

2018-09-05-dundee Software Carpentry: R lesson speaker notes

These notes are for the tutor(s) on the first morning session of the Software Carpentry course held on 5-7th September 2018 at the University of Dundee, teaching the refresher in R.

- Learning objectives
- Prerequisites
- Things to remember
 - Clearing the console in R
 - Get a 'clean' console
- SLIDES
 - TITLE: Programming in R
 - ETHERPAD
 - LEARNING OBJECTIVES
- SECTION 01: RSTUDIO
 - LEARNING OBJECTIVES
 - WHAT IS RSTUDIO?
 - RSTUDIO OVERVIEW - **INTERACTIVE DEMO**
 - BUILT-IN FUNCTIONS
 - GETTING HELP FOR BUILT-IN FUNCTIONS
 - NUMERICAL COMPARISONS
 - WORKING IN RSTUDIO
- SECTION 02: MY FIRST RSTUDIO PROJECT
 - LEARNING OBJECTIVES
 - PROJECT MANAGEMENT IN RSTUDIO
 - OBTAINING DATA
 - INVESTIGATING GAPMINDER
- SECTION 03: PROGRAM FLOW CONTROL
 - LEARNING OBJECTIVES
 - IF() ... ELSE**
 - FOR() LOOPS
 - WHILE() LOOPS
 - CHALLENGE
 - VECTORISATION
 - CHALLENGE
- SECTION 04: FUNCTIONS
 - LEARNING OBJECTIVES
 - WHY FIUNCTIONS?
 - DEFINING A FUNCTION
 - DOCUMENTING FUNCTIONS
 - FUNCTION ARGUMENTS
- SECTION 05: DYNAMIC REPORTS
 - LEARNING OBJECTIVES
 - LITERATE PROGRAMMING

- CREATE AN RMarkdown FILE
 - COMPONENTS OF AN RMarkdown FILE
 - CREATING A REPORT
 - SECTION 06: `dplyr`
 - LEARNING OBJECTIVES
 - WHAT AND WHY IS `dplyr`?
 - SPLIT-APPLY-COMBINE
 - `select()`
 - `filter()`
 - CHALLENGE
 - `group_by()`
 - `summarize()`
 - CHALLENGE
 - `count()` and `n()`
 - `mutate()`
-

Learning objectives

- Introduction/refresher for `RStudio`
 - understand what `RStudio` is
 - know the main windows of `RStudio` and what functions they provide
 - Introduction/refresher for `RStudio` and `git/GitHub` project setup
 - create a project in `RStudio`
 - use good practice for project layout in `RStudio`
 - place a project under `git` version control with `RStudio`
 - Refresher for flow control in `R`
 - understand and use `if()...else()` statements
 - understand and use `for()` loops
 - understand and use `while()` loops
 - Refresher for functions in `R`
 - understand the composition of an `R` function
 - how to call functions
 - how to write functions
 - understand when to write functions for good code structure
 - Introduction to `RMarkdown` and `knitr`
 - understand the purpose of literate programming
 - understand what a Markdown document is
 - understand and be able to use `RMarkdown` syntax
 - Good programming practice
 - good choices for variable names
 - understand the importance of good documentation
 - when and how to write comments in code
-

Prerequisites

We assume that the learners have prior exposure to many concepts:

- R
- variables and variable assignment
- R data types and data structures, especially `data.frames`
- using R packages
- R base graphics and `ggplot2`

Things to remember

Clearing the console in R

- remove all variables

```
rm(list=ls())
```

Get a 'clean' console

```
CTRL + L
```

SLIDES

TITLE: Programming in R

ETHERPAD

- **DEMONSTRATE LINK AND PAGE**
 - Please use the course etherpad to
 - make notes
 - ask questions (someone will be looking at the page)
 - share your knowledge with the rest of the class
 - relive the class afterwards
-

LEARNING OBJECTIVES

- We're being **QUITE AMBITIOUS**, but we've a lot of time this morning, so should be OK
- We're covering some **FUNDAMENTALS OF RSTUDIO**
 - **CREATING** projects and **PUTTING UNDER VERSION CONTROL**
- We're covering some **FUNDAMENTALS OF PROGRAMMING** in R, but principles that are **APPLICABLE TO ANY LANGUAGE**

- We're learning some **BEST PRACTICES FOR WRITING AND ORGANISING CODE**
 - Much of the morning session is **INTENDED AS A REFRESHER**
 - We'll be **ASSUMING YOU ALREADY USE R** so are familiar with some aspects:
 - R syntax
 - data types and data structures (e.g. `data.frames`)
 - variables, and variable assignment
 - R packages
 - R base graphics and `ggplot2`
 - **IF ANYTHING IS NEW OR UNCLEAR, PLEASE ASK STRAIGHT AWAY**
-

SECTION 01: RSTUDIO

LEARNING OBJECTIVES

- We're going to cover the **BASIC ELEMENTS OF AN RSTUDIO SESSION**
 - How RStudio **HELPS WITH LIVE ANALYSES**
 - How RStudio **HELPS WITH WRITING CODE FOR REPRODUCIBLE ANALYSIS**
-

WHAT IS RSTUDIO?

- RStudio is an **INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)**
 - available on **ALL MAJOR OPERATING SYSTEMS**
 - available **AS A WEBSERVER**
 - On the left is a Mac screenshot, Windows on the right
 - RStudio provides **PANES** so you can:
 - write **LIVE CODE** (console pane)
 - **VISUALISE AND QUERY DATA LIVE** (graphics and environment pane)
 - write **SCRIPTS AND DOCUMENTS FOR REUSE** (editor pane)
 - **MANAGE PROJECTS AND FILES** (file/git panes)
-

RSTUDIO OVERVIEW - INTERACTIVE DEMO

- **REMIND PEOPLE THEY CAN USE RED/GREEN STICKIES AT ANY TIME**
 - **(INTRODUCE RED/GREEN STICKIES IF NECESSARY)**
- **ASK PEOPLE TO START RSTUDIO**
 - There will be problems. Deal with them, now. It's OK if a couple of people are getting help when you start.



Red sticky for a question or issue



Green sticky if complete

- **DESCRIBE THE STARTING VIEW OF RSTUDIO**
- You should see **THREE PANELS**
 - Interactive **R CONSOLE**: **type here and get instant feedback**
 - **ENVIRONMENT/HISTORY** window
 - Files/Plots/Packages/Help/Viewer: **interacting with files on the computer, and viewing help and some output**
- **REMEMBER THE WINDOWS ARE MOBILE AND PEOPLE COULD HAVE THEM IN ANY CONFIGURATION - THE EXACT ARRANGEMENT IS UNIMPORTANT**
- We're going to use **R** in the interactive console to get used to some of the features of the language, and **RStudio**.
 - **THE RIGHT ANGLED BRACKET IS A PROMPT**: **R** expects input
 - Type calculations, then press **return**
- **DEMO CODE: ASK PEOPLE TO TYPE ALONG**

```
> 1 + 100
[1] 101
> 30 / 3
[1] 10
```

- **RESULT IS INDICATED WITH A NUMBER [1]** this indicates the line with output in it
- If you type an **INCOMPLETE COMMAND**, **R** will wait for you to complete it with the prompt **+**
- **DEMO CODE**

```
> 1 +
+
```

- The **PROMPT CHANGES TO + WHEN R EXPECTS MORE INPUT**
- You can either complete the line, or use **Esc (Ctrl-C)** to exit

```
> 1 +
+ 6
[1] 7
> 1 +
+
```

```
>
```

- R obeys the usual **PRECEDENCE OPERATIONS** (`(, **/^, /, *, +, -`)
- **DEMO CODE**
 - **NOTE SPACES AROUND OPERATORS**

```
> 3 + 5 * 2
[1] 13
> (3 + 5) * 2
[1] 16
> 3 + 5 * 2 ^ 2
[1] 23
> 3 + 5 * (2 ^ 2)
[1] 23
```

- **ARROW KEYS** recover old commands
- The **HISTORY TAB** shows all commands used
- R will report in **SCIENTIFIC NOTATION**
 - **CHECK THAT EVERYONE KNOWS WHAT SCIENTIFIC NOTATION IS**



Red sticky for a question or issue



Green sticky if complete

```
> 2 / 1000
[1] 0.002
> 2 / 10000
[1] 2e-04
> 5e3
[1] 5000
```

BUILT-IN FUNCTIONS

- R has many **STANDARD MATHEMATICAL FUNCTIONS**
- **FUNCTION SYNTAX**
 - type the function name
 - open parentheses
 - type input value
 - close parentheses
 - press return
- **DEMO CODE** - ask for example functions

```
> sin(1)
[1] 0.841471
> log(1)
[1] 0
> log10(10)
[1] 1
> log(10)
[1] 2.302585
```

GETTING HELP FOR BUILT-IN FUNCTIONS

- How do we learn more about a function, or the difference between `log()` and `log10()`?
 - **USE R BUILT-IN HELP**
- **DEMO CODE**

```
> ?log
> help(sin)
```

- This brings up help in the **HELP WINDOW**
 - Scroll to the bottom of the page to find **EXAMPLE CODE**
- You can also use the **SEARCH BOX** at the top of the help window (try `reduce`)

```
> ??log
> args(log)
function (x, base = exp(1))
NULL
> args(log10)
function (x)
NULL
```

- If you're not sure about spelling, the editor has **AUTOCOMPLETION** which will suggest all possible endings for something you type (try `chartr`)
- **USE TAB TO SEE AUTOCOMPLETIONS FOR VARIABLES**

```
> myvar = 10
> myv[TAB]
```

NUMERICAL COMPARISONS

- We can do **COMPARISONS** in R
 - Comparisons return `TRUE` or `FALSE`.
- **DEMO CODE**

```

> 1 == 1
[1] TRUE
> 1 != 2
[1] TRUE
> 1 < 2
[1] TRUE
> 1 <= 1
[1] TRUE
> 1 > 0
[1] TRUE
> 1 >= -9
[1] TRUE

```

- **NOTE:** when comparing numbers, it's better to use `all.equal()` (*machine numeric tolerance*) **ASK IF THERE'S ANYONE FROM MATHS/PHYSICS/COMPUTER SCIENCE**

```

> pi - 1e-8 == pi
[1] FALSE
> all.equal(pi, pi - 1e-8)
[1] TRUE
> all.equal(1.0, 1.0)
[1] TRUE
> all.equal(1.0, 1.1)
[1] "Mean relative difference: 0.1"
> ?all.equal
> all.equal(pi, pi - 1e-8)
[1] TRUE
> all.equal(pi, pi - 1e-8, 1e-16)
[1] "Mean relative difference: 3.183099e-09"
> all.equal(pi, pi - 1e-32)
[1] TRUE
> all.equal(pi, pi - 1e-32, 1e-16)
[1] TRUE
# The precision is set as the square root calculation below - this may
differ from machine to machine
> .Machine$double.eps
[1] 2.220446e-16
> sqrt(.Machine$double.eps)
[1] 1.490116e-08

```

- **THE ORDER/CONSTRUCTION OF MATHEMATICAL OPERATIONS CAN MATTER**
 - Write somewhere if possible: $a = \log(0.01^{\{200\}})$, $b = 200 \times \log(0.01)$
 - These two mathematical expressions are exactly equal: $a = b$
 - But computers are not mathematicians, they're machines. Numbers are susceptible to this *rounding error*, so what happens is this:


```
> log(0.01 ^ 200)
[1] -Inf
> 200 * log(0.01)
[1] -921.034
```

- **COMPUTERS DO WHAT YOU TELL THEM, NOT NECESSARILY WHAT YOU WANT**
-

WORKING IN RSTUDIO

- **RStudio** offers **SEVERAL WAYS TO WRITE CODE**
 - We'll not see all of them today
 - You've seen **DIRECT INTERACTION IN THE CONSOLE** (entering variables)
 - **RStudio** also has an editor for writing scripts, notebooks, markdown documents, and Shiny applications (**EXPLAIN BRIEFLY**)
 - It can also be used to write plain text
- **INTERACTIVE DEMO OF R SCRIPT**
- Click on **File -> New File -> Text File**. **NOTE THAT THE EDITOR WINDOW OPENS**
- Enter the following text, and **EXPLAIN CSV**
 - plain text file
 - one row per line
 - column entries separated by commas
 - first row is header data
 - **NEEDS A BLANK LINE AT THE END**
 - **DATA DESCRIBES CATS**
 - Note that the tab is currently **Untitled1**

```
coat,weight,likes_string
calico,2.1,1
black,5.0,0
tabby,3.2,1
```

- **SAVE THE FILE AS `feline_data.csv`**
 - Click on disk icon
 - Enter filename `feline_data.csv`
 - Note that the name in the tab has changed
- **CLOSE THE EDITOR FOR THAT FILE**
- Click on **File -> New File -> R Script**.
- **EXPLAIN COMMENTS** while entering the code below

- **COMMENTS ANNOTATE YOUR CODE:** reminders for you, and information for others
- Comments should **EXPLAIN THE WHY, NOT THE HOW** - the code should be clear enough to explain *how* at task is performed

```
# Script for exploring RStudio

# Load cat data
cats <- read.csv(file = "feline_data.csv")
```

- **EXPLAIN `read.csv()`**
 - `read.csv()` is a **FUNCTION** that reads data from a **CSV-FORMAT FILE** into a variable in R
- **SAVE THE SCRIPT**
 - Click on **File** -> **Save**
 - Enter filename `cats` (**EXTENSION IS AUTOMATICALLY APPLIED**)
 - Note the tab name has changed to `cats.R`
- **SHOW THE ENVIRONMENT TAB**
 - This describes all variables in the current R environment.
- **ASK: DO YOU SEE THE VARIABLE IN THE ENVIRONMENT?**
 - **NO** - because the code hasn't been executed, only written.
- **RUN THE SCRIPT**
 - Click on **Source**
 - **NOTE THIS RUNS THE WHOLE SCRIPT**
 - **NOTE THE CONSOLE ENTRY**
- Go to the **Environment** tab
 - **NOTE THE DATA WAS LOADED IN THE VARIABLE `cats`**
 - Note that there is a description of the data (3 obs. [rows] of 3 variables [columns])
 - **CLICK ON THE VARIABLE AND NOTE THAT THE TABLE IS NOW VISIBLE** - this is helpful
 - **YOU CANNOT EDIT THE DATA IN THIS TABLE** - you can sort and filter, but not modify the data.
 - This **ENFORCES GOOD PRACTICE: DATA SEPARATION** (compare to Excel).



Red sticky for a question or issue



Green sticky if complete

SECTION 02: MY FIRST RSTUDIO PROJECT

LEARNING OBJECTIVES

- Good practice for RStudio project structure
 - Load data into an RStudio project
 - Produce summary statistics of data
 - Extract subsets of data
 - Plotting data in R
-

PROJECT MANAGEMENT IN RSTUDIO

- RStudio **TRIES TO BE HELPFUL** and provides the 'Project' concept
 - Keeps **ALL PROJECT FILES IN A SINGLE DIRECTORY**
 - **INTEGRATES WITH GIT**
 - Enables switching between projects within RStudio
 - Keeps project histories
- **INTERACTIVE DEMO**
- **CREATE PROJECT**
- Click **File -> New Project**
 - Options for how we want to create a project: -brand new in a new working directory
 - turn an existing directory into a project (project gets directory name)
 - or checkout a project from **GitHub** or some other repository
- Click **New Directory**
 - Options for various things we can do in RStudio. Here we want **New Project**
- Click **New Project**
 - We are asked for a directory name. **ENTER swc-r-lesson**
 - We are asked for a parent directory. **PUT YOURS ON THE DESKTOP; STUDENTS CAN CHOOSE ANYWHERE SENSIBLE**
 - **CHOOSE TO CREATE A GIT REPOSITORY**
 - This might not be available to everyone, depending on setup, so **PAUSE HERE**



Red sticky for a question or issue



Green sticky if complete

- Click **Create Project**
- **YOU SHOULD SEE AN EMPTY-ISH RSTUDIO WINDOW**

- **INSPECT PROJECT ENVIRONMENT**
- First, **NOTE THE WINDOWS**: console; environment; files
- **CONSOLE** is empty
- **ENVIRONMENT** is empty
- **FILES** shows
 - **CURRENT WORKING DIRECTORY** (see breadcrumb trail) **IS ROOT FOR PROJECT**
 - **THREE FILES**:
 - ***.Rproj** - information about your project
 - **.Rhistory** - records actions taken on the project
 - **.gitignore** - if you created a git repository, this contains paths/names of files to be ignored
- **CREATE DIRECTORIES IN PROJECT**
- Create directories called **scripts** and **data**
 - Click on **New Folder**
 - Enter directory name (**scripts**)
 - Note that the directory now exists in the **Files** tab
 - Do the same for **data/**
- **NOTE THAT WE WILL POPULATE THE DIRECTORIES AS WE GO**
- **LOOK AT THE GIT INTEGRATION**
- There is a **.gitignore** file indicating that the project is under **git** version control
- There is a **NEW TAB** called **Git** in the Environment pane
 - **CLICK ON GIT TAB**
 - There are two files, **NOTHING IS STAGED YET**
 - **STAGE THE FILES** by clicking on the checkboxes
 - It is **GOOD PRACTICE** to place the project **.Rproj** file under version control
 - **NOTE STATUS CHANGES FROM ? TO A** (added)
 - **COMMIT THE FILES** by clicking **Commit**
 - **NOTE THE NEW WINDOW**
 - Show the **diff** for both files: green means added/new line
 - **ADD A COMMIT MESSAGE** - remind learners of good practice
 - good **Git** commit messages are imperative and short
 - **CLICK COMMIT**
 - Close the message box down
- **NOTE THAT THIS IS JUST LIKE WORKING WITH GIT AT THE COMMAND LINE**
- **OPEN THE TERMINAL TAB**
 - **NOTE WE ARE IN THE WORKING DIRECTORY**

◦ RUN COMMANDS

```
$ git status
On branch master
nothing to commit, working tree clean
$ git ls-files
.gitignore
swc-r-lesson.Rproj
$ ls
data/                scripts/             swc-r-lesson.Rproj
```

OBTAINING DATA

- We've already created some cat data manually
 - **THIS IS UNUSUAL** - most data comes in the form of plain text files
- **START DEMO**
- **GO TO ETHERPAD**
 - **DOWNLOAD DATA** - right-click on link and save to project's **data/** subdirectory
 - **PUT DATA UNDER VERSION CONTROL** - this is **GOOD PRACTICE**
 - For reproducibility, keep raw data with the analysis as much as is reasonable
 - Discuss when it might not be reasonable
 - **NOTE CHANGES IN DATA/ SUBDIRECTORY**
 - the directory shows up in the **Git** tab
 - **STAGE DATA/**
 - note that the filename is now shown
 - **COMMIT THE DATAFILE**

INVESTIGATING GAPMINDER

- **INSPECT DATA IN FILES WINDOW**
 - Click on filename, and select **View File**
 - Note: **THERE ARE NO ROW NAMES**
 - Ask: **IS THIS WELL-FORMATTED DATA?** (can you tell what the data represents?)
- **WHAT IS THE DATA TYPE**
 - Tabular, with **EACH COLUMN SEPARATED BY A COMMA**, so **CSV**
 - **IN THE CONSOLE** use `read.table()` to read the data in

```
gapminder <- read.table("data/gapminder-FiveYearData.csv", sep=",",
header=TRUE)
```

- Note: **IF WE DON'T ASSIGN THE RESULT TO A VARIABLE WE JUST SEE THE DATA**
- Now we've loaded our data, let's take a look at it
- **DEMO IN CONSOLE**
 - 1704 rows, 6 columns
 - Investigate types of columns
 - **POINT OUT THAT THE TYPE OF A COLUMN IS INTEGER IF IT'S A FACTOR**
 - **LENGTH OF A DATAFRAME IS THE NUMBER OF COLUMNS**

```

> 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 ...
> typeof(gapminder$year)
[1] "integer"
> typeof(gapminder$country)
[1] "integer"
> str(gapminder$country)
Factor w/ 142 levels "Afghanistan",...: 1 1 1 1 1 1 1 1 1 1 ...
> levels(gapminder$country)
 [1] "Afghanistan"          "Albania"              "Algeria"
"Angola"
 [5] "Argentina"           "Australia"            "Austria"
"Bahrain"
 [...]
```

```

> length(gapminder)
[1] 6
> nrow(gapminder)
[1] 1704
> ncol(gapminder)
[1] 6
> dim(gapminder)
[1] 1704  6
> colnames(gapminder)
[1] "country" "year"    "pop"     "continent" "lifeExp"
"gdPercap"
> 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

```

```

> summary(gapminder)
      country      year      pop      continent
lifeExp
Afghanistan: 12  Min.   :1952  Min.   :6.001e+04  Africa :624  Min.
:23.60
Albania    : 12  1st Qu.:1966  1st Qu.:2.794e+06  Americas:300  1st
Qu.:48.20
Algeria    : 12  Median :1980  Median :7.024e+06  Asia    :396
Median :60.71
Angola     : 12  Mean   :1980  Mean   :2.960e+07  Europe  :360  Mean
:59.47
Argentina  : 12  3rd Qu.:1993  3rd Qu.:1.959e+07  Oceania : 24  3rd
Qu.:70.85
Australia  : 12  Max.   :2007  Max.   :1.319e+09      Max.
:82.60
(Other)    :1632
gdpPercap
Min.      :   241.2
1st Qu.   :  1202.1
Median    :  3531.8
Mean      :  7215.3
3rd Qu.   :  9325.5
Max.      :113523.1

```

SECTION 03: PROGRAM FLOW CONTROL

LEARNING OBJECTIVES

- In this short section, you'll learn how to **perform actions depending on values of data** in R
- You'll also learn how to **repeat operations, using `for()` loops**
- **These are very important general concepts, that recur in many programming languages**
- Much of the time, you can avoid using them in R data analyses, because `dplyr` exists, and because R is **vectorised**

IF() ... ELSE**

- We **often want to run a piece of code, or take an action, dependent on whether some data has a particular value
 - (if it is true or false, say)
- When this is the case, we can use the general `if() ... else` structure, which is common to most programming languages
- **DEMO IN SCRIPT**
- **CREATE NEW SCRIPT** (save as `scripts/flow_control.R`)

- Let's say that we want to print a message if some value is greater than 10
- **NOTE AUTOCOMPLETION/BRACKETS ETC.**
- **THE CODE TO BE RUN GOES IN CURLY BRACES**

```
# A script to investigate flow control in R

# Synthetic data
x <- 8

# Example conditional
if (x > 10) {
  print("x is greater than 10")
}
```

- **SOURCE THE FILE**

- **NOTHING HAPPENS** ($x > 10$ is **FALSE**)
- The `if()` block executes **ONLY IF THE VALUE IN PARENTHESES EVALUATES AS TRUE**

- **MODIFY THE SCRIPT**

- Add the `else` block
- **Source** the code: **WE GET A MESSAGE**
- **BUT IS THE MESSAGE TRUE?**

```
# Example if statement
if (x > 10) {
  print("x is greater than 10")
} else {
  print("x is less than 10")
}
```

- **SET `x <- 10` AND TRY AGAIN**

- Is the answer correct?



Red sticky for a question or issue



Green sticky if complete

- **MODIFY THE SCRIPT WITH `else if()` STATEMENT**

- **Source** the script: **NO OUTPUT**

```
# A data point
x <- 10
```



```
# Example if statement
if (x > 10) {
  print("x is greater than 10")
} else if (x < 10) {
  print("x is less than 10")
}
```

- **MODIFY THE SCRIPT WITH A FINAL `else` STATEMENT**

- Source the script: **EQUALS** output

```
# A data point
x <- 9

# Example if statement
if (x > 10) {
  print("x is greater than 10")
} else if (x < 10) {
  print("x is less than 10")
} else {
  print("x is equal to 10")
}
```

- **TRY SOME OTHER VALUES for `x`**

SLIDE: Challenge

- Build up the solution with each concept in turn

```
> gapminder$year
 [1] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007 1952
1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007
 [25] 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007 1952
1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007
 [37] ...
> gapminder$year == 2002
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE
FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
 [21] FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
FALSE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE
 [37] ...
> any(gapminder$year == 2002)
 [1] TRUE
> any(gapminder$year == 2001)
 [1] FALSE
# Are there any records for a year
year <- 2002
if(any(gapminder$year == year)){
```

```
print("Record(s) for this year found.")
}
```



Red sticky for a question or issue



Green sticky if complete

FOR() LOOPS

- If you want to iterate over a set of values, then `for()` loops can be used
- `for()` loops are **A VERY COMMON PROGRAMMING CONSTRUCTION**
- They express the idea: **FOR EACH ITEM IN A GROUP, DO SOMETHING (WITH THAT ITEM)**
- **DEMO IN SCRIPT** (`scripts/flow_control.R`)
 - Say we have a *vector* `c(1,2,3)`, and we want to print each item
 - We can **loop over all the items** and print them
- **The loop structure is**
 - `for()`, where the argument names a variable (`i`) - the *iterator*, and a set of values: `for(i in c('a', 'b', 'c'))`
 - A **CODE BLOCK** defined by curly braces (**note automated completion)
 - The **contents of the code block are executed for each value of the iterator**

```
# Example for loop
for (i in c('a', 'b', 'c')) {
  print(i)
}
```

- **Loops can (but shouldn't always) be nested**
- **DEMO IN SCRIPT**
 - The outer loop is executed and, **for each value in the outer loop, the inner loop is executed to completion**

```
# Example nested for loop
for (i in 1:5) {
  for (j in c('a', 'b', 'c')) {
    print(paste(i, j))
  }
}
```

- The simplest way to capture output from a loop is to add a new item to a vector each iteration of the loop
- **DEMO IN SCRIPT**
 - **REMIND:** using `c()` to append to a vector

```
# Capturing loop output in a vector
output <- c()
for (i in 1:5) {
  for (j in c('a', 'b', 'c')) {
    output <- c(output, paste(i, j))
  }
}
print(output)
```

- **GROWING OUTPUT FROM LOOPS IS COMPUTATIONALLY VERY EXPENSIVE**
 - Doing this will *really* slow down your scripts for larger datasets
 - Better to define the empty output container first (**IF YOU KNOW THE DIMENSIONS**)
- **MODIFY IN SCRIPT**

```
# Capturing loop output in a matrix
output_matrix <- matrix(nrow=5, ncol=3)
j_letters = c('a', 'b', 'c')
for (i in 1:5) {
  for (j in 1:3) {
    output_matrix[i, j] <- paste(i, j_letters[j])
  }
}
print(output_matrix)
```

WHILE() LOOPS

- Sometimes you need to perform some action **ONLY WHILE A CONDITION IS TRUE**
 - This isn't as common as a `for()` loop
 - It's another **GENERAL PROGRAMMING CONSTRUCTION**
- **DEMO IN SCRIPT**
 - We'll generate random numbers until one falls below a threshold
 - `runif()` generates random numbers from a uniform distribution
 - **ASK LEARNERS HOW TO GET HELP ON THIS**

```
> ?runif
```

- We print random numbers until one is less than 0.1

- **RUN A COUPLE OF TIMES TO SHOW OUTPUT IS RANDOM**

```
# Example while loop
z <- 1
while(z > 0.1){
  z <- runif(1)
  print(z)
}
```

CHALLENGE

- Best to give example of what `letters` is, and demonstrate how the help mechanism fails for `%in%`
- Also show that `%in%` doesn't work for membership of a string - needs a vector

```
> letters
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
"r" "s" "t" "u" "v" "w" "x" "y" "z"
> ?%in%
Error: unexpected SPECIAL in "?%in%"
> ?in
Error: unexpected 'in' in "?in"
> ?"%in%"
> 'e' %in% letters
[1] TRUE
> 'e' %in% 'aeiou'
[1] FALSE
> 'e' %in% c('a', 'e', 'i', 'o', 'u')
[1] TRUE
```

- Then, in the script

```
# Challenge solution
gapminder <- read.table("data/gapminder-FiveYearData.csv", sep=";",
header=TRUE)
for (c in levels(gapminder$country)) {
  if (startsWith(c, 'M')) {
    value <- TRUE
  } else {
    value <- FALSE
  }
  print(paste(c, value))
}
```

- **COMMIT THE SCRIPT TO THE REPO WHEN DONE**



Red sticky for a question or issue



Green sticky if complete

VECTORISATION

- `for()` and `while()` loops are useful, but they are **NOT THE MOST EFFICIENT WAY TO WORK WITH DATA IN R**
- **MOST FUNCTIONS IN R ARE VECTORISED**
 - When applied to a vector, they work on all elements in the vector
 - **NO NEED TO USE A LOOP**
- **DEMO IN CONSOLE**
 - **OPERATORS** are vectorised

```
> x <- 1:4
> x
[1] 1 2 3 4
> x * 2
[1] 2 4 6 8
```

- **YOU CAN OPERATE ON VECTORS TOGETHER**

```
> y <- 6:9
> y
[1] 6 7 8 9
> x + y
[1] 7 9 11 13
> x * y
[1] 6 14 24 36
```

- **COMPARISON OPERATORS ARE VECTORISED**

```
> x > 2
[1] FALSE FALSE TRUE TRUE
> y < 7
[1] TRUE FALSE FALSE FALSE
> any(y < 7)
[1] TRUE
> all(y < 7)
[1] FALSE
```

- **MANY FUNCTIONS WORK ON VECTORS**

```
> log(x)
[1] 0.0000000 0.6931472 1.0986123 1.3862944
> x^2
[1] 1 4 9 16
> sin(x)
[1] 0.8414710 0.9092974 0.1411200 -0.7568025
```

CHALLENGE

```
# Challenge solution
countries <- levels(gapminder$country)
mstart <- startsWith(countries, 'M')
print(countries[mstart])
```

- **COMMIT MODIFIED SCRIPT**



Red sticky for a question or issue



Green sticky if complete

SECTION 04: FUNCTIONS

LEARNING OBJECTIVES

- **YOU'VE ALREADY BEEN USING BUILT-IN FUNCTIONS (e.g. `log()`)** and, I hope, have found them useful
- Functions let us run a complex series of commands in one go
 - **YOU WOULDN'T HAVE TO WANT TO WRITE/REPEAT BASIC CALCULATIONS FOR `log()` EACH TIME YOU USE IT**
 - They keep the operation under a **MEMORABLE OR DESCRIPTIVE NAME**, which makes the code **READABLE AND UNDERSTANDABLE**, and they are invoked with that name
 - There are a **DEFINED SET OF INPUTS AND OUTPUTS** for a function, so **WE KNOW WHAT BEHAVIOUR TO EXPECT**

WHY FUNCTIONS?

- Functions let us **RUN A COMPLEX SERIES OF RELATED COMMANDS IN ONE GO**
 - Can be **LOGICALLY** or **FUNCTIONALLY** related

- It helps when functions have **DESCRIPTIVE AND MEMORABLE NAMES**, as this makes code **READABLE AND UNDERSTANDABLE**
 - We invoke functions with their name
 - We **A DEFINED SET OF INPUTS AND OUTPUTS** - aids clarity and understanding
 - **FUNCTIONS ARE THE BUILDING BLOCKS OF PROGRAMMING**
 - As a **RULE OF THUMB** it is good to write small functions with one obvious, clearly-defined task.
 - As you will see we can **CHAIN SMALL FUNCTIONS TOGETHER TO MANAGE COMPLEXITY**
-

DEFINING A FUNCTION

- Functions have a **STANDARD FORM** in R
 - We **DECLARE A <function_name>**
 - We use the `function` keyword to assign the function to `<function_name>`
 - Inputs (*arguments*) to a function are defined in parentheses: These are defined as variables for use within the function **AND DO NOT EXIST OUTSIDE THE FUNCTION**
 - The code block (**cCURLY BRACES**) encloses the function code, the *function body*.
 - **NOTE THE INDENTATION** - *Easier to read, but does not affect execution*
 - The code `<does_something>`
 - The `return()` function returns the value, when the function is called
- **DEMO IN SCRIPT**
 - **CREATE NEW SCRIPT** `functions.R`
 - Write and `Source`

```
# Example function
# Returns the sum of two input values
my_sum <- function(a, b) {
  the_sum <- a + b
  return(the_sum)
}
```

- **ADD SCRIPT TO VERSION CONTROL**
- **DEMO IN CONSOLE**
 - **SOURCE** the script

```
> my_sum(3, 7)
[1] 10
> a
```

```
Error: object 'a' not found
> b
Error: object 'b' not found
```

- **GOOD VARIABLE NAMING IS IMPORTANT**

- For a function this size, and so simple, it's clear what **a** and **b** are - but that is not always the case
- We can make the function clearer by changing these names

- **CHANGE VARIABLE NAMES IN-PLACE**

- **TEST THE SCRIPT**

```
# Example function
# Returns the sum of two input values
my_sum <- function(val1, val2) {
  the_sum <- val1 + val2
  return(the_sum)
}
```

```
> source('~/Desktop/swc-r-lesson/scripts/functions.R')
> my_sum(3, 7)
[1] 10
> a
Error: object 'a' not found
> val1
Error: object 'val1' not found
> val2
Error: object 'val2' not found
```

- **ADD SCRIPT TO VERSION CONTROL**

- **DEMO IN SCRIPT**

- Let's define another function: convert temperature from fahrenheit to Kelvin

```
# Convert Fahrenheit to Kelvin
fahr_to_kelvin <- function(temp) {
  kelvin <- (temp - 32) * (5 / 9) + 273.15
  return(kelvin)
}
```

- **SOURCE AND DEMO IN SCRIPT**


```
> fahr_to_kelvin(32)
[1] 273.15
> fahr_to_kelvin(-40)
[1] 233.15
> fahr_to_kelvin(212)
[1] 373.15
> temp
Error: object 'temp' not found
```

- **LET'S MAKE ANOTHER FUNCTION CONVERTING KELVIN TO CELSIUS**
- **DEMO IN SCRIPT**
 - [Source](#) the script

```
# Convert Kelvin to Celsius
kelvin_to_celsius <- function(temp) {
  celsius <- temp - 273.15
  return(celsius)
}
```

- **SOURCE AND DEMO IN CONSOLE**

```
> kelvin_to_celsius(273.15)
[1] 0
> kelvin_to_celsius(233.15)
[1] -40
> kelvin_to_celsius(373.15)
[1] 100
```

- **WE COULD DEFINE A NEW FUNCTION TO CONVERT FAHRENHEIT TO CELSIUS**
 - But it's **EASIER TO COMBINE EXISTING FUNCTIONS** we've already written
 - **AVOIDS INTRODUCING NEW BUGS**
 - Efficient to **REUSE CODE**
- **DEMO IN CONSOLE**

```
> fahr_to_kelvin(212)
[1] 373.15
> kelvin_to_celsius(fahr_to_kelvin(212))
[1] 100
```

- **DEMO IN SCRIPT**

```
# Fahrenheit to Celsius
fahr_to_celsius <- function(temp) {
  celsius <- kelvin_to_celsius(fahr_to_kelvin(temp))
  return(celsius)
}
```

- **DEMO IN CONSOLE**

- **NOTE: AUTOMATICALLY TAKES ADVANTAGE OF R's VECTORISATION**

```
> fahr_to_celsius(212)
[1] 100
> fahr_to_celsius(32)
[1] 0
> fahr_to_celsius(-40)
[1] -40
> fahr_to_celsius(c(-40, 32, 212))
[1] -40  0 100
```

DOCUMENTING FUNCTIONS

- It's important to have well-named functions (and variables... **THIS IS A FORM OF DOCUMENTATION - SELF-DOCUMENTING CODE**)
- But it's not a detailed explanation
- You've found R's help useful, but it doesn't exist for your functions until you write it
- **YOUR FUTURE SELF WILL THANK YOU FOR DOING IT!**
- **SOME GOOD PRINCIPLES TO FOLLOW WHEN WRITING DOCUMENTATION ARE:**
 - Say **WHAT** the code does (and **WHY**) - *more important than how* (the code does that)
 - Define your inputs and outputs
 - Provide an example

- **DEMO IN CONSOLE**

```
> ?fahr_to_celsius
No documentation for 'fahr_to_celsius' in specified packages and
libraries:
you could try '??fahr_to_celsius'
> ??fahr_to_celsius
```

- **DEMO IN SCRIPT**

- We add documentation as comment strings in the function

```
# Fahrenheit to Celsius
fahr_to_celsius <- function(temp) {
  # Convert input temperature from fahrenheit to celsius scale
  #
  # temp      - numeric
  #
  # Example:
  # > fahr_to_celsius(c(-40, 32, 212))
  # [1] -40  0 100
  celsius <- kelvin_to_celsius(fahr_to_kelvin(temp))
  return(celsius)
}
```

- **SOURCE** the script
- **DEMO IN CONSOLE**

```
> ?fahr_to_celsius
No documentation for 'fahr_to_celsius' in specified packages and
libraries:
you could try '??fahr_to_celsius'
```

- We read the documentation by providing the function name **ONLY**

```
> fahr_to_celsius
function(temp) {
  # Convert input temperature from fahrenheit to celsius scale
  #
  # temp      - numeric
  #
  # Example:
  # > fahr_to_celsius(c(-40, 32, 212))
  # [1] -40  0 100
  celsius <- kelvin_to_celsius(fahr_to_kelvin(temp))
  return(celsius)
}
```

- **COMMIT SCRIPT TO VERSION CONTROL**

FUNCTION ARGUMENTS

- **DEMO IN SCRIPT** (`functions.R`)
 - `Source` script

```
# Report countries in gapminder data
list_countries <- function(data) {
```

```
# Returns countries from the gapminder dataset
#
# data          - gapminder data.frame
#
# Example:
# countries <- list_countries(gapminder)
countries <- levels(data$country)
return(countries)
}
```

- **SOURCE SCRIPT**
- **DEMO IN CONSOLE**

```
> list_countries(gapminder)
[1] "Afghanistan"      "Albania"          "Algeria"
[4] "Angola"           "Argentina"        "Australia"
[7] "Austria"          "Bahrain"          "Bangladesh"
[...]
```

- So, those are *all* the `gapminder` countries - but what if we want to **GET COUNTRIES STARTING WITH A GIVEN LETTER?**
- **DEMO IN SCRIPT** (`functions.R`)
 - `Source` script

```
> list_countries(gapminder, 'M')
Error in list_countries(gapminder, "M") : unused argument ("M")
```

- The function doesn't understand what we want
- We need to **TELL THE FUNCTION TO EXPECT A LETTER**
 - **DEMO IN SCRIPT**
 - Don't forget to update the documentation

```
# Report countries in gapminder data
list_countries <- function(data, letter) {
  # Returns countries from the gapminder dataset,
  # filtered by starting letter
  #
  # data          - gapminder data.frame
  # letter        - character
  #
  # Example:
  # countries <- list_countries(gapminder)
countries <- levels(data$country)
matches <- startsWith(countries, letter)
return(countries[matches])
}
```

- Now the function should accept a letter, and report only countries starting with the letter
- **SOURCE THE SCRIPT**

```
> list_countries(gapminder, 'M')
[1] "Madagascar" "Malawi"      "Malaysia"    "Mali"        "Mauritania"
"Mauritius"    "Mexico"
[8] "Mongolia"    "Montenegro" "Morocco"     "Mozambique"  "Myanmar"
```

- So that works, but we have a problem:

```
> list_countries(gapminder)
Error in startsWith(countries, letter) :
  argument "letter" is missing, with no default
```

- **NO LETTER PROVIDED MEANS NO OUTPUT**
 - We need to handle this
 - 1 - **PROVIDE A DEFAULT VALUE (NULL)**
 - 2 - **TEST FOR VALUE AND TAKE ALTERNATIVE ACTIONS**
- **DEMO IN SCRIPT**

```
# Report countries in gapminder data
list_countries <- function(data, letter=NULL) {
  # Returns countries from the gapminder dataset, optionally
  # filtered by starting letter
  #
  # data          - gapminder data.frame
  # letter        - character
  #
  # Example:
  # countries <- list_countries(gapminder)
  countries <- levels(data$country)
  if (!is.null(letter)) {
    matches <- startsWith(countries, letter)
    countries <- countries[matches]
  }
  return(countries)
}
```

- **SOURCE SCRIPT**
- **DEMO IN CONSOLE**

```
> source('~/Desktop/swc-r-lesson/scripts/functions.R')
> list_countries(gapminder)
[1] "Afghanistan"      "Albania"          "Algeria"
```

```

[4] "Angola"           "Argentina"           "Australia"
[7] "Austria"          "Bahrain"             "Bangladesh"
[... ]
> list_countries(gapminder, 'M')
[1] "Madagascar" "Malawi"           "Malaysia"  "Mali"       "Mauritania"
"Mauritius"  "Mexico"
[8] "Mongolia"  "Montenegro" "Morocco"   "Mozambique" "Myanmar"
> list_countries(gapminder, 'G')
[1] "Gabon"           "Gambia"           "Germany"   "Ghana"
"Greece"          "Guatemala"
[7] "Guinea"         "Guinea-Bissau"

```

- COMMIT SCRIPT TO VERSION CONTROL

SECTION 05: DYNAMIC REPORTS

LEARNING OBJECTIVES

- In this section, we'll be learning how to create **REPRODUCIBLE, ATTRACTIVE, DYNAMIC REPORTS** with **RMarkdown**
- To do so, we'll learn some **RMarkdown SYNTAX**, and how to put **WORKING R CODE** into a document
- We'll also look at generating the report in **A NUMBER OF FILE FORMATS**, for sharing.

LITERATE PROGRAMMING

- What we're about to do is an example of **Literate Programming**, a concept introduced by Donald Knuth
- The idea of Literate Programming is that
 - The program or analysis is explained in **NATURAL LANGUAGE****
 - The **CODE** needed to run the program/analysis is **EMBEDDED IN THE DOCUMENT**
 - The whole document is executable
- We can produce these documents in **RStudio**

CREATE AN RMarkdown FILE

- In **R**, literate programming is ****implemented in RMarkdown files**
- To create one: **File \$→ New File \$→ R Markdown**
 - There is a dialog box
 - **ENTER A TITLE** (*Literate Programming*)
 - **CLICK OK**
 - Save the file (**Ctrl-S**)
 - **CREATE NEW SUBDIRECTORY** (*markdown*)**
 - **SAVE AS** *literate_programming.Rmd*

- The file gets the **EXTENSION** `.Rmd`
 - The file is **AUTOPOPULATED** with example text

COMPONENTS OF AN RMarkdown FILE

- The **HEADER REGION** is fenced by `---`
 - **METADATA** (author, title, date)
 - Requested **OUTPUT FORMAT**

```
---
title: "Literate Programming"
author: "Leighton Pritchard"
date: "04/12/2017"
output: html_document
---
```

- Natural language is written as plain text, with some **EXTRA CHARACTERS FOR FORMATTING**
 - **NOTE THE HASHES #, ASTERISKS * AND ANGLED BRACKETS <>**
- R code runs in the document, and is **fenced by backticks**
- **CLICK ON KNIT**
 - A new (pretty) document is produced in a new window
- **CROSS REFERENCE MARKDOWN TO DOCUMENT**
 - **Title, Author, Date**
 - **Header**
 - **Link**
 - **Bold**
 - **R code and output**
 - **Plots**
- **SHOW THAT AN HTML FILE IS PRODUCED**
- **CLICK ON KNIT TO PDF**
 - A new `.pdf` document opens in a new window
- **CROSS REFERENCE MARKDOWN TO DOCUMENT**
 - **NOTE:** The formatting isn't identical
- **CLICK ON KNIT TO WORD**
 - A new `Word` document opens up

- **CROSS REFERENCE MARKDOWN TO DOCUMENT**
 - **NOTE:** The formatting isn't identical
 - **NOTE THE LOCATION OF THE OUTPUT FILES - ALL IN THE SOURCE DIRECTORY**
 - **CLOSE THE OUTPUT**
-

CREATING A REPORT

- We'll **CREATE A REPORT** on the `gapminder` data
 - We'll be using **LITERATE PROGRAMMING**
 - We'll also be learning some `dplyr` and `ggplot2` as we go
-

SECTION 06: `dplyr`

LEARNING OBJECTIVES

- You're going to **learn to manipulate `data.frames` with the six verbs of `dplyr`**
 - `select()`
 - `filter()`
 - `group_by()`
 - `summarize()`
 - `mutate()`
 - `%>%` (pipe)
-

WHAT AND WHY IS `dplyr`?

- `dplyr` is a package in the **TIDYVERSE**; it exists to enable **RAPID ANALYSIS OF DATA BY GROUPS**
 - For example, if we wanted numerical (rather than graphical) analysis of the `gapminder` data by continent, we'd use `dplyr`
- So far, we know how to subset, but **REPETITIVE APPLICATION IS TEDIOUS****
- **DEMO IN CONSOLE**
 - If we wanted to use the `gapminder` data to calculate average GDP per continent, we can
 - This needs repetition each time we calculate for a continent


```
> mean(gapminder[gapminder$continent == "Africa", "gdpPercap"])
[1] 2193.755
> mean(gapminder[gapminder$continent == "Asia", "gdpPercap"])
[1] 7902.15
> mean(gapminder[gapminder$continent == "Americas", "gdpPercap"])
[1] 7136.11
```

- **WE MIGHT MANAGE TO REPEAT BY CONTINENT, LIKE HERE - BUT *BY COUNTRY*?**
- **AVOIDING REPETITION IMPROVES YOUR CODE**
 - More **ROBUST**
 - More **READABLE**
 - More **REPRODUCIBLE**

SPLIT-APPLY-COMBINE

- The general principle `dplyr` supports is **SPLIT-APPLY-COMBINE**
- We have a **DATASET WITH SEVERAL GROUPS** (column `x`)
- We want to **PERFORM THE SAME OPERATION ON EACH GROUP, INDEPENDENTLY** - take a mean of `y` for each group defined in `x`, for example
 - So we **SPLIT** the data into groups, on `x`
 - Then we **APPLY** the operation (take the mean for each group)
 - Then we **COMBINE** the results into a new table

`select()`

- **CREATE NEW RMarkdown FILE**
 - Title it `dplyr`
 - Save in `markdown`
- **DELETE DEFAULT TEXT**
- **WRITE INTRODUCTION**

```
# Introduction
```

```
This `RMarkdown` document contains examples for learning how to use
`dplyr` to analyse dataframes by groups
```

```
## Importing `dplyr`
```

```
Import `dplyr` with the `require()` function, in the `{r setup}` block
```

```
We also import `knitr` to get pretty tables and formatting.
```

```
## Load data

```{r load_data}
gapminder <- read.table("../data/gapminder-FiveYearData.csv", sep="," ,
header=TRUE)
kable(head(gapminder))
```
```

- Describe **FORMAT SYMBOLS**
- Note `kable()` **PRETTIFIES TABLES**
- Add `require(dplyr)` and `require(knitr)` to `{r setup}` block
- Remember the **RELATIVE PATHS**
- **KNIT THE DOCUMENT**
 - It's not particularly fancy, but it works.



Red sticky for a question or issue



Green sticky if complete

- **WRITE TEXT FOR `select()`**

```
# `dplyr` verbs

## `select()`

The `select()` *verb* selects *columns* by name.

If we want to select year, country and GDP data from `gapminder`, we
specify the dataset, then the column names with the `select()` function:

```{r select}
kable(head(select(gapminder, year, country, gdpPercap)))
```
```

- **KNIT THE DOCUMENT**
 - The `R` command and its output are both displayed.
- Add text about using the **PIPE `%>%`**
- Here, we applied a function, but we can also **'PIPE' DATA FROM ONE VERB TO ANOTHER**
 - These work **LIKE PIPE IN THE SHELL****
 - **SPECIAL PIPE SYMBOL: `%>%`**
 - Specify **ONLY COLUMNS**

As well as using the `select()` function, we can *pipe* data into *verbs* using the `%>%` (*pipe*) operator. We *pipe* in the data, and only need to specify chosen columns:

```
```{r select_pipe}
kable(head(gapminder %>% select(year, country, gdpPerCap)))
```
```

- **KNIT THE DOCUMENT**

- The **R** command and its output are both displayed.



Red sticky for a question or issue



Green sticky if complete

filter()

- `filter()` selects rows on the basis of a specified condition, or combination of conditions
 - We can **USE IT AS A FUNCTION, OR WITH PIPES**
- **WRITE TEXT FOR `filter()`**

```
## `filter()`
```

The `filter()` *verb* selects *rows* on the basis of a specified condition, or combination of conditions. It can also be used as a function or with *pipes*:

```
```{r filter}
kable(head(filter(gapminder, continent=="Europe")))
```

```{r filter_pipe}
kable(head(gapminder %>% filter(continent=="Europe")))
```
```

- **KNIT THE DOCUMENT**

- The **R** command and its output are both displayed.



Red sticky for a question or issue



Green sticky if complete

- **WRITE TEXT FOR `%>%` (PIPES)**

```
## `>` (pipe)
```

The `>` (pipe) operator makes **chaining verbs** together (into a **pipeline**) more readable. When several lines of code are needed, end lines with a pipe so that `R` knows to continue.

```
```{r pipes}
eurodata <- gapminder %>%
 filter(continent == "Europe") %>%
 select(year, country, gdpPercap)
kable(head(eurodata))
```
```

- **KNIT THE DOCUMENT**

- The **R** command and its output are both displayed.



Red sticky for a question or issue



Green sticky if complete

CHALLENGE

- **WRITE CHALLENGE TEXT IN DOCUMENT**

- Solve the challenge in your **RMarkdown** document

```
### Challenge
```

Write a single command (which may span multiple lines, with pipes) to produce a dataframe containing:

- country
- life expectancy
- year data

only for African nations.

Call the dataframe `afrodata`. How many rows does it have?

```
```{r challenge_1}
afrodata <- gapminder %>%
 filter(continent == "Africa") %>%
 select(country, year, lifeExp)
kable(head(afrodata))
```
```

The dataframe `afrodata` has `r nrow(afrodata)` rows.

- **EXPLAIN INLINE R SYNTAX**
 - This means we don't have to run analysis and type output
 - We can use **R** variables directly in the text
- **KNIT THE DOCUMENT**
 - The **R** command and its output are both displayed.



Red sticky for a question or issue



Green sticky if complete

group_by()

- The `group_by()` *verb* **SPLITS data.frames INTO GROUPS BASED ON A COLUMN PROPERTY**
 - It returns a **tibble** - a table with extra metadata describing the groups in the table
- ****WRITE TEXT FOR `group_by()`**

```
## `group_by()`
```

The `group_by()` **verb** splits `data.frame`'s into groups, based on a column property. It returns a `tibble` - a table with extra metadata describing groups in the table.

```
```{r group_by}
group_by(gapminder, continent)
```
```

This is an extremely useful step in split-apply-combine.

- **KNIT THE DOCUMENT**
 - The **R** command and its output are both displayed.



Red sticky for a question or issue



Green sticky if complete

summarize()

- The **COMBINATION OF `group_by()` AND `summarize()` IS VERY POWERFUL AND THE BASIS OF SPLIT-APPLY-COMBINE**
- **WRITE TEXT FOR `summarize()`**

```
## `summarize()`
```

The combination of `group_by()` and `summarize()` is very powerful, and the basis of `split-apply-combine`.

We can create *new variables* using functions that repeat their action on each group produced with `group_by()`. For instance, we can calculate mean GDP by continent for `gapminder` data:

```
```{r summarize}
meangdpContinent <- gapminder %>%
 group_by(continent) %>%
 summarize(meangdpPerCap = mean(gdpPerCap))
kable(meangdpContinent)
```
```

- **KNIT THE DOCUMENT**

- The **R** command and its output are both displayed.



Red sticky for a question or issue



Green sticky if complete

CHALLENGE

- **WRITE CHALLENGE TEXT IN DOCUMENT**

- Solve the challenge in your **RMarkdown** document

```
### Challenge
```

Write a single command (which may span multiple lines, with pipes) to produce a dataframe describing:

– average life expectancy by country

Call the dataframe `meanlifexpCountry`. Which nation has the greatest and which the shortest life expectancy?

```
```{r challenge_2}
afrodata <- gapminder %>%
 filter(continent == "Africa") %>%
 select(country, year, lifeExp)
kable(head(afrodata))
```
```

The nation with the longest life expectancy is `r meanlifexpCountry %>% filter(meanlifeExp==max(meanlifeExp)) %>% select(country)` and that with

```
the shortest is `r meanlifexpCountry %>%
filter(meanlifeExp==min(meanlifeExp)) %>% select(country)`.
```

- **EXPLAIN INLINE R SYNTAX**

- This means we don't have to run analysis and type output
- We can use R calculations directly in the text

- **KNIT THE DOCUMENT**

- The R command and its output are both displayed.



Red sticky for a question or issue



Green sticky if complete

`count()` and `n()`

- Two other useful functions are related to `summarize()`
 - `count()` reports a **NEW TABLE OF COUNTS BY GROUP**
 - `n()` is used to represent the **COUNT OF ROWS**, when calculating new values in `summarize()`
- **WRITE TEXT FOR `count()` AND `n()`**

```
## `count()` and `n()`
```

The **verbs** `count()` and `n()` are related to `summarize()`.

- `count()` reports a new table of counts, by group
- `n()` represents the count of rows when calculating new values with `summarize()`

We can use `count()` to get the number of results in 2002 in each continent:

```
```{r count}
result <- gapminder %>%
 filter(year == 2002) %>%
 count(continent, sort=TRUE)
kable(result)
```
```

We can use `n()` as a value in a calculation to help calculate standard error of life expectancy for each continent:

```
```{r n}
result <- gapminder %>%
 group_by(continent) %>%
 summarize(selifeExp = sd(lifeExp)/sqrt(n()))
```

```
kable(result)
```

```

- **KNIT THE DOCUMENT**
 - The **R** command and its output are both displayed.
- **NOTE:** standard error is (std dev)/sqrt(n)



Red sticky for a question or issue



Green sticky if complete

mutate()

- **mutate() CALCULATES NEW VARIABLES (COLUMNS) ON THE BASIS OF EXISTING COLUMNS**
 - Say we want to calculate the **total GDP of each nation, each year, in \$bn**
 - We'd multiply the GDP per capita by the total population, and divide by 1bn
 - We have a new data table, which is the **gapminder** data, plus an extra column
- **WRITE TEXT FOR mutate()**

```
## `mutate()`
```

The `mutate()` verb calculates new variables (columns) in a dataframe, from data in existing columns.

If we want to calculate the total GDP of each nation, each year, in \$billion, we could multiply the GDP per capita by the total population, and divide by 1billion, then put this in a new column:

```
```{r mutate}
gdpbynation <- gapminder%>%
 mutate(gdpBillion = gdpPercap * pop / 1e9)
kable(head(gdpbynation))
```
```

- **KNIT THE DOCUMENT**
 - The **R** command and its output are both displayed.



Red sticky for a question or issue



Green sticky if complete

- We can chain verbs together with pipes to **CALCULATE SEVERAL SUMMARIES IN A SINGLE summarize() COMMAND**

- We can use the output of `mutate()` in the `summarize()` command
 - We're going to calculate the **total (and standard deviation) of GDP per continent, per year**
 - Calculate total GDP first
 - Group by continent and year
 - Summarise mean and sd of GDP per capita, and total GDP
- **WRITE TEXT FOR CONCLUSION**

```
# Conclusion
```

We can perform powerful data analyses using ``dplyr`` in ``R``, while still having readable, understandable code.

For example, we can **chain verbs** to calculate the total (and standard deviation) of GDP per continent, for each year, as follows:

```
```R
  ```{r conclusion}
  gdpbyyearContinent <- gapminder %>%
    mutate(gdpBillion = gdpPerCap * pop / 1e9) %>%
    group_by(continent, year) %>%
    summarize(meangdpPerCap = mean(gdpPerCap),
              sdgdpPerCap = sd(gdpPerCap),
              meangdpBillion = mean(gdpBillion),
              sdgdpBillion = sd(gdpBillion))
  kable(head(gdpbyyearContinent))
```
```

- **KNIT THE DOCUMENT**
  - The `R` command and its output are both displayed.
- **POINT OUT THAT LEARNERS NOW HAVE A LITERATE PROGRAMMING DOCUMENT OF ALL THE MAIN `dplyr` VERBS WITH INTERACTIVE `R` CODE**



Red sticky for a question or issue



Green sticky if complete