SPEAKER_NOTES.md - R

Speaker Notes for the 2016-01-12 Software Carpentry R lesson

TYPE ALL EXAMPLES AS YOU GO. THIS KEEPS THE SPEED SANE, AND ALLOWS YOU TO EXPLAIN EVERY STEP.

START SLIDES WITH `reveal-md slides.md --theme=white`

`R for reproducible scientific analysis`

SLIDE (Learning objectives)

- Welcome
- Teaching

    - Talk around slide

- **Our goal is not just to "do stuff"**

    - do it so that anyone can easily and exactly replicate our workflow and results

Introduction to R and RStudio

SLIDE (Why `R` / `RStudio` ?)

- Talk around slide

SLIDE (`R` / `RStudio` presentation)

- Live presentation section
- Everyone start up `RStudio`

Summarise windows

- Four (maybe three) subwindows:
    - Interactive `R` console
    - Editor (may be missing on startup - will appear when files are opened)
    - Environment/History
    - Files/Plots/Packages/Help

Create a working directory with version control

- **We're following practices of project management**

  - We'll create a project directory, with `Git` version control
  - Helps ensure data integrity
  - Makes sharing code easier (lab-mates, publication)
  - Easier to recover after a Christmas break

- **Create the new directory**

  - `File->New Project`
  - `New Directory`
  - `Empty Project`
  - Enter sensible name, e.g. `swc-workshop`
  - Check box for `Create a git repository`
  - `Create project`

GREEN/RED STICKY CHECK

- Describe contents of new folder
  - `.gitignore`
  - `.Rproj`

SLIDE (Best practices)

- Talk around slide

Create directory structure

SLIDE (Creating files/directories)

- Live presentation section

- **Create subdirectory for data**

  - In `Files` tab, create `data` subdirectory

- **Create new `R` script**

  - `File -> New File -> R script`
  - save in working directory with sensible name, e.g. `swc-script.R`

GREEN/RED STICKY CHECK

Version control

- **Show Git tab on right**

- **Stage files**

  - Three files shown (including `.gitignore` and the new script file)
  - Yellow status markers mean they're not in the repository
  - Click check-boxes to stage them
  - Note that **we don't version disposable output**

- **Commit files**

  - Click `Commit`
  - Describe new dialogue window
  - Show contents/changes to files
  - Add commit message ("Initialised repository")
  - Commit
  - Show commit summary
  - Exit

GREEN/RED STICKY CHECK

SLIDE (Challenge 1)

Run through challenge (5min?) - hint about editing `.gitignore`

- Right-click link on presentation and download to `data`
- Create `graphs` subdirectory in `Files` tab
- Edit `.gitignore` to add `graphs/` folder and save
- Stage `.gitignore` in Git tab
- Commit in Git tab, and add appropriate commit message
- Demo History window for Git

SLIDE (`R` as a calculator)

Interacting with `R`

- **Two ways**

  - Type commands in the console
  - Use the script editor and save the script

- **Console**

  - Output shown here
  - Good for experimentation
  - Commands 'forgotten' when you close a session

- **Script**

- Keeps record of what you did
- Easier to reproduce and share

Working at the console

- `R` shows a `>` if it is expecting input

```
> 1 + 100
[1] 101
```

- `R` shows `+` if it's waiting for completion (`Esc` to exit)

```
> 1 +
+
```

Working from script file

- Can write same commands in the script file (`1 + 100`)
  - Use `Run` to execute
  - Use `Ctrl-Enter` to execute
  - Output appears in the console
  - Show `#` comments - good practice to comment
  - More examples (order of precedence):

```
> 3 + 5 * 2
[1] 13
> (3 + 5) * 2
[1] 16
```

- Show `Source` operation: runs all script

```
> # Using R as a calculator demo
> 1 + 100
[1] 101
> 3 + 5 * 2
[1] 13
> (3 + 5) * 2
[1] 16
```

- More examples
  - scientific notation

```
> 1/40
[1] 0.025
> 2/10000
[1] 2e-04
> 5e3
[1] 5000
```

Mathematical functions

- General format: `fn(arg)`
  - autocompletion - example: `factorial(6)`

```
> sin(1)
[1] 0.841471
> log(1)
[1] 0
> log10(10)
[1] 1
> exp(0.5)
[1] 1.648721
```

Comparisons

- Return `TRUE` / `FALSE` logical values

```
> 1 == 1
[1] TRUE
> 1 == 2
[1] FALSE
> 1 != 2
[1] TRUE
> 1 < 2
[1] TRUE
> 1 > 2
[1] FALSE
> 1 <= 2
[1] TRUE
> 1 >= 2
[1] FALSE
```

- Computer representation of numbers are approximate: important for comparisons
  - Any physicists/computer scientists in the room?
  - Numbers may not be equal, but be 'the same'
  - Use `all.equal` instead of `==`

```
> all.equal(pi-1e-7, pi)
[1] "Mean relative difference: 3.183099e-08"
> all.equal(pi-1e-8, pi)
[1] TRUE
> pi-1e-8 == pi
[1] FALSE
```

Variables and assignment

- **Variables hold values**, just like in Python
- Two ways to assign variables
  - The `<-` form is more widely used
  - Consistency more important than choice

```
> x <- 1/40
> x
[1] 0.025
> x = 1/40
> x
[1] 0.025
```

- **Look at the Environment tab** automatic updates

```
> x <- 100
```

- Variables can be used as arguments to functions

```
> log(x)
[1] 4.60517
> sqrt(x)
[1] 10
```

- Variables can be used to reassign values to themselves

```
> x
[1] 100
> x <- x + 1
> x
[1] 101
```

SLIDE (Good variable names)

- Talk around slide

SLIDE (MCQ1)

- Pose question

Package management

SLIDE (Package Management)

- See what packages are installed with `installed.packages()`
  - **demo this one**
- Add a new package using `install.packages("packagename")`
  - **demo this one with** `install.packages("ggplot2")`
- Update packages with `update.packages()`
  - **demo this one**
- You can remove a package with `remove.packages("packagename")`
- To make a package available for use, use `library(packagename)`
  - **demo**

```
> ggplot()
Error: could not find function "ggplot"
> library(ggplot2)
Warning message:
package ‘ggplot2’ was built under R version 3.2.3
> ggplot()
Warning message:
In max(vapply(evaled, length, integer(1))) :
  no non-missing arguments to max; returning -Inf
```

SLIDE (Challenge 2)

Solution:

```
install.packages("plyr")
install.packages("gapminder")
install.packages("dplyr")
install.packages("tidyr")
```

Getting help for functions

SLIDE (Functions, and getting help)

- Talk around slide

- Demo: `round(3.14159)`:
  - argument: `3.14159`
  - value: `3`

```
> round(3.14159)
[1] 3
```

SLIDE (Getting help for functions)

- **Carrying on with `round()` from last slide**
- What other arguments can `round()` take?
  - Use `args(fname)`

```
> args(round)
function (x, digits = 0)
NULL
```

- Can use the `digits` argument by naming it, or not (but order matters)

```
> round(3.14159, digits=2)
[1] 3.14
> round(3.14159, 2)
[1] 3.14
```

- **Best practice**: always use the argument name
  - clearer to others
  - if function changes, order may change
  - difficult to remember the purpose of each argument, if not explicit
- What does a function do?
  - Use `?fname` or `help(fname)` to get the complete help text
  - Demo: `?round` - go through main points
- What package is my function in?
  - (i.e. I can't find it, and don't know what to install)
  - Demo: `??melt` - show that we need `reshape2`
- Is there a function that does X?
  - e.g. you know the name of a test, such as Kolmogorov-Smirnov
  - Demo: `help.search("smirnov")`, `?ks.test`

SLIDE (Where can I get more help?)

- Talk around slide

SLIDE (Asking the right questions)

- Talk around slide
- For `dput()` example use `dput(head(iris))`
- Demo `sessionInfo()`

Data Types and Structures in `R`

SLIDE (Data Structures in `R`)

- Good place to ask about pace/if a break is needed?

SLIDE (Learning Objectives)

- Talk around the slide
- **R is largely used for data analysis**
    - The management and manipulation of data depends on the type of data we have
    - A large amount of day-to-day frustration of learners comes down to problems with data types
    - It's *very important* to understand how `R` sees your data

SLIDE (Five "atomic" data types)

- Talk around slide

SLIDE (Atomic data types)

- **Create some variables in script**

```
# Some variables
truth <- TRUE
lie <- FALSE
i <- 3L
d <- 3.0
c <- 3 + 0i
txt <- "TRUE"
```

- Show equivalence of integer, double and complex

```
> typeof(i)
[1] "integer"
> typeof(d)
[1] "double"
> i == c
[1] TRUE
> d == c
[1] TRUE
> i == d
[1] TRUE
> is.numeric(i)
[1] TRUE
> is.numeric(d)
[1] TRUE
> is.numeric(c)
[1] FALSE
```

- Show other types

```
> typeof(truth)
[1] "logical"
> typeof(lie)
[1] "logical"
> typeof(txt)
[1] "character"
```

- `is.X()` tests for a data type

```
> is.logical(lie)
[1] TRUE
> is.logical(txt)
[1] FALSE
> is.integer(i)
[1] TRUE
> is.integer(d)
[1] FALSE
```

SLIDE (Five data structures)

- Talk around slide
    - more on `data.frame` in detail later

Vectors

SLIDE (Vectors)

- Vectors are the most common data structure
- Vectors can contain only one data type

- vectors also known as "atomic vectors"

- **The `c()` function**

    - `c()` is the "concatenate" function

```
> x <- c(10, 12, 45, 33)
> x
[1] 10 12 45 33
```

- **Number sequences**

    - can use `:` or `seq()` functions
    - both functions *return* vectors

```
> series <- 1:10
> series
 [1]  1  2  3  4  5  6  7  8  9 10
> series <- seq(15)
> series
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> seq(1, 10, by=0.5)
 [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0  8.5
[18]  9.5 10.0
```

- **What type is our vector?**

    - Use the `str()` (structure) function

```
> str(x)
 num [1:4] 10 12 45 33
> str(series)
 int [1:15] 1 2 3 4 5 6 7 8 9 10 ...
> is.numeric(x)
[1] TRUE
> is.numeric(series)
[1] TRUE
> is.integer(x)
[1] FALSE
> is.integer(series)
[1] TRUE
```

- **Series is `integer` type, but `x` is not**

    - The `c()` function automatically turns integers into 'real'/'double' numbers
    - To specify integers, use `L`:

```
> y <- c(10L, 12L, 45L, 33L)
> y
[1] 10 12 45 33
> x
[1] 10 12 45 33
> is.integer(x)
[1] FALSE
> is.integer(y)
[1] TRUE
```

- **Extending a vector**
  - Append new elements to a vector with `c()`

```
> x
[1] 10 12 45 33
> x <- c(x, 57)
> x
[1] 10 12 45 33 57
```

- **Character vectors**
  - You can use `c()` to create vectors from any datatype, including characters

```
> t <- c('a', 'b', 'c')
> t
[1] "a" "b" "c"
> str(t)
 chr [1:3] "a" "b" "c"
```

SLIDE (Challenge 2)

- Point out that `R` will attempt to "coerce" the datatype to be one that can represent all items in the vector.

Solution:

```
> xx <- c(1.7, 'a')
> str(xx)
 chr [1:2] "1.7" "a"
> xx <- c(TRUE, 2)
> str(xx)
 num [1:2] 1 2
> xx <- c('a', TRUE)
> str(xx)
 chr [1:2] "a" "TRUE"
```

SLIDE (Coercion)

- Talk around slide

- **DEMO**

```
> x
[1] 10 12 45 33 57
> str(x)
 num [1:5] 10 12 45 33 57
> as.character(x)
[1] "10" "12" "45" "33" "57"
> as.complex(x)
[1] 10+0i 12+0i 45+0i 33+0i 57+0i
> as.logical(x)
[1] TRUE TRUE TRUE TRUE TRUE
```

- Sometimes coercion is not possible

```
> x <- c('a', 'b', 'c')
> str(x)
 chr [1:3] "a" "b" "c"
> as.numeric(x)
[1] NA NA NA
Warning message:
NAs introduced by coercion
```

SLIDE (Useful vector functions)

- There are functions that will give information about the vector

```
> x <- 0:10
> tail(x)
[1]  5  6  7  8  9 10
> tail(x, n=2)
[1]  9 10
> head(x)
[1] 0 1 2 3 4 5
> head(x, n=2)
[1] 0 1
> length(x)
[1] 11
> str(x)
 int [1:11] 0 1 2 3 4 5 6 7 8 9 ...
```

- Vector elements can also be named (this is *similar to*, but not the same as a Python dictionary)

```
> x <- 1:4
> names(x)
NULL
> str(x)
 int [1:4] 1 2 3 4
> names(x) <- c('a', 'b', 'c', 'd')
> x
a b c d
1 2 3 4
> str(x)
 Named int [1:4] 1 2 3 4
 - attr(*, "names")= chr [1:4] "a" "b" "c" "d"
```

Factors

SLIDE (Factors)

- Talk around slide

SLIDE (Factors demo)

- **Create factor**
  - Use the `factor()` function with a vector as the argument
  - Predefined values are those present on creation
  - **Typos can give unexpected levels!**

```
> x <- factor(c('yes', 'no', 'no', 'yes', 'yes'))
> x
[1] yes no  no  yes yes
Levels: no yes
> levels(x)
[1] "no"  "yes"
> str(x)
 Factor w/ 2 levels "no","yes": 2 1 1 2 2
```

- **Ordering levels**
  - Level order may be important
  - Models expect the baseline/control to be the first level
  - By default, `factor()` orders factors alphabetically

```
> x <- factor(c('case', 'control', 'control', 'case'))
> x
[1] case    control control case
Levels: case control
> levels(x)
[1] "case"    "control"
```

- Here, `case` will be considered the baseline/control factor.
- This is not what modelling functions expect - results will be difficult to interpret.
  - Use the `levels=` argument to fix

```
> x <- factor(c('case', 'control', 'control', 'case'), levels=c('control', 'case'))
> x
[1] case    control control case
Levels: control case
> levels(x)
[1] "control" "case"
```

- **`table()` and `barplot()` functions**

  - The `table()` function can be used to tabulate the number of members of each category
  - Introduces the `Plots` tab for output

```
> expt <- factor(c('a', 'b', 'a', 'c', 'a', 'control', 'a', 'b', 'c'))
> str(expt)
 Factor w/ 4 levels "a","b","c","control": 1 2 1 3 1 4 1 2 3
> table(expt)
expt
      a       b       c control 
      4       2       2       1 
> barplot(table(expt))
```

Matrices

SLIDE (Matrices)

- **Creating a matrix**

  - Matrices are essentially atomic vectors with extra dimensions
  - `set.seed()` makes our pseudorandom numbers reproducible
  - `rnorm()` selects values from a standard normal distribution
  - Create matrix with `matrix()`, passing a vector and specifying the number of rows and columns

```
> set.seed(1)
> x <- matrix(rnorm(18), ncol=6, nrow=3)
> x
           [,1]       [,2]      [,3]       [,4]       [,5]        [,6]
[1,] -0.6264538  1.5952808 0.4874291 -0.3053884 -0.6212406 -0.04493361
[2,]  0.1836433  0.3295078 0.7383247  1.5117812 -2.2146999 -0.01619026
[3,] -0.8356286 -0.8204684 0.5757814  0.3898432  1.1249309  0.94383621
> str(x)
 num [1:3, 1:6] -0.626 0.184 -0.836 1.595 0.33 ...
```

- `RStudio` treats vectors as 'Values' and matrices as 'Data', in the environment
- `RStudio` also lets you see the matrix in the editor window (**demo this**)

```
> str(x)
 num [1:3, 1:6] -0.626 0.184 -0.836 1.595 0.33 ...
> length(x)
[1] 18
> nrow(x)
[1] 3
> ncol(x)
[1] 6
```

SLIDE (Challenge 3)

Solution:

```
> m <- matrix(1:50, ncol=5, nrow=10)
> m
      [,1] [,2] [,3] [,4] [,5]
 [1,]    1   11   21   31   41
 [2,]    2   12   22   32   42
 [3,]    3   13   23   33   43
 [4,]    4   14   24   34   44
 [5,]    5   15   25   35   45
 [6,]    6   16   26   36   46
 [7,]    7   17   27   37   47
 [8,]    8   18   28   38   48
 [9,]    9   19   29   39   49
[10,]   10   20   30   40   50
> ?matrix
> m <- matrix(1:50, ncol=5, nrow=10, byrow=TRUE)
> m
      [,1] [,2] [,3] [,4] [,5]
 [1,]    1    2    3    4    5
 [2,]    6    7    8    9   10
 [3,]   11   12   13   14   15
 [4,]   16   17   18   19   20
 [5,]   21   22   23   24   25
 [6,]   26   27   28   29   30
 [7,]   31   32   33   34   35
 [8,]   36   37   38   39   40
 [9,]   41   42   43   44   45
[10,]   46   47   48   49   50
```

Lists

- **Creating a list**

- Directly with `list()`
- By coercion with `as.list()`
- Elements indicated/recovered by double-brackets: `[[]]`
- **Numbering is 1-based - not like Python/other languages**

```
> x <- list(1, 'a', TRUE, 1+4i)
> x
[[1]]
[1] 1
[[2]]
[1] "a"
[[3]]
[1] TRUE
[[4]]
[1] 1+4i
> x[[3]]
[1] TRUE
```

- elements can be named
  - named elements can be recovered with `$` notation

```
> xlist <- list(a="SWC Workshop", b=1:10, data=head(iris))
> xlist
$a
[1] "SWC Workshop"
$b
 [1]  1  2  3  4  5  6  7  8  9 10
$data
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
> xlist$a
[1] "SWC Workshop"
```

SLIDE (Challenge 4)

Solution:

```
> my_list <- list(
+   data_types=c("logical", "integer", "double", "complex", "character"),
+   data_structures=c("vector", "matrix", "factor", "list")
+ )
> str(my_list)
List of 2
 $ data_types     : chr [1:5] "logical" "integer" "double" "complex" ...
 $ data_structures: chr [1:4] "vector" "matrix" "factor" "list"
```

Functions

SLIDE (Functions)

SLIDE (Learning objectives)

- Talk around slide
- **Why functions?**
  - You've already seen the power of functions, for encapsulating complex analyses into simple commands
  - Functions work similarly in `R` to in Python

SLIDE (What is a function?)

- Talk around slide

Defining a function

SLIDE (Defining a function)

- Talk around slide
- **Create a new `R` script file to hold functions**
  - `File -> New File -> R Script`
  - `File -> Save -> functions-lesson.R`
  - Check what's happened in Git tab
- **Write new function in script**
  - Describe parts of function:
  - *prototype* with inputs
  - code block/body
  - indentation (readability)
  - addition, and return statements

- function scope, internal variables (readability)
- assignment of function to variable
- comments (readability)

```
# Returns sum of two inputs
my_sum <- function(a, b) {
  the_sum <- a + b
  return(the_sum)
}
# Converts fahrenheit to Kelvin
fahr_to_kelvin <- function(temp) {
  kelvin <- ((temp - 32) * (5 / 9)) + 273.15
  return(kelvin)
}
```

- **Run the functions**
  - `source` the script
  - tab-completion works!
  - boiling and freezing points

```
> fahr_to_kelvin(32)
[1] 273.15
> fahr_to_kelvin(212)
[1] 373.15
```

SLIDE (Challenge 1)

Solution:

```
kelvin_to_celsius <- function(temp) {
  celsius <- temp - 273.15
  return(celsius)
}
```

SLIDE (Challenge 2)

Solution:

```
fahr_to_celsius <- function(temp) {
  kelvin <- fahr_to_kelvin(temp)
  celsius <- kelvin_to_celsius(kelvin)
  return(celsius)
}
```

INSERTED EXAMPLE

- Just as in Python, we can use `for` loops to apply a function to several values

- Avoids repetition

```
for (i in 32:100) {
  print(fahr_to_celsius(i))
}
```

- Can also apply functions to vectors

```
fahr_to_celsius(32:100)
```

- Also `if` and `if/else` statements, as in Python:

```
if (5 > 1) {
  print("condition is true")
}
```

```
if (5 < 1) {
  print("condition is true")
} else {
  print("condition is false")
}
```

- **Commit to local Git repo**

SLIDE (Testing functions)

- Talk around slide
- **Known good values**
    - water freezes at 32F/0C, boils at 212F/100C

```
> fahr_to_celsius(32)
[1] 0
> fahr_to_celsius(212)
[1] 100
```

- **Known bad values**
    - All values are fair game on Fahrenheit/Celsius, but can't go below 0K

```
> kelvin_to_celsius(-10)
[1] -283.15
```

- **We'd need to modify this for real use!**

Data Frames

SLIDE (Data Frames)

SLIDE (Learning Objectives)

- Talk around slide

SLIDE (`data.frame` s)

- Talk around slide

SLIDE (My first `data.frame`)

- **Create a `data.frame`**
    - Name columns explicitly

```
> df <- data.frame(id=c('a', 'b', 'c', 'd', 'e', 'f'), x=1:6, y=c(214:219))
> df
  id x   y
1  a 1 214
2  b 2 215
3  c 3 216
4  d 4 217
5  e 5 218
6  f 6 219
> length(df)
[1] 3
> dim(df)
[1] 6 3
> ncol(df)
[1] 3
> nrow(df)
[1] 6
> summary(df)
 id          x              y        
 a:1   Min.   :1.00   Min.   :214.0  
 b:1   1st Qu.:2.25   1st Qu.:215.2  
 c:1   Median :3.50   Median :216.5  
 d:1   Mean   :3.50   Mean   :216.5  
 e:1   3rd Qu.:4.75   3rd Qu.:217.8  
 f:1   Max.   :6.00   Max.   :219.0  
```

- Rows are named automatically, by default.
- The length of a `data.frame` is the number of columns it has
- Use `dim()` , `nrow()` , `ncol()` to get the numbers of rows and columns

- **`data.frame` s coerce strings/characters to become factors!**

- We don't always want this
- Can use the `stringsAsFactors` argument to change this behaviour

```
> str(df)
'data.frame':	6 obs. of  3 variables:
 $ id: Factor w/ 6 levels "a","b","c","d",..: 1 2 3 4 5 6
 $ x : int  1 2 3 4 5 6
 $ y : int  214 215 216 217 218 219
> df <- data.frame(id=c('a', 'b', 'c', 'd', 'e', 'f'), x=1:6, y=c(214:219),
                   stringsAsFactors=FALSE)
> df
  id x   y
1  a 1 214
2  b 2 215
3  c 3 216
4  d 4 217
5  e 5 218
6  f 6 219
> str(df)
'data.frame':	6 obs. of  3 variables:
 $ id: chr  "a" "b" "c" "d" ...
 $ x : int  1 2 3 4 5 6
 $ y : int  214 215 216 217 218 219
```

- **Show `data.frame` in Editor tab**

SLIDE (Challenge 1)

(5min)

- Solution:
  - missing quotes in `author_last`
  - missing date in `year`

```
author_book <- data.frame(author_first=c("Charles", "Ernst",
                                          "Theodosius"),
                          author_last=c("Darwin", "Mayr", "Dobzhansky"),
                          year=c(1859, 1942, 1970))
```

SLIDE (Challenge 2)

(5min)

Solution:

```
> str(country_climate)
'data.frame':	4 obs. of  5 variables:
 $ country            : Factor w/ 4 levels "Australia","Canada",..: 2 3 4 1
 $ climate            : Factor w/ 4 levels "cold","hot","hot/temperate",..: 1 2 4 3
 $ temperature        : Factor w/ 4 levels "10","15","18",..: 1 4 3 2
 $ northern_hemisphere: Factor w/ 2 levels "FALSE","TRUE": 2 2 1 1
 $ has_kangaroo       : num  0 0 0 1
```

Adding rows and columns

- **Adding a column with `cbind()`**
    - By default the column doesn't get a name
    - to provide a name, use the name as an argument

```
> df
  id x   y
1  a 1 214
2  b 2 215
3  c 3 216
4  d 4 217
5  e 5 218
6  f 6 219
> df <- cbind(df, 6:1)
> df
  id x   y 6:1
1  a 1 214   6
2  b 2 215   5
3  c 3 216   4
4  d 4 217   3
5  e 5 218   2
6  f 6 219   1
> df <- cbind(df, caps=LETTERS[1:6])
> df
  id x   y 6:1 caps
1  a 1 214   6    A
2  b 2 215   5    B
3  c 3 216   4    C
4  d 4 217   3    D
5  e 5 218   2    E
6  f 6 219   1    F
```

- Note that `caps` is a factor:

```
> LETTERS
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T"
[23] "W" "X" "Y" "Z"
> typeof(LETTERS)
[1] "character"
> str(df)
'data.frame':	6 obs. of  5 variables:
 $ id  : chr  "a" "b" "c" "d" ...
 $ x   : int  1 2 3 4 5 6
 $ y   : int  214 215 216 217 218 219
 $ 6:1 : int  6 5 4 3 2 1
 $ caps: Factor w/ 6 levels "A","B","C","D",..: 1 2 3 4 5 6
```

- **Renaming a column**
  - Use `names()` or `colnames()` to change the name of a column

```
> colnames(df)
[1] "id"   "x"    "y"    "6:1"  "caps"
> colnames(df)[4]
[1] "6:1"
> colnames(df)[4] <- 'SixToOne'
> colnames(df)
[1] "id"       "x"        "y"        "SixToOne" "caps"
```

- **Adding a row with `rbind`**
  - Add a list (multiple types across columns)
  - Need to take care that datatypes match the columns of the `data.frame`
  - *Particularly a problem with characters and factors!*

```
> df <- rbind(df, list('g', 11, 42, 0, 'G'))
Warning message:
In `[<-.factor`(`*tmp*`, ri, value = "G") :
  invalid factor level, NA generated
> df
  id  x   y SixToOne caps
1  a  1 214        6    A
2  b  2 215        5    B
3  c  3 216        4    C
4  d  4 217        3    D
5  e  5 218        2    E
6  f  6 219        1    F
7  g 11  42        0 <NA>
```

- `R` *tried to be helpful, and put a* `NA` *special value to indicate missing data*
- Two options to add the data:

- Coerce the column to be a `character` type
- Add a level to the factor in that column (mostly what we want to do)

```
> str(df$caps)
 Factor w/ 6 levels "A","B","C","D",..: 1 2 3 4 5 6 NA
> levels(df$caps)
[1] "A" "B" "C" "D" "E" "F"
> c(levels(df$caps), 'G')
[1] "A" "B" "C" "D" "E" "F" "G"
> levels(df$caps) <- c(levels(df$caps), 'G')
> str(df$caps)
 Factor w/ 7 levels "A","B","C","D",..: 1 2 3 4 5 6 NA
```

- Now we can add the row

```
> df <- rbind(df, list('g', 11, 42, 0, 'G'))
> df
  id  x   y SixToOne caps
1  a  1 214        6    A
2  b  2 215        5    B
3  c  3 216        4    C
4  d  4 217        3    D
5  e  5 218        2    E
6  f  6 219        1    F
7  g 11  42        0 <NA>
8  g 11  42        0    G
```

- But we have a problem:
  - There's an `<NA>` in the data that we don't want
  - This can happen in many different ways for real data
  - We'll deal with this in the next section

Reading in data

SLIDE (Reading in data)

- **Most of the time you work with pre-prepared data**
  - We don't often have to build `data.frame` s by hand
  - Most data likely to come from software in a standard form
  - Sometimes it's not in the best condition, though…
- **Inspecting data in file**
  - `Files` tab: navigate to data file
  - click on file

- **Discuss data**
  - Point out comma-separations (not always best choice - Euro data)
  - Point out header line
  - Inspecting the structure of the data means we can specify proper arguments in `read.table`
- **Read data**
  - Using `read.table`
  - Put in script

```
# Load gapminder data
gapminder <- read.table(
  file="data/gapminder-FiveYearData.csv",
  header=TRUE, sep=","
)
head(gapminder)
      country year      pop continent lifeExp gdpPercap
1 Afghanistan 1952  8425333      Asia  28.801  779.4453
2 Afghanistan 1957  9240934      Asia  30.332  820.8530
3 Afghanistan 1962 10267083      Asia  31.997  853.1007
4 Afghanistan 1967 11537966      Asia  34.020  836.1971
5 Afghanistan 1972 13079460      Asia  36.088  739.9811
6 Afghanistan 1977 14880372      Asia  38.438  786.1134
> str(gapminder)
'data.frame':   1704 obs. of  6 variables:
 $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
 $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
 $ gdpPercap: num  779 821 853 836 740 ...
```

- **Load a dataset from the internet**
  - Using `read.csv` - a special case of `read.table`
  - Automatically uses `sep=","` and `header=TRUE` (not all data well-behaved!)
  - Files need not be local - can use a URL for online data
  - Put in script

```
> # Load survey data
> surveys <- read.csv('http://files.figshare.com/2236372/combined.csv')
> head(surveys)
  record_id month day year plot_id species_id sex hindfoot_length weight   genus  spe
1         1     7  16 1977       2         NL   M              32     NA Neotoma albi
2        72     8  19 1977       2         NL   M              31     NA Neotoma albi
3       224     9  13 1977       2         NL                  NA     NA Neotoma albi
4       266    10  16 1977       2         NL                  NA     NA Neotoma albi
5       349    11  12 1977       2         NL                  NA     NA Neotoma albi
6       363    11  12 1977       2         NL                  NA     NA Neotoma albi
    taxa plot_type
1 Rodent   Control
2 Rodent   Control
3 Rodent   Control
4 Rodent   Control
5 Rodent   Control
6 Rodent   Control
> str(surveys)
'data.frame':	34786 obs. of  13 variables:
 $ record_id      : int  1 72 224 266 349 363 435 506 588 661 ...
 $ month          : int  7 8 9 10 11 11 12 1 2 3 ...
 $ day            : int  16 19 13 16 12 12 10 8 18 11 ...
 $ year           : int  1977 1977 1977 1977 1977 1977 1977 1978 1978 1978 ...
 $ plot_id        : int  2 2 2 2 2 2 2 2 2 2 ...
 $ species_id     : Factor w/ 48 levels "AB","AH","AS",..: 16 16 16 16 16 16 16 16 16
 $ sex            : Factor w/ 6 levels "","F","M","P",..: 3 3 1 1 1 1 1 1 3 1 ...
 $ hindfoot_length: int  32 31 NA NA NA NA NA NA NA NA ...
 $ weight         : int  NA NA NA NA NA NA NA NA 218 NA ...
 $ genus          : Factor w/ 26 levels "Ammodramus","Ammospermophilus",..: 13 13 13
 $ species        : Factor w/ 40 levels "albigula","audubonii",..: 1 1 1 1 1 1 1 1 1
 $ taxa           : Factor w/ 4 levels "Bird","Rabbit",..: 4 4 4 4 4 4 4 4 4 4 ...
 $ plot_type      : Factor w/ 5 levels "Control","Long-term Krat Exclosure",..: 1 1 1
```

- **Good point to revisit staging/committing to local repo**
    - Go to Git tab
    - Stage current script
    - Inspect with Diff - see what's changed
    - Add commit message
    - Commit

Indexing and Subsetting data

SLIDE (Indexing and Subsetting data)

SLIDE (Learning outcomes)

- **We don't always need to use all of the data**
    - There might be incomplete or inappropriate data we need to skip
    - We may only care about a subset of samples/observations
    - We typically want to run cross-validation of statistical models
- Talk around slide

Subset by index

SLIDE (Subset by index)

- **Every element in a collection is indexed**
    - Each item in a collection can be referred to by the index
    - Demonstrate with a vector:

```
> x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
> names(x) <- letters[1:5]
> x
  a   b   c   d   e
5.4 6.2 7.1 4.8 7.5
```

- We can extract elements
    - individually
    - in groups
    - as a 'slice'
- **NOTE: Elements are numbered from 1, not 0 (unlike Python)**

```
> x
  a   b   c   d   e
5.4 6.2 7.1 4.8 7.5
> x[1]
  a
5.4
> x[4]
  d
4.8
> x[c(2,4)]
  b   d
6.2 4.8
> x[1:3]
  a   b   c
5.4 6.2 7.1
> x[c(1,1,3)]
  a   a   c
5.4 5.4 7.1
```

- **Asking for an element that isn't there**
  - `x[0]` gives an empty vector
  - `x[6]` gives a missing value `NA`

```
> x[0]
named numeric(0)
> x[6]
<NA>
  NA
```

Skip/remove by index

- **Use a negative number to return all *other* elements**

```
> x
  a   b   c   d   e
5.4 6.2 7.1 4.8 7.5
> x[-2]
  a   c   d   e
5.4 7.1 4.8 7.5
> x[c(-1,-5)]
  b   c   d
6.2 7.1 4.8
```

- **Assign the result back to the original collection to *remove* elements**

```
> x
  a   b   c   d   e
5.4 6.2 7.1 4.8 7.5
> x <- x[-4]
> x
  a   b   c   e
5.4 6.2 7.1 7.5
```

SLIDE (Challenge 1)

Solution:

```
> x[-1:3]
Error in x[-1:3] : only 0's may be mixed with negative subscripts
> -1:3
[1] -1  0  1  2  3
> 1:3
[1] 1 2 3
> -(1:3)
[1] -1 -2 -3
> x[-(1:3)]
  d   e
4.8 7.5
```

Logical masks

SLIDE (Logical masks)

- Talk around slide
- **Logical mask vectors**
    - Any vector of `TRUE` / `FALSE` values the same size as the vector we subset works
    - Shorter vectors cycle round

```
> x <- c(5.4, 6.2, 7.1, 4.8, 7.5)
> names(x) <- letters[1:5]
> x
  a   b   c   d   e 
5.4 6.2 7.1 4.8 7.5 
> mask <- c(TRUE, FALSE, TRUE, FALSE, TRUE)
> mask
[1]  TRUE FALSE  TRUE FALSE  TRUE
> x[mask]
  a   c   e 
5.4 7.1 7.5 
> mask_short = c(FALSE, TRUE)
> x[mask_short]
  b   d 
6.2 4.8 
```

- Any function that generates a logical output can produce a *mask*
    - Can combine comparators with `&`, `|`, `!`

```
> x > 7
    a     b     c     d     e 
FALSE FALSE  TRUE FALSE  TRUE 
> x[x > 7]
  c   e 
7.1 7.5 
> (x > 5) & (x < 7)
    a     b     c     d     e 
 TRUE  TRUE FALSE  TRUE FALSE 
> x[(x > 5) & (x < 7)]
  a   b 
5.4 6.2 
```

SLIDE (Challenge 2)

Solution:

```
(x < 5) | (x > 7)
    a     b     c     d     e 
FALSE FALSE  TRUE  TRUE  TRUE 
> x[(x < 5) | (x > 7)]
  c   d   e 
7.1 4.8 7.5 
```

Subset by name

SLIDE (Subset by name)

- **Extracting subsets from vectors by name**
    - Can use names directly
    - Can use vectors of names
    - Can't easily skip/remove, this way

```
> x['a']
  a
5.4
> x[c('b', 'e')]
  b   e
6.2 7.5
> x[-c('b', 'e')]
Error in -c("b", "e") : invalid argument to unary operator
```

- **Can use logical comparisons**
    - `names() ==` gives a logical vector
    - `names() %in%` for multiple selections

```
> names(x)
[1] "a" "b" "c" "d" "e"
> names(x) == 'c'
[1] FALSE FALSE  TRUE FALSE FALSE
> x[names(x) == 'c']
  c
7.1
> x[names(x) == c('a', 'e')]
  a
5.4
Warning message:
In names(x) == c("a", "e") :
  longer object length is not a multiple of shorter object length
> names(x) %in% c('a', 'e')
[1]  TRUE FALSE FALSE FALSE  TRUE
> x[names(x) %in% c('a', 'e')]
  a   e
5.4 7.5
> x[!(names(x) %in% c('a', 'c'))]
  b   d   e
6.2 4.8 7.5
```

- **Can use indexing**
    - `which(names())` returns a vector of indexes
    - `==` and `%in%` as before

```
> names(x)
[1] "a" "b" "c" "d" "e"
> names(x) == 'c'
[1] FALSE FALSE  TRUE FALSE FALSE
> which(names(x) == 'c')
[1] 3
> x[which(names(x) == 'c')]
  c
7.1
> x[which(names(x) %in% c('a', 'c'))]
  a   c
5.4 7.1
> x[-which(names(x) %in% c('a', 'c'))]
  b   d   e
6.2 4.8 7.5
```

SLIDE (Challenge 3)

(5min)

- Can't use `x['a']` as it only returns a single value

Solution:

```
x[names(x) == 'a']
```

Subsets of matrices

SLIDE (Subsets of matrices)

- Talk around slide
- **Create matrix**

```
> set.seed(1)
> m <- matrix(rnorm(6*4), ncol=4, nrow=6)
> m
           [,1]       [,2]        [,3]        [,4]
[1,] -0.6264538  0.4874291 -0.62124058  0.82122120
[2,]  0.1836433  0.7383247 -2.21469989  0.59390132
[3,] -0.8356286  0.5757814  1.12493092  0.91897737
[4,]  1.5952808 -0.3053884 -0.04493361  0.78213630
[5,]  0.3295078  1.5117812 -0.01619026  0.07456498
[6,] -0.8204684  0.3898432  0.94383621 -1.98935170
```

- **Specify row and column to extract submatrices**
  - can use ranges or subset data

  - **Does not return data with same indexes!**
  - Leave a row or column argument blank to retrieve all rows or columns
  - Extracting a single row or column returns a vector
  - `R` throws an error if indexes are out of bounds

```
> m[3:4, c(3,1)]
            [,1]       [,2]
[1,]  1.12493092 -0.8356286
[2,] -0.04493361  1.5952808
> m[, c(3,1)]
            [,1]       [,2]
[1,] -0.62124058 -0.6264538
[2,] -2.21469989  0.1836433
[3,]  1.12493092 -0.8356286
[4,] -0.04493361  1.5952808
[5,] -0.01619026  0.3295078
[6,]  0.94383621 -0.8204684
> m[3:4,]
           [,1]       [,2]        [,3]      [,4]
[1,] -0.8356286  0.5757814  1.12493092 0.9189774
[2,]  1.5952808 -0.3053884 -0.04493361 0.7821363
> m[,]
           [,1]       [,2]        [,3]        [,4]
[1,] -0.6264538  0.4874291 -0.62124058  0.82122120
[2,]  0.1836433  0.7383247 -2.21469989  0.59390132
[3,] -0.8356286  0.5757814  1.12493092  0.91897737
[4,]  1.5952808 -0.3053884 -0.04493361  0.78213630
[5,]  0.3295078  1.5117812 -0.01619026  0.07456498
[6,] -0.8204684  0.3898432  0.94383621 -1.98935170
> str(m[3:4,])
 num [1:2, 1:4] -0.836 1.595 0.576 -0.305 1.125 ...
> str(m[3,])
 num [1:4] -0.836 0.576 1.125 0.919
 > m[, c(3,6)]
Error in m[, c(3, 6)] : subscript out of bounds
```

Subsets of lists

Slide (Subsets of lists)

- Talk around slide
- **Create list**
  - Inspect content

```
> xlist <- list(a="SWC", b=1:10, data=head(iris))
> str(xlist)
List of 3
 $ a   : chr "SWC"
 $ b   : int [1:10] 1 2 3 4 5 6 7 8 9 10
 $ data:'data.frame':   6 obs. of  5 variables:
  ..$ Sepal.Length: num [1:6] 5.1 4.9 4.7 4.6 5 5.4
  ..$ Sepal.Width : num [1:6] 3.5 3 3.2 3.1 3.6 3.9
  ..$ Petal.Length: num [1:6] 1.4 1.4 1.3 1.5 1.4 1.7
  ..$ Petal.Width : num [1:6] 0.2 0.2 0.2 0.2 0.2 0.4
  ..$ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1
```

- **Extract list**
  - Uses `[` operator
  - essentially slicing
  - returns a list

```
> xlist[1]
$a
[1] "SWC"
> xlist[1:2]
$a
[1] "SWC"
$b
 [1]  1  2  3  4  5  6  7  8  9 10
```

- **Extract element**
  - Uses `[[` operator
  - returns the atomic data type
  - *you can only extract one element at a time*
  - can use the element name

```
> xlist[[1]]
[1] "SWC"
> xlist[[2]]
 [1]  1  2  3  4  5  6  7  8  9 10
> xlist[[1:2]]
Error in xlist[[1:2]] : subscript out of bounds
> xlist[['data']]
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

- **Extract by name**
    - Uses the `$` operator (or `[[]]` as above)

```
> xlist$a
[1] "SWC"
```

- **Extract element contents**
    - Can subset each of the elements in the list, in the same command

```
> xlist$data[4,]
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
4          4.6         3.1          1.5         0.2  setosa
```

Subsets of `data.frame`s

SLIDE (Subsets of `data.frame`s)

- Talk around slide
- **Extract column as dataframe**
    - Use the `[]` operator - returns a dataframe

```
> str(gapminder)
'data.frame':   1704 obs. of  6 variables:
 $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
 $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
 $ gdpPercap: num  779 821 853 836 740 ...
> head(gapminder[3])
       pop
1  8425333
2  9240934
3 10267083
4 11537966
5 13079460
6 14880372
> head(gapminder["pop"])
       pop
1  8425333
2  9240934
3 10267083
4 11537966
5 13079460
6 14880372
```

- **Extract column as atomic vector**
  - Use the `[[]]` or `$` operators

```
> head(gapminder[[5]])
[1] 28.801 30.332 31.997 34.020 36.088 38.438
> head(gapminder[["lifeExp"]])
[1] 28.801 30.332 31.997 34.020 36.088 38.438
> head(gapminder$lifeExp)
[1] 28.801 30.332 31.997 34.020 36.088 38.438
```

- **Extract row/column as dataframe**
  - Use two arguments, as for matrices
  - Returns a dataframe if elements are mixed types
  - To get a column dataframe, use `drop=False` argument

```
> gapminder[1:3,]
      country year      pop continent lifeExp gdpPercap
1 Afghanistan 1952  8425333      Asia  28.801  779.4453
2 Afghanistan 1957  9240934      Asia  30.332  820.8530
3 Afghanistan 1962 10267083      Asia  31.997  853.1007
> gapminder[3,]
      country year      pop continent lifeExp gdpPercap
3 Afghanistan 1962 10267083      Asia  31.997  853.1007
> head(gapminder[, 3, drop=FALSE])
       pop
1  8425333
2  9240934
3 10267083
4 11537966
5 13079460
6 14880372
```

SLIDE (Challenge 4)

(10min)

Solution:

```
> head(gapminder[gapminder$year == 1957,])
       country year      pop continent lifeExp gdpPercap
2  Afghanistan 1957  9240934      Asia  30.332   820.853
14     Albania 1957  1476505    Europe  59.280  1942.284
26     Algeria 1957 10270856    Africa  45.685  3013.976
38      Angola 1957  4561361    Africa  31.999  3827.940
50   Argentina 1957 19610538  Americas  64.399  6856.856
62   Australia 1957  9712569   Oceania  70.330 10949.650
> head(gapminder[, -c(1:4)])
  lifeExp gdpPercap
1  28.801  779.4453
2  30.332  820.8530
3  31.997  853.1007
4  34.020  836.1971
5  36.088  739.9811
6  38.438  786.1134
> head(gapminder[gapminder$year %in% c(2002, 2007),])
       country year      pop continent lifeExp gdpPercap
11 Afghanistan 2002 25268405      Asia  42.129  726.7341
12 Afghanistan 2007 31889923      Asia  43.828  974.5803
23     Albania 2002  3508512    Europe  75.651 4604.2117
24     Albania 2007  3600523    Europe  76.423 5937.0295
35     Algeria 2002 31287142    Africa  70.994 5288.0404
36     Algeria 2007 33333216    Africa  72.301 6223.3675
```

`data.frame` manipulation with `dplyr`

SLIDE (`data.frame` manipulation with `dplyr`)

SLIDE (Learning objectives)

- Talk around slide

SLIDE (What and why is `dplyr` ?)

- Talk around slide

SLIDE (Split-Apply-Combine)

- Talk around slide
- **A general technique for reducing the amount of repetition in code**
  - good when datasets can be grouped

SLIDE (What and why is `dplyr` ?)

- Talk around slide
- **Load `dplyr`**

```
> library(dplyr)

Attaching package: ‘dplyr’

The following objects are masked from ‘package:stats’:

    filter, lag

The following objects are masked from ‘package:base’:

    intersect, setdiff, setequal, union
```

`select()` and `filter()`

SLIDE (`select()`)

- Talk around figure

SLIDE (`select()` and `filter()`)

- **`select()` keeps only the selected variables/columns**

- Note that we don't use strings for the column names

```
> head(gapminder)
      country year      pop continent lifeExp gdpPercap
1 Afghanistan 1952  8425333      Asia  28.801  779.4453
2 Afghanistan 1957  9240934      Asia  30.332  820.8530
3 Afghanistan 1962 10267083      Asia  31.997  853.1007
4 Afghanistan 1967 11537966      Asia  34.020  836.1971
5 Afghanistan 1972 13079460      Asia  36.088  739.9811
6 Afghanistan 1977 14880372      Asia  38.438  786.1134
> head(select(gapminder, year, country, gdpPercap))
  year     country gdpPercap
1 1952 Afghanistan  779.4453
2 1957 Afghanistan  820.8530
3 1962 Afghanistan  853.1007
4 1967 Afghanistan  836.1971
5 1972 Afghanistan  739.9811
6 1977 Afghanistan  786.1134
```

- **Using the %>% pipe**
    - Analogous to the `|` pipe in the shell
    - Can perform selections without specifying the `data.frame` in the function itself
    - (this *is* useful…)
    - **NOTE: Pipes let us split commands over several lines**

```
> year_country_gdp <- gapminder %>% select(year, country, gdpPercap)
> head(year_country_gdp)
  year     country gdpPercap
1 1952 Afghanistan  779.4453
2 1957 Afghanistan  820.8530
3 1962 Afghanistan  853.1007
4 1967 Afghanistan  836.1971
5 1972 Afghanistan  739.9811
6 1977 Afghanistan  786.1134
```

- **Using `filter()` to keep only some data values**
    - Filter lets us restrict rows on the basis of data content

```
> head(filter(gapminder, continent=="Europe"))
  country year     pop continent lifeExp gdpPercap
1 Albania 1952 1282697    Europe   55.23  1601.056
2 Albania 1957 1476505    Europe   59.28  1942.284
3 Albania 1962 1728137    Europe   64.82  2312.889
4 Albania 1967 1984060    Europe   66.22  2760.197
5 Albania 1972 2263554    Europe   67.69  3313.422
6 Albania 1977 2509048    Europe   68.93  3533.004
```

- **Combining `filter()` and `select()` with pipes**
    - `dplyr` makes combining selection/filtering easy, using pipes
    - Note: we don't need to define an intermediate `data.frame`
    - Note: we don't need to use clunky indexing/names

```
> year_country_gdp_euro <- gapminder %>% filter(continent=="Europe")
>        %>% select(year, country, gdpPercap)
> head(year_country_gdp_euro)
  year country gdpPercap
1 1952 Albania  1601.056
2 1957 Albania  1942.284
3 1962 Albania  2312.889
4 1967 Albania  2760.197
5 1972 Albania  3313.422
6 1977 Albania  3533.004
```

SLIDE (Challenge 1)

Solution:

```
> head(gapminder %>% filter(continent=="Africa") %>% select(lifeExp, country, year))
  lifeExp country year
1  43.077 Algeria 1952
2  45.685 Algeria 1957
3  48.303 Algeria 1962
4  51.407 Algeria 1967
5  54.518 Algeria 1972
6  58.014 Algeria 1977
```

`group_by()` and `summarize`

SLIDE (Reducing repetition)

- Talk around slide

SLIDE (`group_by()`)

- Talk round figure
    - separates out `data.frame` on the basis of values in `a`

SLIDE (`summarize()`)

- Talk round figure
    - Creates new variables that repeat over a series of `data.frame` s

SLIDE (`group_by()` and `summarize()`)

- Talk around slide
- **`group_by()` produces a "grouped `data.frame`"**
  - Not the same as a `data.frame`!
  - Like a `list` where each item is a `data.frame` whose rows correspond only to a particular value of `continent`
  - `tally()` counts up the rows in each group

```
> gapminder %>% group_by(continent)
Source: local data frame [1,704 x 6]
Groups: continent [5]
       country  year      pop continent lifeExp gdpPercap
        (fctr) (int)    (dbl)    (fctr)   (dbl)     (dbl)
1  Afghanistan  1952  8425333      Asia  28.801  779.4453
2  Afghanistan  1957  9240934      Asia  30.332  820.8530
3  Afghanistan  1962 10267083      Asia  31.997  853.1007
4  Afghanistan  1967 11537966      Asia  34.020  836.1971
5  Afghanistan  1972 13079460      Asia  36.088  739.9811
6  Afghanistan  1977 14880372      Asia  38.438  786.1134
7  Afghanistan  1982 12881816      Asia  39.854  978.0114
8  Afghanistan  1987 13867957      Asia  40.822  852.3959
9  Afghanistan  1992 16317921      Asia  41.674  649.3414
10 Afghanistan  1997 22227415      Asia  41.763  635.3414
..         ...   ...      ...       ...     ...       ...
> str(gapminder)
'data.frame':	1704 obs. of  6 variables:
 $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
 $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
 $ gdpPercap: num  779 821 853 836 740 ...
> str(gapminder %>% group_by(continent))
Classes ‘grouped_df’, ‘tbl_df’, ‘tbl’ and 'data.frame':	1704 obs. of  6 variables:
 $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
 $ pop      : num  8425333 9240934 10267083 11537966 13079460 ...
 $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
 $ gdpPercap: num  779 821 853 836 740 ...
 - attr(*, "vars")=List of 1
  ..$ : symbol continent
 - attr(*, "drop")= logi TRUE
 - attr(*, "indices")=List of 5
  ..$ : int  24 25 26 27 28 29 30 31 32 33 ...
  ..$ : int  48 49 50 51 52 53 54 55 56 57 ...
  ..$ : int  0 1 2 3 4 5 6 7 8 9 ...
```

```
  ..$ : int  12 13 14 15 16 17 18 19 20 21 ...
  ..$ : int  60 61 62 63 64 65 66 67 68 69 ...
 - attr(*, "group_sizes")= int  624 300 396 360 24
 - attr(*, "biggest_group_size")= int 624
 - attr(*, "labels")='data.frame':  5 obs. of  1 variable:
  ..$ continent: Factor w/ 5 levels "Africa","Americas",..: 1 2 3 4 5
  ..- attr(*, "vars")=List of 1
  .. ..$ : symbol continent
  ..- attr(*, "drop")= logi TRUE
> gapminder %>%
+ group_by(continent) %>%
+ tally()
Source: local data frame [5 x 2]
  continent     n
     (fctr) (int)
1    Africa   624
2  Americas   300
3      Asia   396
4    Europe   360
5   Oceania    24
```

- **`summarize()` creates summary information for each group**
    - We need to tell `summarize()` a function to apply to each of our grouped `data.frame` s
    - We also tell it a variable name to place that calculated value into
    - `summarize` returns a `data.frame`

```
> gapminder %>% group_by(continent)
            %>% summarize(meangdpPercap=mean(gdpPercap))
Source: local data frame [5 x 2]

  continent meangdpPercap
     (fctr)         (dbl)
1    Africa      2193.755
2  Americas      7136.110
3      Asia      7902.150
4    Europe     14469.476
5   Oceania     18621.609
> gapminder %>% group_by(continent)
            %>% summarize(sdgdpPercap=sd(gdpPercap))
Source: local data frame [5 x 2]

  continent sdgdpPercap
     (fctr)       (dbl)
1    Africa    2827.930
2  Americas    6396.764
3      Asia   14045.373
4    Europe    9355.213
5   Oceania    6358.983
> str(gapminder %>% group_by(continent) %>% summarize(sdgdpPercap=sd(gdpPercap)))
Classes ‘tbl_df’, ‘tbl’ and 'data.frame':   5 obs. of  2 variables:
 $ continent  : Factor w/ 5 levels "Africa","Americas",..: 1 2 3 4 5
 $ sdgdpPercap: num  2828 6397 14045 9355 6359
```

SLIDE (Challenge 2)

- Use `filter()` to get the rows you need

Solution:

```
> lifeExp_bycountry <- gapminder %>% group_by(country)
                       %>% summarize(meanlifeExp=mean(lifeExp))
> head(lifeExp_bycountry)
Source: local data frame [6 x 2]
      country meanlifeExp
       (fctr)       (dbl)
1 Afghanistan    37.47883
2     Albania    68.43292
3     Algeria    59.03017
4      Angola    37.88350
5   Argentina    69.06042
6   Australia    74.66292
> lifeExp_bycountry %>% filter(meanlifeExp == max(meanlifeExp))
Source: local data frame [1 x 2]
  country meanlifeExp
   (fctr)       (dbl)
1 Iceland    76.51142
> lifeExp_bycountry %>% filter(meanlifeExp == min(meanlifeExp))
Source: local data frame [1 x 2]
       country meanlifeExp
        (fctr)       (dbl)
1 Sierra Leone    36.76917
```

SLIDE (Group by multiple variables)

- Talk around slide
- **Use multiple variables with `group_by()`, `summarize()`**
    - Can write this in the script for sanity

```
> gdp_bycontinent_byyear <- gapminder %>%
+ group_by(continent, year) %>%
+ summarize(mean_gdpPercap=mean(gdpPercap))
> head(gdp_bycontinent_byyear)
Source: local data frame [6 x 3]
Groups: continent [1]
  continent  year mean_gdpPercap
     (fctr) (int)          (dbl)
1    Africa  1952       1252.572
2    Africa  1957       1385.236
3    Africa  1962       1598.079
4    Africa  1967       2050.364
5    Africa  1972       2339.616
6    Africa  1977       2585.939
> gdp_pop_bycontinents_byyear <- gapminder %>%
  group_by(continent,year) %>%
  summarize(mean_gdpPercap=mean(gdpPercap),
            sd_gdpPercap=sd(gdpPercap),
            mean_pop=mean(pop),
            sd_pop=sd(pop))
> head(gdp_pop_bycontinents_byyear)
Source: local data frame [6 x 6]
Groups: continent [1]
  continent  year mean_gdpPercap sd_gdpPercap mean_pop   sd_pop
     (fctr) (int)          (dbl)        (dbl)    (dbl)    (dbl)
1    Africa  1952       1252.572     982.9521  4570010  6317450
2    Africa  1957       1385.236    1134.5089  5093033  7076042
3    Africa  1962       1598.079    1461.8392  5702247  7957545
4    Africa  1967       2050.364    2847.7176  6447875  8985505
5    Africa  1972       2339.616    3286.8539  7305376 10130833
6    Africa  1977       2585.939    4142.3987  8328097 11585184
```

SLIDE (`mutate()`)

- Talk around slide
- **`mutate()` lets us create new variabls on the fly**
    - We can calculate total GDP from GDP per person, and population

```
> head(gapminder %>% mutate(gdp_billion=gdpPercap*pop/10^9))
      country year      pop continent lifeExp gdpPercap gdp_billion
1 Afghanistan 1952  8425333      Asia  28.801  779.4453    6.567086
2 Afghanistan 1957  9240934      Asia  30.332  820.8530    7.585449
3 Afghanistan 1962 10267083      Asia  31.997  853.1007    8.758856
4 Afghanistan 1967 11537966      Asia  34.020  836.1971    9.648014
5 Afghanistan 1972 13079460      Asia  36.088  739.9811    9.678553
6 Afghanistan 1977 14880372      Asia  38.438  786.1134   11.697659
> gdp_pop_bycontinents_byyear <- gapminder %>%
```

```
+   mutate(gdp_billion=gdpPercap*pop/10^9) %>%
+   group_by(continent,year) %>%
+   summarize(mean_gdpPercap=mean(gdpPercap),
+             sd_gdpPercap=sd(gdpPercap),
+             mean_pop=mean(pop),
+             sd_pop=sd(pop),
+             mean_gdp_billion=mean(gdp_billion),
+             sd_gdp_billion=sd(gdp_billion))
> head(gdp_pop_bycontinents_byyear)
Source: local data frame [6 x 8]
Groups: continent [1]
  continent  year mean_gdpPercap sd_gdpPercap mean_pop   sd_pop mean_gdp_billion
     (fctr) (int)          (dbl)        (dbl)    (dbl)    (dbl)            (dbl)
1    Africa  1952       1252.572     982.9521  4570010  6317450         5.992295
2    Africa  1957       1385.236    1134.5089  5093033  7076042         7.359189
3    Africa  1962       1598.079    1461.8392  5702247  7957545         8.784877
4    Africa  1967       2050.364    2847.7176  6447875  8985505        11.443994
5    Africa  1972       2339.616    3286.8539  7305376 10130833        15.072242
6    Africa  1977       2585.939    4142.3987  8328097 11585184        18.694899
Variables not shown: sd_gdp_billion (dbl)
> gdp_pop_bycontinents_byyear <- gapminder %>%
+   group_by(continent,year) %>%
+   summarize(mean_gdpPercap=mean(gdpPercap),
+             sd_gdpPercap=sd(gdpPercap),
+             mean_pop=mean(pop),
+             sd_pop=sd(pop)) %>%
+   mutate(gdp_billion=mean_gdpPercap*mean_pop/10^9)
> gdp_pop_bycontinents_byyear <- gapminder %>%
+   group_by(continent,year) %>%
+   summarize(mean_gdpPercap=mean(gdpPercap),
+             sd_gdpPercap=sd(gdpPercap),
+             mean_pop=mean(pop),
+             sd_pop=sd(pop)) %>%
+   mutate(mean_gdp_billion=mean_gdpPercap*mean_pop/10^9)
> head(gdp_pop_bycontinents_byyear)
Source: local data frame [6 x 7]
Groups: continent [1]
  continent  year mean_gdpPercap sd_gdpPercap mean_pop   sd_pop mean_gdp_billion
     (fctr) (int)          (dbl)        (dbl)    (dbl)    (dbl)            (dbl)
1    Africa  1952       1252.572     982.9521  4570010  6317450         5.724268
2    Africa  1957       1385.236    1134.5089  5093033  7076042         7.055054
3    Africa  1962       1598.079    1461.8392  5702247  7957545         9.112641
4    Africa  1967       2050.364    2847.7176  6447875  8985505        13.220489
5    Africa  1972       2339.616    3286.8539  7305376 10130833        17.091772
6    Africa  1977       2585.939    4142.3987  8328097 11585184        21.535946
```

Creating publication-quality graphics

SLIDE (Creating publication-quality graphics)

SLIDE (Learning objectives)

- Talk around slide

The grammar of graphics

SLIDE (The grammar of graphics)

- Talk around slide

- **Grammar of graphics is non-intuitive, but gives advantages**

  - Data and its representation handled separately
  - Means that components can be customised to a particular representation easily

SLIDE (A basic scatterplot)

- Talk around slide

```
> library(ggplot2)
> qplot(lifeExp, gdpPercap, data=gapminder, colour=continent)
```

- **Show the plot**

  - Describe features
  - x-, y-axes; colours by continent; legend
  - main features - Europe high life expectancy, Africa low GDP per capita

- **What is happening under the surface? How can you reproduce this?**

  - Convenience functions can be quick and easy, but aren't readily modifiable
  - We'd like to build plots *like* this in other situations - how can we do that?

SLIDE (What is a scatterplot? Aesthetics…)

- Talk around slide

SLIDE (What is a scatterplot? Aesthetics…)

- Talk around slide

- **Aesthetics decide where and how data are plotted**

  - They essentially create a new dataset that contains aesthetic information

SLIDE (What is a scatterplot? `geom` s)

- Talk around slide
- **`geom` s determine the "type" of plot**
  - Not all `geom` s make sense for a given dataset (though they may be 'grammatical')
  - Can combine multiple `geom` s to produce new graphs

SLIDE (`ggplot2` layers)

- Talk around slide

SLIDE (Building a scatterplot)

- **Creating a `ggplot` object**
  - Can't plot these directly
  - Can store them in variables for convenience/reproducibility

```
> ggplot(data=gapminder, aes(x=lifeExp, y=gdpPercap))
> p <- ggplot(data=gapminder, aes(x=lifeExp, y=gdpPercap))
> > str(p)
List of 9
 $ data       :'data.frame':    1704 obs. of  6 variables:
  ..$ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
  ..$ year     : int [1:1704] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
  ..$ pop      : num [1:1704] 8425333 9240934 10267083 11537966 13079460 ...
  ..$ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
  ..$ lifeExp  : num [1:1704] 28.8 30.3 32 34 36.1 ...
  ..$ gdpPercap: num [1:1704] 779 821 853 836 740 ...
  […]
```

- **We need to add a layer**
  - At minimum, use a `geom`
  - This uses the default dataset we specified in `p` , unless told otherwise
  - `geom_point` tells `ggplot2` we want to represent data as points (scatterplot)
  - We get only a scatterplot of points, but no colours

```
> p + geom_point()
```

- **We can modify aesthetics**
  - In the default dataset, or in the `geom` layer
  - Aesthetics/data in the `geom` layer override those in the default

```
> p + geom_point(aes(colour=continent))
> p <- ggplot(data=gapminder, aes(x=lifeExp, y=gdpPercap, colour=continent))
> p + geom_point()
```

SLIDE (Challenge 1)

Solution:

```
> p <- ggplot(data=gapminder, aes(x=year, y=lifeExp, colour=continent))
> p + geom_point()
```

This is not a good way to view the data - we need a new geometry!

Layers

SLIDE (Layers)

- Talk around slide
- **The last challenge representation didn't look good**
  - Change `geom` to line chart

```
> p + geom_line()
```

- This looks wrong
  - Lines connect continents, not countries (which is what we want)
- **Group data on a variable**
  - Use `by` to group data by country

```
> p + geom_line(aes(by=country))
```

- That looks better
- **Overlay a second `geom` to see datapoints**
  - Use the `+` operator to keep adding `geom` s
  - Layers are drawn in the specified order

```
> p + geom_line(aes(by=country)) + geom_point()
> p + geom_line(aes(by=country)) + geom_point(aes(colour=NULL))
> p + geom_point(aes(colour=NULL)) + geom_line(aes(by=country))
```

Transformations and statistics

SLIDE (Transformations)

- Talk around slide

- **Scaling axes**
    - Difficult to distinguish GDP on the y-axis
    - Rescale with a transformation

```
> p <- ggplot(data=gapminder, aes(x=lifeExp, y=gdpPercap, colour=continent))
> p + geom_point()
> p + geom_point() + scale_y_log10()
```

- **Transformations can be layered**

```
> p + geom_point(aes(size=pop)) + scale_size("population")
> p + geom_point(aes(size=pop)) + scale_size("population") + scale_y_log10()
```

- **Scaling colours**
    - Transformations are also how colours are 'scaled'

```
> p + geom_point() + scale_y_log10() + scale_colour_brewer()
> p + geom_point() + scale_y_log10() + scale_colour_grey()
```

SLIDE (Statistics)

- Talk around slide
- **Adding a smoother to the data**
    - Adds as another layer on the plot

```
> p <- ggplot(data=gapminder, aes(x=lifeExp, y=gdpPercap))
> p + geom_point()
> p + geom_point() + scale_y_log10()
> p + geom_point() + scale_y_log10() + geom_smooth()
```

- **Adding a KDE**
    - Adds as another layer on the plot

```
> p + geom_point() + scale_y_log10() + geom_density_2d()
```

Multi-panel figures

SLIDE (Multi-panel figures)

- Talk around slide
- **Faceting**
    - Grouping data by country, colouring by continent

- One big plot is messy, hard to read.
- Using `facet_wrap` splits out plots on groups

```
> p <- ggplot(data=gapminder, aes(x=year, y=lifeExp, colour=continent, by=country))
> p + geom_line()
> p + geom_line() + scale_y_log10()
> p + geom_line() + scale_y_log10() + facet_wrap(~continent)
```

- **Grouping on country**
    - Even the continent plots are a bit jumbled
    - Group by country just by changing the argument

```
> p + geom_line() + scale_y_log10() + facet_wrap(~country)
```

- Very hard to read in `RStudio`
- Export graph as pdf and visualise
    - Click `Export -> Save as PDF`
    - PDF Size: A4
    - Orientation: Landscape
    - File name (something sensible)
    - View plot after saving
    - `Save`

SLIDE (Challenge 2)

Solution:

```
> p <- ggplot(data = gapminder, aes(x = gdpPercap, fill=continent))
> p + geom_density()
> p + geom_density(alpha=0.6)
> p + geom_density(alpha=0.6) + scale_x_log10()
> p + geom_density(alpha=0.6) + scale_x_log10() + facet_wrap(~year)
```

Wrapping up

SLIDE (Wrapping Up)

SLIDE (Learning objectives)

- Talk around slide

SLIDE (Best practices)

- Talk around slide