name 'Fred') are delimited either by a pair of single quotes (') or double quotes("). If you need to include one of the pair in the string, it should be preceded by a backslash(\). Alternatively, a single quote can be embedded in a string delimited by double quotes and vice-versa. For example we can assign:

-	Minus, can be unary or binary
+	Plus, can be unary or binary
!	Unary not
~	Tilde, used for model formulae
?	Help
:	Sequence, binary (in model formulae: interaction)
*	Multiplication, binary
/	Division, binary
^	Exponentiation, binary
%x%	Special binary operators, x is any valid name
%%	Modulus, binary
%/%	Integer divide, binary
%*%	Matrix product, binary
%o%	Outer product, binary
%x%	Kronecker product, binary
%in%	Matching operator for sets, binary
<	Less than, binary
>	Greater than, binary
==	Equal to, binary
>=	Greater than or equal to, binary
<=	Less than or equal to, binary
&	And, binary, vectorized
&&	And, binary, not vectorized
|	Or, binary, vectorized
||	Or, binary, not vectorized
<-	Left assignment, binary
->	Right assignment, binary
$	List subset, binary

Table 27.1: Standard operators in R

```
stringA <- "the string Henry's"
stringA
stringB <- 'the string George\'s'
stringB
```

There are a number of other special characters which need to be preceded by a backslash ("escaped"). You'll find these in the Language Reference Manual.

- `summary()` provides a useful summary of the variables in the data frame (as shown). It computes means, sd, max, min, quartiles of numeric data and for factors (categorical vairables) it provides counts of each category we have already seen the
- `str()` function which lists the structure of an object. NOTE that the `str()` command for space-saving purposes, lists variables row-by-row, though the data frame actually has them stored as columns. This data frame has many variables which are factors.
- `head()` shows the first 6 rows (records or cases) of a data frame.

27.4 Further reading

The following are useful references for learning R. The first three are already a little dated, but still contain much useful intrductory information.

Zuur, Ieno et al.[34] uses mostly simple ecological examples with small datasets. It has a strong focus on assembling and manipulating these datasets.

Adler's Nutshell book[1] contains a wealth of information including parameters used with standard plots and an introduction to the Bioconductor package (which we will discuss later in the course). (There is a new edition which includes material on accessing and analysing web data.)

Dalgaard [8] contains more basic statistical material, but has a concise well-written introduction to R and a nice compendium of functions.

The books by Kabacoff [16] (I like this one) and Lander [18] are more advanced and more recent and cover topics like ggplot2 and creating reports in Word directly from R.

Wickham's book [29] on ggplot2 is the standard reference for that material and now there is a second edition. The book by Chang [6], provides many more examples and may be an easier starting point.

Crawley's book ([7]is a 1000+ page blockbuster. I have only read the first edition, which had much information, but some of the organization and indexing of subject matter was a little poor.

Wickham's latest books [28, 30] are more advanced. The first cited gives much detail of the underlying properties of R objects and the second provides clear guidelines on how to produce your own R package.

Munzert et al [23] provide a very accessible entry to big data analytics using R packages. Nolan and Temple Lang [25] take an in-depth look at R packages for accessing XML and other web technologies. Both these books provide a broad basic background for accessing data from the web, prior to its analysis. Entry-level analysis of 'big data' is now available in many texts, a number of which use R packages for the purpose. A few examples are: Lantz (2013) [19], Forte (2015)[11] and [26]. More advanced material with R code can be found in James et al [15] and Torgo (2011)[27]. And without R code, Bishop's book [2] is a very wide-ranging coverage of machine learning.

For reproducible research the book by Xie [33] describes 'knitr', the software used to create documents containing code and Gandrud's book[12] provides a nice wide-ranging discussion of the various other software and packages which aid production of documents containing reproducible research. A newer book by Xie [32] describes the use of the bookdown package which supports creation of technical documents and books.

If you are interested in Event History Analysis, Göran Broström[4] here at Umeå University has written a book on that topic using R. For related material on survival analysis, Mills' book [21] comes well recommended (though I have not seen it myself).

Bivand's book[3] is the classic for spatial presentation and analysis in R (and there is a new edition), while Dorman[10] offers a gentler introduction. Højsgaard et al's book[14] is a thorough introduction to graphical models (including Bayesian networks) and Nagarajan et al[24] describe some of the newer methods. Gondro [13] introduces R packages used for genomic analysis.

The CRAN website http://www.r-project.org/ has a link to 'Manuals' where there is a free document called 'An introduction to R'. The website also has an extensive (144 in March, 2014) list of books on R. Browsing the web will also uncover many teaching aids and introductory courses.

Finally, and certainly not least are two recent books which have heavily influenced the presentation in these notes. Wickham and Grolemund's 'R for data science'[31], focusses exclusively on rectangular data - dataframes - and since that is by far the form of data of most interest to public health scentists, it provides an excellent basis for using R without the steep learning curve. Mailund's book [20] complements this book with somewhat more advanced material, but with a very similar philosophy.

27.4.1 Exercises

1. Use `sample()` to estimate the frequency of a 1 arising from 1 million throws of a dye. Use `runif()` in the range 0-6. What fraction of the data are in the range 0-1

2. Create a sample of 1000000 taken from a N(0,1) distribution. Compute the fraction of your deviates that are less than 2. Compare with the `pnorm()` result.

There are some more nice examples at https://nsaunders.wordpress.com/2010/08/20/a-brief-introduction-to-apply-in-r/

27.5 Exercise

Here is a little function (very slightly modified) from Kabacoff's book[**Kabacoff2015**]. Try selecting some numerical variables from the backPain data set and passing them into this function. Can you think of other statistics you might want to add to the output? It is not hard to add them!

```
# Slightly modified from Kabacoff's 'R in Action'
 mystats <- function(x, na.omit=FALSE){

if (na.omit) x <- x[!is.na(x)]
m <- mean(x)
n <- length(x)
s <- sd(x)
r <- range(x)
IQR <- IQR(x)
skew <- sum((x-m)^3/s^3)/n
kurt <- sum((x-m)^4/s^4)/n - 3
   return(c(n=n, mean=m, stdev=s, range= r, IQR = IQR, skew=skew,
            kurtosis= kurt ))
 }
 myvars <- c("mpg", "hp", "wt")
 sapply(mtcars[myvars], mystats)
```

Bibliography

[1] J. Adler. *R in a nutshell*. Sebastopol, CA: O'Reilly, 2010.

[2] C.M. Bishop. *Pattern recognition and machine learning*. New York: Springer, 2006.

[3] R.S. Bivand, E.J. Pebesma, and V. Gómez-Rubio. *Applied spatial data analysis*. New York: Springer, 2008.

[4] G Broström. *Event history analysis with R*. Boca Raton: CRC Press, 2012.

[5] S. Chacon and B. Straub. *Pro Git*. pdf. New York: Apress, 2014.

[6] W. Chang. *R graphics cookbook*. Sebastopol, CA: O'Reilly, 2012.

[7] M. J. Crawley. *The R book*. Chichester: Wiley, 2012.

[8] P. Dalgaard. *Introductory statistics with R*. New York: Springer, 2008.

[9] A. J. Dobson. *An introduction to generalized linear models*. Boca Raton: Chapman and Hall/CRC, 2018.

[10] M. Dorman. *Learning R for geospatial analysis*. Birmingham: Packt, 2014.

[11] Rui Miguel Forte. *Mastering predictive analytics with R*. Birmingham, England: Packt Publishing, 2015.

[12] C. Gandrud. *Reproducible research with R and R Studio*. Boca Raton: Chapman and Hall/CRC, 2013.

[13] Cedric Gondro. *Primer to analysis of genomic data using R*. Cham, Switzerland: Springer, 2015.

[14] Søren Højsgaard, David Edwards, and Stefan Lauritzen. *Graphical models with R*. New York: Springer, 2012.

[15] G. James et al. *An introduction to statistical learning with applications in R*. New York: Springer, 2013.

[16] R. Kabacoff. *R in action: data analysis and graphics with R*. Shelter Island, NY: Manning, 2014.

[17] D. Kahle and H. Wickham. "ggmap: spatial visualization with ggplot2". In: *The R Journal* 5.1 (2013), pp. 144–161.

[18] J.P. Lander. *R for everyone: advanced analytics and graphics*. Addison Wesley Data and Analytics Series. Upper Saddle River: Addison Wesley, 2014.

[19] Brett Lantz. *Machine learning with R*. Birmingham, England: Packt Publishing, 2013.

[20] T. Mailund. *Beginning data science in R*. New York: Apress/Springer, 2017.

[21] Melinda Mills. *Introducing survival and event history analysis*. London: Sage Publications, 2011.

[22] F. Mittelbach and M. Goossens. *The Latex companion*. Boston: Addison-Wesley, 2004.

[23] S. Munzert et al. *Automated data collection with R*. Chichester, UK: Wiley, 2015.

[24] R. Nagarajan, M. Scutari, and S. Lèbre. *Bayesian networks in R*. New York: Springer, 2013.

[25] Deborah Nolan and Duncan Temple Lang. *XML and web technologies for data sciences with R*. New York: Springer, 2014.

[26] J. Silge and R. Robinson. *Text mining with R*. Sebastopol, CA: O'Reilly, 2017.

[27] L. Torgo. *Data mining with R*. Boca Raton: Chapman and Hall/CRC Press, 2011.

[28] H. Wickham. *Advanced R*. Boca Raton: CRC Press, 2015.

[29] H. Wickham. *ggplot2: elegant graphics for data analysis*. New York: Springer, 2009.

[30] H. Wickham. *R packages*. Sebastopol, CA: O'Reilly, 2015.

[31] H. Wickham and G. Grolemund. *R for data science*. Sebastopol, CA: O'Reilly, 2017.

[32] Yihui Xie. *Bookdown Dynamic documents with R and knitr*. Authoring books and technical documents with R Markdown. Boca Raton: CRC Press, 2017.

[33] Yihui Xie. *Dynamic documents with R and knitr*. Boca Raton: CRC Press, 2013.

[34] A.F. Zuur, E.N. Ieno, and E.H.W.G. Meesters. *A beginner's guide to R*. Dordrecht: Springer, 2009.

Index