Introduction to R (./index.html)

Introduction to R and R Studio

Overview

Teaching: 20 min
Exercises: 10 min
Questions

- How do you find your way around RStudio?
- How do you interact with R?

Objectives

- Describe the purpose and use of each pane in the RStudio IDE
- Locate buttons and options in the RStudio IDE
- Manage a workspace in an interactive R session
- Manage packages

Why should you use R?

What is R and what makes it different from the other software packages out there? (https://select-statistics.co.uk /blog/what-is-r-and-why-should-you-use-it/) R is designed specifically for statistical computing and graphics, and is able to perform any task associated with handling and manipulating data. It is free and open source, the latter meaning that anyone can interrogate the code to see what's going on – there's no black box involved. R provides a flexible analysis toolkit where all of the standard statistical techniques are built-in. Not only that, but there is a large R community who regularly contribute new functionality through add-on 'packages'. In fact, finding a particular statistical model or technique that is not already available through R is a tricky task indeed!

Introduction to RStudio

Welcome to the R portion of the Software Carpentry workshop.

Throughout this lesson, we're going to teach you some of the fundamentals of the R language as well as some best practices for organizing code for scientific projects that will make your life easier.

We'll be using RStudio: a free, open source R integrated development environment. It provides a built in editor, works on all platforms (including on servers) and provides many advantages such as integration with version control and project management.

Basic layout

When you first open RStudio, you will be greeted by three panels:

- The interactive R console/terminal (entire left)
- Environment/History (tabbed in upper right)
- Files/Plots/Packages/Help/Viewer (tabbed in lower right)

RStudio layout

Once you open files, such as R scripts, an editor panel will also open in the top left.

RStudio layout with .R file open

You can move the panels around in RStudio so that their arrangement suits you.

Work flow within RStudio

There are two main ways one can work within RStudio.

1. Test and play within the interactive R console then copy code into a .R file to run later.
   - This works well when doing small tests and initially starting off.
   - It quickly becomes laborious
2. Start writing in an .R file and use RStudio's short cut keys for the Run command to push the current line, selected lines or modified lines to the interactive R console.
   - This is a great way to start; all your code is saved for later
   - You will be able to run the file you create from within RStudio or using R's `source()` function.

Tip: Running segments of your code

RStudio offers you great flexibility in running code from within the editor window. There are buttons, menu choices, and keyboard shortcuts. To run the current line, you can 1. click on the `Run` button above the editor panel, or 2. select "Run Lines" from the "Code" menu, or 3. hit Ctrl + Return in Windows or Linux or ⌘ + Return on OS X. (This shortcut can also be seen by hovering the mouse over the button). To run a block of code, select it and then `Run` . If you have modified a line of code within a block of code you have just run, there is no need to reselct the section and `Run` , you can use the next button along, `Re-run the previous region` . This will run the previous code block including the modifications you have made.

Projects

R Studio provides in-built support for keeping all files associated with a project together. This includes the input data, R Scripts, analytical results and figures.

A good project layout will ultimately make your life easier:

- It will help ensure the integrity of your data;
- It makes it simpler to share your code with someone else (a lab-mate, collaborator, or supervisor);
- It allows you to easily upload your code with your manuscript submission;
- It makes it easier to pick the project back up after a break.

Challenge: Creating a self-contained project

We're going to create a new project in RStudio:

1. Click the "File" menu button, then "New Project".
2. Click "New Directory".
3. Click "Empty Project".
4. Type in the name of the directory to store your project, e.g. "intro-to-r".
5. Click the "Create Project" button.

Now when we start R in this project directory, or open this project with RStudio, all of our work on this project will be entirely self-contained in this directory.

Best practices for project organization

Although there is no “best” way to lay out a project, there are some general principles to adhere to that will make project management easier:

Treat raw data as read only

This is probably the most important goal of setting up a project. Data is typically time consuming and/or expensive to collect. Working with them interactively (e.g., in Excel) where they can be modified means you are never sure of where the data came from, or how it has been modified since collection. It is therefore a good idea to treat your raw data as “read-only”.

Data Cleaning

In many cases your data will be “dirty”: it will need significant processing to get into a useful format. This task is sometimes called “data munging”. It is a good idea to have particular scripts just for the munging process.

Treat generated output as disposable

Anything generated by your scripts should be treated as disposable: it should all be able to be regenerated from your scripts.

📌 Tip: Good Enough Practices for Scientific Computing

Good Enough Practices for Scientific Computing (https://github.com/swcarpentry/good-enough-practices-in-scientific-computing/blob/gh-pages/good-enough-practices-for-scientific-computing.pdf) gives the following recommendations for project organization:

1. Put each project in its own directory, which is named after the project.
2. Put text documents associated with the project in the `doc` directory.
3. Put raw data and metadata in the `data` directory, and files generated during cleanup and analysis in a `results` directory.
4. Put source for the project’s scripts and programs in the `src` directory, and programs brought in from elsewhere or compiled locally in the `bin` directory.
5. Name all files to reflect their content or function.

📌 Tip: ProjectTemplate - a possible solution

One way to automate the management of projects is to install the third-party package, `ProjectTemplate`. This package will set up an ideal directory structure for project management. This is very useful as it enables you to have your analysis pipeline/workflow organised and structured. Together with the default RStudio project functionality and Git you will be able to keep track of your work as well as be able to share your work with collaborators.

1. Install `ProjectTemplate`.
2. Load the library
3. Initialise the project:

```
install.packages("ProjectTemplate")
library("ProjectTemplate")
create.project("../my_project", merge.strategy = "allow.non.conflict")
```

For more information on ProjectTemplate and its functionality visit the home page ProjectTemplate (http://projecttemplate.net/index.html)

Save the data in the data directory

Now we have a good directory structure we will now place/save the data file in the `data/` directory.

Challenge 1

Download the gapminder data from here (../data/gapminder.csv).

1. Download the file (right mouse click -> “Save as”)
2. Make sure it’s saved under the name `gapminder.csv`
3. Save the file in the `data/` folder within your project.

We will load and inspect these data later.

Challenge 2

It is useful to get some general idea about the dataset, directly from the command line, before loading it into R. Understanding the dataset better will come in handy when making decisions on how to load it in R. Use the command-line shell to answer the following questions:

1. What is the size of the file?
2. How many rows of data does it contain?
3. What kinds of values are stored in this file?

Solution to Challenge 2

By running these commands in the shell:

```
ls -lh data/gapminder.csv
```

```
-rw-r--r--  1 pea25i  348785    80K 20 Mar 11:17 data/gapminder.csv
```

The file size is 80K.

```
wc -l data/gapminder.csv
```

```
    1705 data/gapminder.csv
```

There are 1705 lines. The data looks like:

```
head data/gapminder.csv
```

```
country,continent,year,lifeExp,pop,gdpPercap
Afghanistan,Asia,1952,28.801,8425333,779.4453145
Afghanistan,Asia,1957,30.332,9240934,820.8530296
Afghanistan,Asia,1962,31.997,10267083,853.10071
Afghanistan,Asia,1967,34.02,11537966,836.1971382
Afghanistan,Asia,1972,36.088,13079460,739.9811058
Afghanistan,Asia,1977,38.438,14880372,786.11336
Afghanistan,Asia,1982,39.854,12881816,978.0114388
Afghanistan,Asia,1987,40.822,13867957,852.3959448
Afghanistan,Asia,1992,41.674,16317921,649.3413952
```

Tip: command line in R Studio

You can quickly open up a shell in RStudio using the **Tools -> Shell…** menu item.

Key Points

- Use RStudio to write and run R programs.
- Set up an RStudio project for each analysis you are performing.

Using R

Overview

Teaching: 50 min
Exercises: 15 min
Questions

- What can R do?
- How can I install new packages?

Objectives

- Define a variable
- Assign data to a variable
- Manage a workspace in an interactive R session
- Use mathematical and comparison operators
- Call functions
- Manage packages

Introduction to R

Your time in R will be split between the R interactive console working in scripts. The console is where you will run all of your code, and can be a useful environment to try out ideas before adding them to an R script file. This console in RStudio is the same as the one you would get if you typed in `R` in your command-line environment.

Anything that you want to keep needs to go in a script.

The first thing you will see in the R interactive session is a bunch of information, followed by a ">" and a blinking cursor. In many ways this is similar to the shell environment you learned about during the shell lessons: it operates on the same idea of a "Read, evaluate, print loop": you type in commands, press Return, R tries to execute them, and then returns a result. Alternatively, you can type the commands into a script file and send them to the R console to be executed with Ctrl+Return (in Windows and Linux) or ⌘+Return on OS X.

Using R as a calculator

The simplest thing you could do with R is do arithmetic:

```
1 + 100
```

```
[1] 101
```

And R will print out the answer, with a preceding “[1]”. Don’t worry about this for now, we’ll explain that later. For now think of it as indicating output.

Like bash, if you type in an incomplete command, R will wait for you to complete it:

```
> 1 +
```

```
+
```

Any time you hit return and the R session shows a “+” instead of a “>”, it means it’s waiting for you to complete the command. If you want to cancel a command you can simply hit Esc and RStudio will give you back the “>” prompt.

Tip: Cancelling commands

If you’re using R from the commandline instead of from within RStudio, you need to use Ctrl+C instead of Esc to cancel the command. This applies to Mac users as well!

Cancelling a command isn’t only useful for killing incomplete commands: you can also use it to tell R to stop running code (for example if it’s taking much longer than you expect), or to get rid of the code you’re currently writing.

When using R as a calculator, the order of operations is the same as you would have learned back in school.

From highest to lowest precedence:

- Parentheses: `(`, `)`
- Exponents: `^` or `**`
- Divide: `/`
- Multiply: `*`
- Add: `+`
- Subtract: `-`

```
3 + 5 * 2
```

```
[1] 13
```

Use parentheses to group operations in order to force the order of evaluation if it differs from the default, or to make clear what you intend.

```
(3 + 5) * 2
```

```
[1] 16
```

This can get unwieldy when not needed, but clarifies your intentions. Remember that others may later read your code.

```
(3 + (5 * (2 ^ 2))) # hard to read
3 + 5 * 2 ^ 2       # clear, if you remember the rules
3 + 5 * (2 ^ 2)     # if you forget some rules, this might help
```

The text after each line of code is called a “comment”. Anything that follows after the hash (or octothorpe) symbol `#` is ignored by R when it executes code.

Really small or large numbers get a scientific notation:

```
2/10000
```

```
[1] 2e-04
```

Which is shorthand for “multiplied by `10^XX`”. So `2e-4` is shorthand for `2 * 10^(-4)`.

You can write numbers in scientific notation too:

```
5e3  # Note the lack of minus here
```

```
[1] 5000
```

Comparing things

We can also do comparison in R:

```
1 == 1  # equality (note two equals signs, read as "is equal to")
```

```
[1] TRUE
```

```
1 != 2  # inequality (read as "is not equal to")
```

```
[1] TRUE
```

```
1 < 2  # less than
```

```
[1] TRUE
```

```
1 <= 1  # less than or equal to
```

```
[1] TRUE
```

```
1 >= -9 # greater than or equal to
```

```
[1] TRUE
```

📌 Tip: Comparing Numbers

A word of warning about comparing numbers: you should never use `==` to compare two numbers unless they are integers (a data type which can specifically represent only whole numbers).

Computers may only represent decimal numbers with a certain degree of precision, so two numbers which look the same when printed out by R, may actually have different underlying representations and therefore be different by a small margin of error (called Machine numeric tolerance).

Instead you should use the `all.equal` function.

Further reading: http://floating-point-gui.de/ (http://floating-point-gui.de/)

Variables and assignment

We can store values in variables using the assignment operator `<-` , like this:

```
x <- 1/40
```

Notice that assignment does not print a value. Instead, we stored it for later in something called a **variable**. `x` now contains the **value** `0.025` :

```
x
```

```
[1] 0.025
```

More precisely, the stored value is a *decimal approximation* of this fraction called a floating point number (http://en.wikipedia.org/wiki/Floating_point).

Look for the `Environment` tab in one of the panes of RStudio, and you will see that `x` and its value have appeared. Our variable `x` can be used in place of a number in any calculation that expects a number:

```
log(x)
```

```
[1] -3.688879
```

Notice also that variables can be reassigned:

```
x <- 100
```

`x` used to contain the value 0.025 and and now it has the value 100.

Assignment values can contain the variable being assigned to:

```
x <- x + 1 #notice how RStudio updates its description of x on the top right tab
y <- x * 2
```

The right hand side of the assignment can be any valid R expression. The right hand side is *fully evaluated* before the assignment occurs.

Missing data

For all data types, it is possible to have missing values. In R, missing values are represented as `NA` . It is very important to understand this is different to `0` , or `""` , which are each values in themselves. `NA` means **missing** (or not available).

If you want to check whether the variable `x` is `NA` it would be natural to use the equality operator (`==`):

```
x <- NA
x == NA
```

```
[1] NA
```

Using `==` with `NA` returns `NA` , because making a comparison to a missing value doesn't make sense. Instead we need to check for the "state of missingness" rather than a value. We can do this with the `is.na()` function:

```
x <- NA
is.na(x)
```

```
[1] TRUE
```

Naming things

Object names can contain letters, numbers, underscores and periods. They cannot start with a number nor contain spaces at all. Different people use different conventions for long variable names, which include:

```
this_is_snake_case
otherPeopleUseCamelCase
some.people.use.periods
And_aFew.People_RENOUNCEconvention
```

Hadley Wickham recommends `snake_case` . What you use is up to you, but **be consistent**.

It is also possible to use the `=` operator for assignment:

```
x = 1/40
```

But this is much less common among R users. The most important thing is to **be consistent** with the operator you use. There are occasionally places where it is less confusing to use `<-` than `=` , and it is the most common symbol used in the community. So the recommendation is to use `<-` .

Challenge 1

Which of the following are valid R variable names?

```
min_height
max.height
_age
.mass
MaxLength
min-length
2widths
celsius2kelvin
```

Solution to challenge 1

The following can be used as R variables:

```
min_height
max.height
MaxLength
celsius2kelvin
```

The following creates a hidden variable:

```
.mass
```

The following will not be able to be used to create a variable

```
_age
min-length
2widths
```

Challenge 2

What will be the value of each variable after each statement in the following program?

```
mass <- 47.5
age <- 122
mass <- mass * 2.3
age <- age - 20
```

Solution to challenge 2

```
mass <- 47.5
```

This will give a value of 47.5 for the variable mass

```
age <- 122
```

This will give a value of 122 for the variable age

```
mass <- mass * 2.3
```

This will multiply the existing value of 47.5 by 2.3 to give a new value of 109.25 to the variable mass.

```
age <- age - 20
```

This will subtract 20 from the existing value of 122 to give a new value of 102 to the variable age.

Challenge 3

Run the code from the previous challenge, and write a command to compare mass to age. Is mass larger than age?

Solution to challenge 3

One way of answering this question in R is to use the `>` to set up the following:

```
mass > age
```

```
[1] TRUE
```

This should yield a boolean value of TRUE since 109.25 is greater than 102.

Commenting

Comments are 'notes' that are added to your script to provide further details for a human to read. They are ignored by R when the code is run. To add comments to your code, just type a `#` before your note text.

It's a good habit to get into writing comments about what the code you're writing is for - think of them as notes to help a future user (probably you!) understand why the code is there.

```
# this is a comment line
# a <- 5 this code will not assign 5 to the variable a
a <- 5 #but this line will
```

Functions

Functions are a stored list of instructions that can be "called" by a user. R has lots of built in functions to perform many different tasks.

To call a function, we type its name, followed by open and closing parentheses, like `function_name()` .

Many functions take *arguments*, which are a list of parameters that can be given to the function to modify its behaviour. In R, argument values are defined with an `=` sign, in the form of: `function_name(arg1 = val1, arg2 = val2)` .

Some functions have some arguments that must be defined, and others that are optional.

Tip: Remembering/finding function names and arguments

To use a function, for example `seq()` , which makes regular sequences of numbers, type `se` and hit TAB. RStudio will provide a popup which will show possible completions. Specify seq() by typing the next letter "q", or by using Up/Down arrows to select.
Inside the parentheses, RStudio will show a list of the arguments that the function expects. To display the details in the Help tab in the lower right pane, press F1.

```
seq(1, 10)  # seq function
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

Typing a `?` before the name of a function will open its help page. As well as providing a detailed description of the function and how it works, scrolling to the bottom of the help page will usually show a collection of code examples which illustrate function usage.

Mathematical functions

R has many built in mathematical functions. To call a function, we simply type its name, followed by open and closing parentheses. Anything we type inside the parentheses is called the function's arguments:

```
sin(1)  # trigonometry functions
```

```
[1] 0.841471
```

```
log10(10) # base-10 logarithm
```

```
[1] 1
```

```
exp(0.5) # e^(1/2)
```

```
[1] 1.648721
```

Vectorization

One final thing to be aware of is that R is *vectorized*, meaning that variables and functions can have vectors (which we'll describe soon) as values. In contrast to physics and mathematics, a vector in R describes a set of values in a certain order of the same data type. For example

```
1:5
```

```
[1] 1 2 3 4 5
```

```
2^(1:5)
```

```
[1]  2  4  8 16 32
```

```
x <- 1:5
2^x
```

```
[1]  2  4  8 16 32
```

This is incredibly powerful and makes for efficient and readable code.

Managing your environment

There are a few useful commands you can use to interact with objects available in an R session.

`ls` will list all of the variables and functions stored in the global environment (your working R session):

```
ls()
```

```
 [1] "a"             "age"           "args"          "dest_md"      
 [5] "mass"          "missing_pkgs"  "required_pkgs" "src_rmd"      
 [9] "x"             "y"            
```

📌 Tip: hidden objects

Like in the shell, `ls` will hide any variables or functions starting with a "." by default. To list all objects, type `ls(all.names=TRUE)` instead

Note here that we didn't give any arguments to `ls`, but we still needed to give the parentheses to tell R to call the function.

If we type `ls` by itself, R will print out the source code for that function!

```
ls
```

```
function (name, pos = -1L, envir = as.environment(pos), all.names = FALSE,
    pattern, sorted = TRUE)
{
    if (!missing(name)) {
        pos <- tryCatch(name, error = function(e) e)
        if (inherits(pos, "error")) {
            name <- substitute(name)
            if (!is.character(name))
                name <- deparse(name)
            warning(gettextf("%s converted to character string",
                sQuote(name)), domain = NA)
            pos <- name
        }
    }
    all.names <- .Internal(ls(envir, all.names, sorted))
    if (!missing(pattern)) {
        if ((ll <- length(grep("[", pattern, fixed = TRUE))) &&
            ll != length(grep("]", pattern, fixed = TRUE))) {
            if (pattern == "[") {
                pattern <- "\\["
                warning("replaced regular expression pattern '[' by  '\\\\['")
            }
            else if (length(grep("[^\\\\]\\[<-", pattern))) {
                pattern <- sub("\\[<-", "\\\\\\[<-", pattern)
                warning("replaced '[<-' by '\\\\[<-' in regular expression pattern")
            }
        }
        grep(pattern, all.names, value = TRUE)
    }
    else all.names
}
<bytecode: 0x7fb69fbace88>
<environment: namespace:base>
```

You can use `rm` to delete objects you no longer need:

```
rm(x)
```

If you have lots of things in your environment and want to delete all of them, you can pass the results of `ls` to the `rm` function:

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

In this case we've combined the two. Like the order of operations, anything inside the innermost parentheses is evaluated first, and so on.

In this case we've specified that the results of `ls` should be used for the `list` argument in `rm`. When assigning values to arguments by name, you *must* use the `=` operator!!

If instead we use `<-`, there will be unintended side effects, or you may get an error message:

```
rm(list <- ls())
```

```
Error in rm(list <- ls()): ... must contain names or character strings
```

Tip: Warnings vs. Errors

Pay attention when R does something unexpected! Errors, like above, are thrown when R cannot proceed with a calculation. Warnings on the other hand usually mean that the function has run, but it probably hasn't worked as expected.

In both cases, the message that R prints out usually give you clues how to fix a problem.

Packages

In R, a package is a collection of functions and documentation that are bundled together to provide some functionality. They are a convenient and standardised way for the community to share code with each other. As of this writing, there are over 10,000 packages available on CRAN (the comprehensive R archive network). R and RStudio have functionality for managing packages:

- You can see what packages are installed by typing `installed.packages()`
- You can install packages by typing `install.packages("packagename")` , where `packagename` is the package name, in quotes.
- You can update installed packages by typing `update.packages()`
- You can remove a package with `remove.packages("packagename")`
- You can make a package available for use with `library(packagename)`

Challenge 4

Clean up your working environment by deleting the mass and age variables.

Solution to challenge 4

We can use the `rm` command to accomplish this task

```
rm(age, mass)
```

```
Warning in rm(age, mass): object 'age' not found
```

```
Warning in rm(age, mass): object 'mass' not found
```

Before we go on, make sure you have the `tidyverse` package installed. You can check by running `library(tidyverse)` , and if it's not available, you can install it with `install.packages(tidyverse)` .

Key Points

- R has the usual arithmetic operators and mathematical functions.
- Use `<-` to assign values to variables.
- Use `ls()` to list the variables in a program.
- Use `rm()` to delete objects in a program.
- Use `install.packages()` to install packages.

Getting help in R

Overview

Teaching: 10 min
Exercises: 10 min
Questions

- How can I get help in R?

Objectives

- To be able read R help files for functions and special operators.
- To be able to use CRAN task views to identify packages to solve a problem.
- To be able to seek help from your peers.

Reading Help files

R, and every package, provide help files for functions. The general syntax to search for help on any function, “function_name”, from a specific function that is in a package loaded into your namespace (your interactive R session):

```
?function_name
help(function_name)
```

This will load up a help page in RStudio (or as plain text in R by itself).

Each help page is broken down into sections:

- Description: An extended description of what the function does.
- Usage: The arguments of the function and their default values.
- Arguments: An explanation of the data each argument is expecting.
- Details: Any important details to be aware of.
- Value: The data the function returns.
- See Also: Any related functions you might find useful.
- Examples: Some examples for how to use the function.

Different functions might have different sections, but these are the main ones you should be aware of.

Tip: Reading help files

One of the most daunting aspects of R is the large number of functions available. It would be prohibitive, if not impossible to remember the correct usage for every function you use. Luckily, the help files mean you don’t have to!

Special Operators

To seek help on special operators, use quotes:

```
?"<-"
```

Getting help on packages

Many packages come with "vignettes": tutorials and extended example documentation. Without any arguments, `vignette()` will list all vignettes for all installed packages; `vignette(package="package-name")` will list all available vignettes for `package-name`, and `vignette("vignette-name")` will open the specified vignette.

If a package doesn't have any vignettes, you can usually find help by typing `help("package-name")`.

When you kind of remember the function

If you're not sure what package a function is in, or how it's specifically spelled you can do a fuzzy search:

```
??function_name
```

When you have no idea where to begin

If you don't know what function or package you need to use CRAN Task Views (http://cran.at.r-project.org/web/views) is a specially maintained list of packages grouped into fields. This can be a good starting point.

When your code doesn't work: seeking help from your peers

If you're having trouble using a function, 9 times out of 10, the answers you are seeking have already been answered on Stack Overflow (http://stackoverflow.com/). You can search using the `[r]` tag.

If you can't find the answer, there are a few useful functions to help you ask a question from your peers:

```
?dput
```

Will dump the data you're working with into a format so that it can be copy and pasted by anyone else into their R session.

`sessionInfo()` will print out your current version of R, as well as any packages you have loaded. This can be useful for others to help reproduce and debug your issue.

```
sessionInfo()
```

```
R version 3.5.1 (2018-07-02)
Platform: x86_64-apple-darwin15.6.0 (64-bit)
Running under: macOS High Sierra 10.13.6

Matrix products: default
BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib

locale:
[1] en_AU.UTF-8/en_AU.UTF-8/en_AU.UTF-8/C/en_AU.UTF-8/en_AU.UTF-8

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] knitr_1.21              requirements_0.0.0.9000 remotes_2.0.2

loaded via a namespace (and not attached):
 [1] compiler_3.5.1  magrittr_1.5    htmltools_0.3.6 tools_3.5.1
 [5] yaml_2.2.0      Rcpp_1.0.0      stringi_1.3.1   rmarkdown_1.11
 [9] stringr_1.4.0   xfun_0.5        digest_0.6.18   rlang_0.3.1
[13] evaluate_0.12
```

Challenge 1

Look at the help for the `log` function. What is the default base used to calculate logarithms? What is the name of the argument used to change the base?

Solution to Challenge 1

The `log()` function calculate natural logarithms by default. To change the base, you can set the `base` argument.

Challenge 2

Look at the help for the `paste` function. You'll need to use this later. What is the difference between the `sep` and `collapse` arguments?

Solution to Challenge 2

To look at the help for the `paste()` function, use:

```
help("paste")
?paste
```

The difference between `sep` and `collapse` is a little tricky. The `paste` function accepts any number of arguments, each of which can be a vector of any length. The `sep` argument specifies the string used between concatenated terms — by default, a space. The result is a vector as long as the longest argument supplied to `paste`. In contrast, `collapse` specifies that after concatenation the elements are *collapsed* together using the given separator, the result being a single string. e.g.

```
paste(c("a","b"), "c")
```

```
[1] "a c" "b c"
```

```
paste(c("a","b"), "c", sep = ",")
```

```
[1] "a,c" "b,c"
```

```
paste(c("a","b"), "c", collapse = "|")
```

```
[1] "a c|b c"
```

```
paste(c("a","b"), "c", sep = ",", collapse = "|")
```

```
[1] "a,c|b,c"
```

(For more information, scroll to the bottom of the `?paste` help page and look at the examples, or try `example('paste')`.)

Challenge 3

Use help to find a function (and its associated parameters) that you could use to load data from a csv file in which columns are delimited with "\t" (tab) and the decimal point is a "." (period). This check for decimal separator is important, especially if you are working with international colleagues, because different countries have different conventions for the decimal point (i.e. comma vs period). hint: use `??csv` to lookup csv related functions.

Solution to Challenge 3

The standard R function for reading tab-delimited files with a period decimal separator is read.delim(). You can also do this with `read.table(file, sep="\t")` (the period is the *default* decimal separator for `read.table()` , although you may have to change the `comment.char` argument as well if your data file contains hash (#) characters

Other ports of call

- Quick R (http://www.statmethods.net/)
- RStudio cheat sheets (http://www.rstudio.com/resources/cheatsheets/)
- Cookbook for R (http://www.cookbook-r.com/)

Key Points

- Use `help()` to get online help in R.

Tidy Data

Overview

Teaching: 20 min
Exercises: 10 min
Questions

- How can I use a consistent underlying data structure?

Objectives

- To recognise tidy or messy data.
- To know where to find more information about the tidyverse.

> Happy families are all alike; every unhappy family is unhappy in its own way
>
> *Leo Tolstoy*

> Tidy data are all alike; every untidy data is untidy in its own way
>
> *Hadley Wickham*

Data can come in many different shapes and forms, and often people invent whatever makes sense to them. This often means that a great deal of time is spent modifying data to be structured in a format that R can use.

Within R, different packages can have different expectations about data structures, which can make it difficult to move between functions in different packages.

The tidyverse (https://www.tidyverse.org/) is a subset of R packages that conform to a particular philosophy about data structure.

The concept of tidy data can be distilled into three principles. A data set can be considered 'tidy' if:

1. Each variable is in its own column
2. Each case is in its own row
3. Each value is in its own cell

Challenge 1

In the following table, what makes it untidy?

id	rep1	rep2
1	1.44	2.07
2	1.77	2.13
3	3.56	3.72

Challenge 2

What it would look like if it was tidy?

Solution to Challenge 2

id	rep	value
1	1	1.44
2	1	1.77
3	1	3.56
1	2	2.07
2	2	2.13
3	2	3.72

In the above the same variable (a measurement value) was stored in two different columns. In this case making the data tidy required converting those two columns into one, which made the dataset have twice as many rows. This is usually called going from "wide" to "long" format, which is often done in the simplest cases of tidying data.

Challenge 3

Open the file `plates.xlsx` (download here (../data/plates.xlsx)). This is a very common format to store data from 96-well plates. What would this look like if it was tidy? Discuss the steps you would need to go through to convert it to a tidy format.

There is a tidyverse package, which doesn't have any functionality, except to attach core packages of the tidyverse.

```
library(tidyverse)
```

```
── Attaching packages ─────────────────────────────────── tidyverse 1.2.1 ──
```

```
✓ ggplot2 3.1.0      ✓ purrr   0.3.0
✓ tibble  2.0.1      ✓ dplyr   0.8.0.1
✓ readr   1.3.1      ✓ stringr 1.4.0
✓ ggplot2 3.1.0      ✓ forcats 0.3.0
```

```
Warning: package 'tibble' was built under R version 3.5.2
```

```
Warning: package 'purrr' was built under R version 3.5.2
```

```
Warning: package 'dplyr' was built under R version 3.5.2
```

```
Warning: package 'stringr' was built under R version 3.5.2
```

```
── Conflicts ────────────────────────────────────── tidyverse_conflicts() ──
✕ dplyr::filter()     masks stats::filter()
✕ dplyr::group_rows() masks kableExtra::group_rows()
✕ dplyr::lag()        masks stats::lag()
```

is the equivalent of

```
library(ggplot2)
library(dplyr)
library(tidyr)
library(readr)
library(purrr)
library(tibble)
library(stringr)
library(forcats)
```

The power of maintaining a consistent approach to data structures will become more clear as we work with data in R. You will see that by transforming data to make it tidy, all subsequent work will be easier to understand, and will make your code more clear as well.

Other great resources

- The original tidy data paper (https://www.jstatsoft.org/article/view/v059i10/v59i10.pdf)

Key Points

- For data to be tidy, each variable must be in its own column
- For data to be tidy, each case must be in its own row
- For data to be tidy, each value must be in its own cell

Dataframes

Overview

Teaching: 25 min
Exercises: 15 min
Questions

- What is a dataframe?
- Why use a dataframe as a tidy data structure?

Objectives

- To learn how to create a dataframe.
- To understand how to find basic information about a dataframe
- To know how to inspect the data in a dataframe

Now that we understand a little bit about why we might prefer our data to be 'tidy', we have one data structure left to learn - the dataframe. Dataframes are where the vast majority of work in R is done. Dataframes can look a lot like any table of data (and we will often refer to them in that way), but they are a very particular structure. Dataframes are a special type of list that is made up of vectors that all have to be the same length. Since vectors must have the same data types, this means that a dataframe produces a rectangular table of data, where each column **must** have the same type. Dataframes are an ideal format for storing and working with tidy data. All the tidyverse tools we will be learning from here are designed to work with data in this form.

Creating a data frame

In order to start working with data, we first need to learn how to read it in to R. For learning how to work with data, we will be using records from the Gapminder (www.gapminder.org) organisation, which contains various statistics for 142 countries betwen 1952 and 2007. This data *is* available as an R package (https://cran.r-project.org /web/packages/gapminder/index.html), but we have prepared a csv version (../data/gapminder.csv) for you to practice with.

Challenge 1

Download the gapminder.csv file and save it in your project directory.

Open the file in a text editor and describe what statistics are recorded.

Solution to Challenge 1

Using the ideas discussed previously about project structure (../01-R-and-RStudio/index.html), we will save the files into a `data` directory within our project. We can then access them with a relative path `data/gapminder.csv` .

Opening the file we can see that there are six columns of data: a country name and continent, the year that the data was recorded, and the life expectancy, population and GDP per capita.

To load this data into R, we will use the `read_csv` function from the `readr` (http://readr.tidyverse.org) package (which will be loaded automatically if you have preciously run `library(tidyverse)` , but here we will load it separately). For reading in different data formats, or for control of the import options, see the optional section on reading data in to R (../17-Additional-content---Reading-Data-In/index.html).

```
library(readr)

gapminder <- read_csv("data/gapminder.csv")
```

```
Parsed with column specification:
cols(
  country = col_character(),
  continent = col_character(),
  year = col_double(),
  lifeExp = col_double(),
  pop = col_double(),
  gdpPercap = col_double()
)
```

```
gapminder
```

```
# A tibble: 1,704 x 6
   country     continent  year lifeExp      pop gdpPercap
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.
 2 Afghanistan Asia       1957    30.3  9240934      821.
 3 Afghanistan Asia       1962    32.0 10267083      853.
 4 Afghanistan Asia       1967    34.0 11537966      836.
 5 Afghanistan Asia       1972    36.1 13079460      740.
 6 Afghanistan Asia       1977    38.4 14880372      786.
 7 Afghanistan Asia       1982    39.9 12881816      978.
 8 Afghanistan Asia       1987    40.8 13867957      852.
 9 Afghanistan Asia       1992    41.7 16317921      649.
10 Afghanistan Asia       1997    41.8 22227415      635.
# … with 1,694 more rows
```

What's a tibble?

You might notice in the output above that it calls itself a tibble, rather than a data frame. A tibble is just the tidyverse's version of a data frame that has a few behaviours tweaked to make it behave more predictibly. Try comparing the output of a base R `data.frame` version of gapminder with `as.data.frame(gapminder)` to get some idea of the differences.

We will try to refer to them as data frames throughout these lessons. But know that dataframes and tibbles are interchangable for our purposes.

Inspecting a dataframe

Looking at the printed output from the `gapminder` dataframe can tell us a lot of information about it. The first line tells us the dimensions of the data, in this case there are 1,704 rows and 6 columns. This information can also be found with:

```
nrow(gapminder)
```

```
[1] 1704
```

```
ncol(gapminder)
```

```
[1] 6
```

```
dim(gapminder)
```

```
[1] 1704    6
```

The next row of output gives the names of the columns, which can also be found using `colnames()` .

```
colnames(gapminder)
```

```
[1] "country"   "continent" "year"      "lifeExp"   "pop"       "gdpPercap"
```

The next row tells you the data type of each column, followed by the data itself. We won't discuss data types in too much detail, but see this section (../16-Additional-content---Data-types/index.html) for a full description.

Should you need the data from a specific column, it can be accessed using the `$` notation.

```
gapminder$lifeExp
```

```
 [1] 28.801 30.332 31.997 34.020 36.088 38.438 39.854 40.822 41.674 41.763
[11] 42.129 43.828 55.230 59.280 64.820 66.220 67.690 68.930 70.420 72.000
[21] 71.581 72.950 75.651 76.423 43.077 45.685 48.303 51.407 54.518 58.014
[31] 61.368 65.799 67.744 69.152 70.994 72.301 30.015 31.999 34.000 35.985
[41] 37.928 39.483 39.942 39.906 40.647 40.963 41.003 42.731 62.485 64.399
[51] 65.142 65.634 67.065 68.481 69.942 70.774 71.868 73.275 74.340 75.320
[61] 69.120 70.330 70.930 71.100 71.930 73.490 74.740 76.320 77.560 78.830
[71] 80.370 81.235 66.800 67.480 69.540 70.140 70.630 72.170 73.180 74.940
 [ reached getOption("max.print") -- omitted 1624 entries ]
```

Overview of a dataframe

There are many other ways to view the data and look at its data types, and the structure of the data. To look at the first 5 rows of the gapminder dataset, use `head()` :

```
head(gapminder, 5)
```

```
# A tibble: 5 x 6
  country     continent  year lifeExp      pop gdpPercap
  <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>
1 Afghanistan Asia       1952    28.8  8425333      779.
2 Afghanistan Asia       1957    30.3  9240934      821.
3 Afghanistan Asia       1962    32.0 10267083      853.
4 Afghanistan Asia       1967    34.0 11537966      836.
5 Afghanistan Asia       1972    36.1 13079460      740.
```

and use `tail()` to look at the last 10 rows:

```
tail(gapminder, 10)
```

```
# A tibble: 10 x 6
   country  continent  year lifeExp      pop gdpPercap
   <chr>    <chr>     <dbl>   <dbl>    <dbl>     <dbl>
 1 Zimbabwe Africa     1962    52.4  4277736      527.
 2 Zimbabwe Africa     1967    54.0  4995432      570.
 3 Zimbabwe Africa     1972    55.6  5861135      799.
 4 Zimbabwe Africa     1977    57.7  6642107      686.
 5 Zimbabwe Africa     1982    60.4  7636524      789.
 6 Zimbabwe Africa     1987    62.4  9216418      706.
 7 Zimbabwe Africa     1992    60.4 10704340      693.
 8 Zimbabwe Africa     1997    46.8 11404948      792.
 9 Zimbabwe Africa     2002    40.0 11926563      672.
10 Zimbabwe Africa     2007    43.5 12311143      470.
```

To look at the structure of the data (particularly when there are many columns) use `glimpse()` :

```
glimpse(gapminder)
```

```
Observations: 1,704
Variables: 6
$ country   <chr> "Afghanistan", "Afghanistan", "Afghanistan", "Afghanis…
$ continent <chr> "Asia", "Asia", "Asia", "Asia", "Asia", "Asia", "Asia"…
$ year      <dbl> 1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992, …
$ lifeExp   <dbl> 28.801, 30.332, 31.997, 34.020, 36.088, 38.438, 39.854…
$ pop       <dbl> 8425333, 9240934, 10267083, 11537966, 13079460, 148803…
$ gdpPercap <dbl> 779.4453, 820.8530, 853.1007, 836.1971, 739.9811, 786.…
```

And use `summary()` to get a summarised breakdown of each column:

```
summary(gapminder)
```

```
   country           continent               year         lifeExp     
 Length:1704        Length:1704        Min.   :1952   Min.   :23.60  
 Class :character   Class :character   1st Qu.:1966   1st Qu.:48.20  
 Mode  :character   Mode  :character   Median :1980   Median :60.71  
                                       Mean   :1980   Mean   :59.47  
                                       3rd Qu.:1993   3rd Qu.:70.85  
                                       Max.   :2007   Max.   :82.60  
      pop              gdpPercap       
 Min.   :6.001e+04   Min.   :   241.2  
 1st Qu.:2.794e+06   1st Qu.:  1202.1  
 Median :7.024e+06   Median :  3531.8  
 Mean   :2.960e+07   Mean   :  7215.3  
 3rd Qu.:1.959e+07   3rd Qu.:  9325.5  
 Max.   :1.319e+09   Max.   :113523.1  
```

Factors

One special type of data in R is a *factor*, which is most often used in statistical modelling for categorical data. We can look at what these are by explicitly reading part of our data frame in as a factor.

Challenge 2

Read in a new gapminder data frame with the continent column as a factor using

```
gapminder_factor <- read_csv("data/gapminder.csv", col_types = cols(continent = col_factor()))
```

What does the output from `glimpse` and `summary` on this new data frame show you? How is it different from our original `gapminder` data frame and why do you think it has a different format for different columns?

Solution to Challenge 2

The output from `summary()` changes depending on the class of the data in the column. For the numeric columns it shows the minimum, maximum, mean and quartile values. For the factor column, it shows a count of each category. For the character column it cannot provide any useful summary.

Let's look at what has changed in this column.

```
class(gapminder_factor$continent)
```

```
[1] "factor"
```

```
gapminder_factor$continent
```

```
 [1] Asia     Asia     Asia     Asia     Asia     Asia     Asia    
 [8] Asia     Asia     Asia     Asia     Asia     Europe   Europe  
[15] Europe   Europe   Europe   Europe   Europe   Europe   Europe  
[22] Europe   Europe   Europe   Africa   Africa   Africa   Africa  
[29] Africa   Africa   Africa   Africa   Africa   Africa   Africa  
[36] Africa   Africa   Africa   Africa   Africa   Africa   Africa  
[43] Africa   Africa   Africa   Africa   Africa   Africa   Americas
[50] Americas Americas Americas Americas Americas Americas Americas
[57] Americas Americas Americas Americas Oceania  Oceania  Oceania 
[64] Oceania  Oceania  Oceania  Oceania  Oceania  Oceania  Oceania 
[71] Oceania  Oceania  Europe   Europe   Europe   Europe   Europe  
[78] Europe   Europe   Europe  
 [ reached getOption("max.print") -- omitted 1624 entries ]
Levels: Asia Europe Africa Americas Oceania
```

Here you can see that the `continent` column in the `gapminder_factor` data set is a factor that lists the continent the country is in. In the final line, you can see that it lists the possible categories of the data with `Levels: Asia Europe Africa Americas Oceania`. The levels of this factor can also be accessed using:

```
levels(gapminder_factor$continent)
```

```
[1] "Asia"     "Europe"   "Africa"   "Americas" "Oceania"
```

But what happens when we look more closely at this data using `glimpse()`

```
glimpse(gapminder_factor$continent)
```

```
 Factor w/ 5 levels "Asia","Europe",..: 1 1 1 1 1 1 1 1 1 1 ...
```

This tells us we have a factor with 5 levels, just like we expected. But when it comes to show the data itself, all we see are a bunch of numbers. This is because, to R, a factor is really just an integer underneath, with the levels telling it how to map the integer to the actual category. So a value of 1 would map to the first level (`Asia`), while a value of 2 would map to the second level (`Europe`), **etc.**.

Expecting factors to behave as characters, rather than integers, is a common cause of errors for people new to R. So always remember to inspect your data with the methods shown here to make sure it is of the right type.

Your turn

So far, you've been walked through investigating a dataframe. Let's use those skills to explore a data set you have not yet been exposed to.

Challenge 3

The `storms` data set comes built in to the tidyverse packages and contains information on hurricanes recorded in the Atlantic Ocean. It can be accessed just by typing `storms` into your console.

Using the tools you have learned so far, explore this data set and describe what it contains. Explain both the structural features of the data set as a whole, as well as its content.

Hint

If you are encountering a data type you haven't seen before, try looking at the `class()` of the column to see if that helps you work out what it is.

Solution to Challenge 3

The object `storms` is a dataframe (a tibble) with 10,010 rows and 13 columns.

- `name` and `status` are character vectors.
- `year` , `month` , `hour` , `lat` , `long` , `ts_diamater` and `hu_diameter` are numeric vectors.
- `day` , `wind` , and `pressure` are integer vectors.
- `category` is an ordered factor vector, with levels `-1 < 0 < 1 < 2 < 3 < 4 < 5`

Key Points

- Dataframes (or tibbles in the tidyverse) are lists where each element is a vector of the same length
- Use `read_csv()` to read comma separated files into a data frame
- Use `nrow()` , `ncol()` , `dim()` , or `colnames()` to find information about a dataframe
- Use `head()` , `tail()` , `summary()` , or `glimpse()` to inspect a dataframe's content

Selecting columns

Overview

Teaching: 15 min
Exercises: 10 min
Questions

- How can I select specific columns from a data frame to work with?

Objectives

- Select columns using several methods
- Rename columns to make them easier to work with

The first task we will cover is extracting specific columns from a dataset to work with. This may be needed if your data is very, very wide and you are only interested in a few columns. The gapminder data doesn't exactly meet that description, but we will continue using it to understand how the functions work.

```
gapminder <- read_csv("data/gapminder.csv")
```

select()

You can specify columns to keep from a data frame using `select()`. This function, as well as many others we will be learning today are found in the `dplyr` package of the tidyverse. At it's simplest, you provide `select()` with a data frame and the column names you would like to keep.

```
#Select the year, country and pop columns
select(gapminder, year, country, pop)
```

```
# A tibble: 1,704 x 3
    year country          pop
   <dbl> <chr>          <dbl>
 1  1952 Afghanistan  8425333
 2  1957 Afghanistan  9240934
 3  1962 Afghanistan 10267083
 4  1967 Afghanistan 11537966
 5  1972 Afghanistan 13079460
 6  1977 Afghanistan 14880372
 7  1982 Afghanistan 12881816
 8  1987 Afghanistan 13867957
 9  1992 Afghanistan 16317921
10  1997 Afghanistan 22227415
# … with 1,694 more rows
```

Here you can see that the `select()` produces a new data frame with just the columns you provided. It also keeps them in the order you specified.

You can also select columns by index position:

```
#Select the first, third and fourth columns
select(gapminder, 1, 3, 4)
```

```
# A tibble: 1,704 x 3
   country      year lifeExp
   <chr>       <dbl>   <dbl>
 1 Afghanistan  1952    28.8
 2 Afghanistan  1957    30.3
 3 Afghanistan  1962    32.0
 4 Afghanistan  1967    34.0
 5 Afghanistan  1972    36.1
 6 Afghanistan  1977    38.4
 7 Afghanistan  1982    39.9
 8 Afghanistan  1987    40.8
 9 Afghanistan  1992    41.7
10 Afghanistan  1997    41.8
# … with 1,694 more rows
```

or you can select columns to *exclude* by using negation:

```
#Remove lifeExp and pop columns
select(gapminder, -lifeExp, -pop)
```

```
# A tibble: 1,704 x 4
   country     continent  year gdpPercap
   <chr>       <chr>     <dbl>     <dbl>
 1 Afghanistan Asia       1952      779.
 2 Afghanistan Asia       1957      821.
 3 Afghanistan Asia       1962      853.
 4 Afghanistan Asia       1967      836.
 5 Afghanistan Asia       1972      740.
 6 Afghanistan Asia       1977      786.
 7 Afghanistan Asia       1982      978.
 8 Afghanistan Asia       1987      852.
 9 Afghanistan Asia       1992      649.
10 Afghanistan Asia       1997      635.
# … with 1,694 more rows
```

If the columns you want are all together in the original data frame, you can select them as a range

```
#Select all columns between year and lifeExp
select(gapminder, year:lifeExp)
```

```
# A tibble: 1,704 x 2
    year lifeExp
   <dbl>   <dbl>
 1  1952    28.8
 2  1957    30.3
 3  1962    32.0
 4  1967    34.0
 5  1972    36.1
 6  1977    38.4
 7  1982    39.9
 8  1987    40.8
 9  1992    41.7
10  1997    41.8
# … with 1,694 more rows
```

```
#Exclude all columns between year and lifeExp
select(gapminder, -(year:lifeExp))
```

```
# A tibble: 1,704 x 4
   country     continent      pop gdpPercap
   <chr>       <chr>        <dbl>     <dbl>
 1 Afghanistan Asia       8425333      779.
 2 Afghanistan Asia       9240934      821.
 3 Afghanistan Asia      10267083      853.
 4 Afghanistan Asia      11537966      836.
 5 Afghanistan Asia      13079460      740.
 6 Afghanistan Asia      14880372      786.
 7 Afghanistan Asia      12881816      978.
 8 Afghanistan Asia      13867957      852.
 9 Afghanistan Asia      16317921      649.
10 Afghanistan Asia      22227415      635.
# … with 1,694 more rows
```

📌 Save your output

In the examples above, the output of the `select()` function is being printed to the screen to demonstrate how it works. Normally, you will want to actually *use* the selected data later on, so don't forget to save the output as a variable.

```
# Just prints to screen, can't use data
select(gapminder, year, country, pop)
```

```
# A tibble: 1,704 x 3
    year country          pop
   <dbl> <chr>          <dbl>
 1  1952 Afghanistan  8425333
 2  1957 Afghanistan  9240934
 3  1962 Afghanistan 10267083
 4  1967 Afghanistan 11537966
 5  1972 Afghanistan 13079460
 6  1977 Afghanistan 14880372
 7  1982 Afghanistan 12881816
 8  1987 Afghanistan 13867957
 9  1992 Afghanistan 16317921
10  1997 Afghanistan 22227415
# … with 1,694 more rows
```

```
# Saves output to variable; can be accessed later
just_population <- select(gapminder, year, country, pop)

just_population
```

```
# A tibble: 1,704 x 3
    year country          pop
   <dbl> <chr>          <dbl>
 1  1952 Afghanistan  8425333
 2  1957 Afghanistan  9240934
 3  1962 Afghanistan 10267083
 4  1967 Afghanistan 11537966
 5  1972 Afghanistan 13079460
 6  1977 Afghanistan 14880372
 7  1982 Afghanistan 12881816
 8  1987 Afghanistan 13867957
 9  1992 Afghanistan 16317921
10  1997 Afghanistan 22227415
# … with 1,694 more rows
```

Challenge 1

Use three different methods to select just the `country` , `year` , `pop` , and `gdpPercap` columns from the `gapminder` dataframe.

Solution to Challenge 1

```
#By name
select(gapminder, country, year, pop, gdpPercap)
```

```
# A tibble: 1,704 x 4
   country      year      pop gdpPercap
   <chr>       <dbl>    <dbl>     <dbl>
 1 Afghanistan  1952  8425333      779.
 2 Afghanistan  1957  9240934      821.
 3 Afghanistan  1962 10267083      853.
 4 Afghanistan  1967 11537966      836.
 5 Afghanistan  1972 13079460      740.
 6 Afghanistan  1977 14880372      786.
 7 Afghanistan  1982 12881816      978.
 8 Afghanistan  1987 13867957      852.
 9 Afghanistan  1992 16317921      649.
10 Afghanistan  1997 22227415      635.
# … with 1,694 more rows
```

```
#By position
select(gapminder, 1, 3, 5, 6)
```

```
# A tibble: 1,704 x 4
   country      year      pop gdpPercap
   <chr>       <dbl>    <dbl>     <dbl>
 1 Afghanistan  1952  8425333      779.
 2 Afghanistan  1957  9240934      821.
 3 Afghanistan  1962 10267083      853.
 4 Afghanistan  1967 11537966      836.
 5 Afghanistan  1972 13079460      740.
 6 Afghanistan  1977 14880372      786.
 7 Afghanistan  1982 12881816      978.
 8 Afghanistan  1987 13867957      852.
 9 Afghanistan  1992 16317921      649.
10 Afghanistan  1997 22227415      635.
# … with 1,694 more rows
```

```
#By exclusion
select(gapminder, -continent, -lifeExp)
```

```
# A tibble: 1,704 x 4
   country      year      pop gdpPercap
   <chr>       <dbl>    <dbl>     <dbl>
 1 Afghanistan  1952  8425333      779.
 2 Afghanistan  1957  9240934      821.
 3 Afghanistan  1962 10267083      853.
 4 Afghanistan  1967 11537966      836.
 5 Afghanistan  1972 13079460      740.
 6 Afghanistan  1977 14880372      786.
 7 Afghanistan  1982 12881816      978.
 8 Afghanistan  1987 13867957      852.
 9 Afghanistan  1992 16317921      649.
10 Afghanistan  1997 22227415      635.
# … with 1,694 more rows
```

Select helper functions

There are a number of helper functions that can be used to select the correct columns. Some commonly used ones include `starts_with()` , `ends_with()` and `contains()` , but you can see a full list by looking at the help file (`?tidyselect::select_helpers`).

These helpers do exactly what you would expect from their names, but to see them in action:

```
#All columns that start with "co"
select(gapminder, starts_with("co"))
```

```
# A tibble: 1,704 x 2
   country     continent
   <chr>       <chr>    
 1 Afghanistan Asia     
 2 Afghanistan Asia     
 3 Afghanistan Asia     
 4 Afghanistan Asia     
 5 Afghanistan Asia     
 6 Afghanistan Asia     
 7 Afghanistan Asia     
 8 Afghanistan Asia     
 9 Afghanistan Asia     
10 Afghanistan Asia     
# … with 1,694 more rows
```

```
#All columns that contain the letter "e"
select(gapminder, contains("e"))
```

```
# A tibble: 1,704 x 4
   continent  year lifeExp gdpPercap
   <chr>     <dbl>   <dbl>     <dbl>
 1 Asia       1952    28.8      779.
 2 Asia       1957    30.3      821.
 3 Asia       1962    32.0      853.
 4 Asia       1967    34.0      836.
 5 Asia       1972    36.1      740.
 6 Asia       1977    38.4      786.
 7 Asia       1982    39.9      978.
 8 Asia       1987    40.8      852.
 9 Asia       1992    41.7      649.
10 Asia       1997    41.8      635.
# … with 1,694 more rows
```

Challenge 2

Select all columns from `gapminder` that end with the letter “p”

Solution to Challenge 2

```
select(gapminder, ends_with("p"))
```

```
# A tibble: 1,704 x 3
   lifeExp      pop gdpPercap
     <dbl>    <dbl>     <dbl>
 1    28.8  8425333      779.
 2    30.3  9240934      821.
 3    32.0 10267083      853.
 4    34.0 11537966      836.
 5    36.1 13079460      740.
 6    38.4 14880372      786.
 7    39.9 12881816      978.
 8    40.8 13867957      852.
 9    41.7 16317921      649.
10    41.8 22227415      635.
# … with 1,694 more rows
```

Challenge 3 (optional)

Run the following code:

```
select(gapminder, contains("P"))
```

Does it behave the way you expected? What additional argument do you need to provide to `contains()` to make it return only the `gdpPercap` column?

Hint

The `contains()` function is actually provided by the `tidyselect` package, try `?tidyselect::contains` if you are having trouble finding the help page for it.

Solution to Challenge 3

```
select(gapminder, contains("P"))
```

```
# A tibble: 1,704 x 3
   lifeExp      pop gdpPercap
     <dbl>    <dbl>     <dbl>
 1    28.8  8425333      779.
 2    30.3  9240934      821.
 3    32.0 10267083      853.
 4    34.0 11537966      836.
 5    36.1 13079460      740.
 6    38.4 14880372      786.
 7    39.9 12881816      978.
 8    40.8 13867957      852.
 9    41.7 16317921      649.
10    41.8 22227415      635.
# … with 1,694 more rows
```

```
select(gapminder, contains("P", ignore.case = F))
```

```
# A tibble: 1,704 x 1
   gdpPercap
       <dbl>
 1      779.
 2      821.
 3      853.
 4      836.
 5      740.
 6      786.
 7      978.
 8      852.
 9      649.
10      635.
# … with 1,694 more rows
```

rename()

`select()` *can* be used to rename columns while you are selecting them using the form `new_name = old_name` :

```
select(gapminder, country_name = country, population = pop)
```

```
# A tibble: 1,704 x 2
   country_name population
   <chr>             <dbl>
 1 Afghanistan     8425333
 2 Afghanistan     9240934
 3 Afghanistan    10267083
 4 Afghanistan    11537966
 5 Afghanistan    13079460
 6 Afghanistan    14880372
 7 Afghanistan    12881816
 8 Afghanistan    13867957
 9 Afghanistan    16317921
10 Afghanistan    22227415
# … with 1,694 more rows
```

but since it drops all the other columns, it is might not be the outcome you were looking for. Instead, `rename()` can rename columns while retaining the rest of the data frame.

```
rename(gapminder, country_name = country, population = pop)
```

```
# A tibble: 1,704 x 6
   country_name continent  year lifeExp population gdpPercap
   <chr>        <chr>     <dbl>   <dbl>      <dbl>     <dbl>
 1 Afghanistan  Asia       1952    28.8    8425333      779.
 2 Afghanistan  Asia       1957    30.3    9240934      821.
 3 Afghanistan  Asia       1962    32.0   10267083      853.
 4 Afghanistan  Asia       1967    34.0   11537966      836.
 5 Afghanistan  Asia       1972    36.1   13079460      740.
 6 Afghanistan  Asia       1977    38.4   14880372      786.
 7 Afghanistan  Asia       1982    39.9   12881816      978.
 8 Afghanistan  Asia       1987    40.8   13867957      852.
 9 Afghanistan  Asia       1992    41.7   16317921      649.
10 Afghanistan  Asia       1997    41.8   22227415      635.
# … with 1,694 more rows
```

This will often be useful to convert column names from your imported data into the consistent naming format (../02-Using-R/index.html#naming-things) you have decided on.

Challenge 4

Rename the `lifeExp` and `gdpPercap` columns to the `snake_case` format

Solution to Challenge 4

```
rename(gapminder, life_exp = lifeExp, gdp_per_cap = gdpPercap)
```

```
# A tibble: 1,704 x 6
   country     continent  year life_exp      pop gdp_per_cap
   <chr>       <chr>     <dbl>    <dbl>    <dbl>       <dbl>
 1 Afghanistan Asia       1952     28.8  8425333        779.
 2 Afghanistan Asia       1957     30.3  9240934        821.
 3 Afghanistan Asia       1962     32.0 10267083        853.
 4 Afghanistan Asia       1967     34.0 11537966        836.
 5 Afghanistan Asia       1972     36.1 13079460        740.
 6 Afghanistan Asia       1977     38.4 14880372        786.
 7 Afghanistan Asia       1982     39.9 12881816        978.
 8 Afghanistan Asia       1987     40.8 13867957        852.
 9 Afghanistan Asia       1992     41.7 16317921        649.
10 Afghanistan Asia       1997     41.8 22227415        635.
# … with 1,694 more rows
```

Key Points

- Use `select()` to choose variables from a dataframe.
- Helper functions make it easier to select the correct columns.
- Use `rename()` to rename variables without dropping columns.

Extract rows

Overview

Teaching: 15 min
Exercises: 10 min
Questions

- How can I extract rows from a data frame based on their values?

Objectives

- Filter data frames using logical operators
- Combine logical expressions in a filter

If we are looking to subset the rows of our data (rather than the columns using `select()`) we can use `filter()` . This function takes a data frame as it's first argument, then a set of conditions to check. Any row of the data frame that meets all the conditions is kept and any that fail are discarded.

For example, to get just the Australian data from the gapminder set:

```
filter(gapminder, country == "Australia")
```

```
# A tibble: 12 x 6
   country   continent  year lifeExp      pop gdpPercap
   <chr>     <chr>     <dbl>   <dbl>    <dbl>     <dbl>
 1 Australia Oceania    1952    69.1  8691212    10040.
 2 Australia Oceania    1957    70.3  9712569    10950.
 3 Australia Oceania    1962    70.9 10794968    12217.
 4 Australia Oceania    1967    71.1 11872264    14526.
 5 Australia Oceania    1972    71.9 13177000    16789.
 6 Australia Oceania    1977    73.5 14074100    18334.
 7 Australia Oceania    1982    74.7 15184200    19477.
 8 Australia Oceania    1987    76.3 16257249    21889.
 9 Australia Oceania    1992    77.6 17481977    23425.
10 Australia Oceania    1997    78.8 18565243    26998.
11 Australia Oceania    2002    80.4 19546792    30688.
12 Australia Oceania    2007    81.2 20434176    34435.
```

or to get only data from 1997 or later:

```
filter(gapminder, year >= 1997)
```

```
# A tibble: 426 x 6
   country     continent  year lifeExp      pop gdpPercap
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>
 1 Afghanistan Asia       1997    41.8 22227415      635.
 2 Afghanistan Asia       2002    42.1 25268405      727.
 3 Afghanistan Asia       2007    43.8 31889923      975.
 4 Albania     Europe     1997    73.0  3428038     3193.
 5 Albania     Europe     2002    75.7  3508512     4604.
 6 Albania     Europe     2007    76.4  3600523     5937.
 7 Algeria     Africa     1997    69.2 29072015     4797.
 8 Algeria     Africa     2002    71.0 31287142     5288.
 9 Algeria     Africa     2007    72.3 33333216     6223.
10 Angola      Africa     1997    41.0  9875024     2277.
# … with 416 more rows
```

Filter operators

Any of the standard (comparison operators)(../02-Using-R/index.html#comparing-things) can be used in a filter

- `==` equal to
- `!=` not equal to
- `<` / `>` less/greater than
- `<=` / `>=` less/greater than or equal to

Any function that produces a `TRUE` / `FALSE` output (such as the `is.na()` function) can be used as well.

Challenge 1

Extract all rows from `gapminder` where the recorded life expectancy is 80 or higher.

Then try to extract just the rows from European countries.

Solution to Challenge 1

```
filter(gapminder, lifeExp >= 80)
```

```
# A tibble: 22 x 6
   country          continent  year lifeExp      pop gdpPercap
   <chr>            <chr>     <dbl>   <dbl>    <dbl>     <dbl>
 1 Australia        Oceania    2002    80.4 19546792    30688.
 2 Australia        Oceania    2007    81.2 20434176    34435.
 3 Canada           Americas   2007    80.7 33390141    36319.
 4 France           Europe     2007    80.7 61083916    30470.
 5 Hong Kong, China Asia       1997    80    6495918    28378.
 6 Hong Kong, China Asia       2002    81.5  6762476    30209.
 7 Hong Kong, China Asia       2007    82.2  6980412    39725.
 8 Iceland          Europe     2002    80.5   288030    31163.
 9 Iceland          Europe     2007    81.8   301931    36181.
10 Israel           Asia       2007    80.7  6426679    25523.
# … with 12 more rows
```

```
filter(gapminder, continent == "Europe")
```

```
# A tibble: 360 x 6
   country continent  year lifeExp     pop gdpPercap
   <chr>   <chr>     <dbl>   <dbl>   <dbl>     <dbl>
 1 Albania Europe     1952    55.2 1282697     1601.
 2 Albania Europe     1957    59.3 1476505     1942.
 3 Albania Europe     1962    64.8 1728137     2313.
 4 Albania Europe     1967    66.2 1984060     2760.
 5 Albania Europe     1972    67.7 2263554     3313.
 6 Albania Europe     1977    68.9 2509048     3533.
 7 Albania Europe     1982    70.4 2780097     3631.
 8 Albania Europe     1987    72   3075321     3739.
 9 Albania Europe     1992    71.6 3326498     2497.
10 Albania Europe     1997    73.0 3428038     3193.
# … with 350 more rows
```

If you provide multiple filter conditions (separated by a comma) then only rows matching **all** of the conditions will be kept.

```
filter(gapminder, country == "Australia", year >= 1997)
```

```
# A tibble: 3 x 6
  country   continent  year lifeExp      pop gdpPercap
  <chr>     <chr>     <dbl>   <dbl>    <dbl>     <dbl>
1 Australia Oceania    1997    78.8 18565243    26998.
2 Australia Oceania    2002    80.4 19546792    30688.
3 Australia Oceania    2007    81.2 20434176    34435.
```

Challenge 2

How would you extract just rows that are healthy (life expectancy of 80 or higher) **and** rich (GDP per capita of $30,000 or higher)?

Solution to Challenge 2

```
filter(gapminder, lifeExp >= 80, gdpPercap >= 30000)
```

```
# A tibble: 13 x 6
   country          continent  year lifeExp       pop gdpPercap
   <chr>            <chr>     <dbl>   <dbl>     <dbl>     <dbl>
 1 Australia        Oceania    2002    80.4  19546792    30688.
 2 Australia        Oceania    2007    81.2  20434176    34435.
 3 Canada           Americas   2007    80.7  33390141    36319.
 4 France           Europe     2007    80.7  61083916    30470.
 5 Hong Kong, China Asia       2002    81.5   6762476    30209.
 6 Hong Kong, China Asia       2007    82.2   6980412    39725.
 7 Iceland          Europe     2002    80.5    288030    31163.
 8 Iceland          Europe     2007    81.8    301931    36181.
 9 Japan            Asia       2007    82.6 127467972    31656.
10 Norway           Europe     2007    80.2   4627926    49357.
11 Sweden           Europe     2007    80.9   9031088    33860.
12 Switzerland      Europe     2002    80.6   7361757    34481.
13 Switzerland      Europe     2007    81.7   7554661    37506.
```

If you need to keep all rows that meet one or the other condition, use the logical OR operator (`|`). `|` resolves as `TRUE` if either of the left and right conditions are `TRUE` .

```
# Tries to find data from Australia AND New Zealand (and returns an empty table)
filter(gapminder, country == "Australia", country == "New Zealand")
```

```
# A tibble: 0 x 6
# … with 6 variables: country <chr>, continent <chr>, year <dbl>,
#   lifeExp <dbl>, pop <dbl>, gdpPercap <dbl>
```

```
# Use | to look for data from Australia OR New Zealand
filter(gapminder, country == "Australia" | country == "New Zealand")
```

```
# A tibble: 24 x 6
   country   continent  year lifeExp      pop gdpPercap
   <chr>     <chr>     <dbl>   <dbl>    <dbl>     <dbl>
 1 Australia Oceania    1952    69.1  8691212    10040.
 2 Australia Oceania    1957    70.3  9712569    10950.
 3 Australia Oceania    1962    70.9 10794968    12217.
 4 Australia Oceania    1967    71.1 11872264    14526.
 5 Australia Oceania    1972    71.9 13177000    16789.
 6 Australia Oceania    1977    73.5 14074100    18334.
 7 Australia Oceania    1982    74.7 15184200    19477.
 8 Australia Oceania    1987    76.3 16257249    21889.
 9 Australia Oceania    1992    77.6 17481977    23425.
10 Australia Oceania    1997    78.8 18565243    26998.
# … with 14 more rows
```

It might seem natural to write `country == "Australia" | "New Zealand"` but try it and you will see that you get an error. Each side of the `|` operator must result in a `TRUE` or `FALSE`, and "New Zealand" is neither.

Challenge 3

Modify your answer to Challenge 2 so that you extract the rows where countries are **either** healthy (life expectancy of 80 or higher) or rich (GDP per capita of $30,000 or higher).

Solution to Challenge 3

```
filter(gapminder, lifeExp >= 80 | gdpPercap >= 30000)
```

```
# A tibble: 65 x 6
   country   continent  year lifeExp      pop gdpPercap
   <chr>     <chr>     <dbl>   <dbl>    <dbl>     <dbl>
 1 Australia Oceania    2002    80.4 19546792    30688.
 2 Australia Oceania    2007    81.2 20434176    34435.
 3 Austria   Europe     2002    79.0  8148312    32418.
 4 Austria   Europe     2007    79.8  8199783    36126.
 5 Belgium   Europe     2002    78.3 10311970    30486.
 6 Belgium   Europe     2007    79.4 10392226    33693.
 7 Canada    Americas   2002    79.8 31902268    33329.
 8 Canada    Americas   2007    80.7 33390141    36319.
 9 Denmark   Europe     2002    77.2  5374693    32167.
10 Denmark   Europe     2007    78.3  5468120    35278.
# … with 55 more rows
```

When you have many possible matches from a row that you want to keep, writing a long expression with many `|` can be time consuming and error prone. Instead, the `%in%` operator can be used to simplify things. `%in%` returns `TRUE` if the left hand side is found somewhere in the right hand side and `FALSE` otherwise.

```
filter(gapminder, country %in% c("Australia", "New Zealand"))
```

```
# A tibble: 24 x 6
   country   continent  year lifeExp      pop gdpPercap
   <chr>     <chr>     <dbl>   <dbl>    <dbl>     <dbl>
 1 Australia Oceania    1952    69.1  8691212    10040.
 2 Australia Oceania    1957    70.3  9712569    10950.
 3 Australia Oceania    1962    70.9 10794968    12217.
 4 Australia Oceania    1967    71.1 11872264    14526.
 5 Australia Oceania    1972    71.9 13177000    16789.
 6 Australia Oceania    1977    73.5 14074100    18334.
 7 Australia Oceania    1982    74.7 15184200    19477.
 8 Australia Oceania    1987    76.3 16257249    21889.
 9 Australia Oceania    1992    77.6 17481977    23425.
10 Australia Oceania    1997    78.8 18565243    26998.
# … with 14 more rows
```

Challenge 4

Extract the rows from all countries in Africa, Asia, or Europe. How many rows does your dataframe have?

Solution to Challenge 4

```
filter(gapminder, continent %in% c("Africa","Asia", "Europe"))
```

```
# A tibble: 1,380 x 6
   country     continent  year lifeExp      pop gdpPercap
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.
 2 Afghanistan Asia       1957    30.3  9240934      821.
 3 Afghanistan Asia       1962    32.0 10267083      853.
 4 Afghanistan Asia       1967    34.0 11537966      836.
 5 Afghanistan Asia       1972    36.1 13079460      740.
 6 Afghanistan Asia       1977    38.4 14880372      786.
 7 Afghanistan Asia       1982    39.9 12881816      978.
 8 Afghanistan Asia       1987    40.8 13867957      852.
 9 Afghanistan Asia       1992    41.7 16317921      649.
10 Afghanistan Asia       1997    41.8 22227415      635.
# … with 1,370 more rows
```

Key Points

- Use `filter()` to choose data based on values.
- The logical operator `%in%` can filter data from a list of possible values

Creating New Variables

Overview

Teaching: 30 min
Exercises: 15 min
Questions

- How can I create new variables in my data frame?
- How do I deal with a data frame made up of different groups?

Objectives

- Create new columns by performing calculations on old variables.
- Use functions to create new variables.

So far, we have been looking at functions that deal just with the variables in our data frame. But what if we are needing to create *new* variables? For that task, we will use the `mutate()` function from the dplyr (https://dplyr.tidyverse.org) package of the tidyverse.

To create a new variable, you provide `mutate()` with the name of the new column and how to calculate the new value.

```
mutate(gapminder, gdp = gdpPercap * pop)
```

```
# A tibble: 1,704 x 7
   country     continent  year lifeExp      pop gdpPercap          gdp
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>        <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.  6567086330.
 2 Afghanistan Asia       1957    30.3  9240934      821.  7585448670.
 3 Afghanistan Asia       1962    32.0 10267083      853.  8758855797.
 4 Afghanistan Asia       1967    34.0 11537966      836.  9648014150.
 5 Afghanistan Asia       1972    36.1 13079460      740.  9678553274.
 6 Afghanistan Asia       1977    38.4 14880372      786. 11697659231.
 7 Afghanistan Asia       1982    39.9 12881816      978. 12598563401.
 8 Afghanistan Asia       1987    40.8 13867957      852. 11820990309.
 9 Afghanistan Asia       1992    41.7 16317921      649. 10595901589.
10 Afghanistan Asia       1997    41.8 22227415      635. 14121995875.
# … with 1,694 more rows
```

This line adds a new column to our gapminder data called `gdp` and the value in this column is calculated by multiplying the `gdpPercap` and the `pop` figure for each row.

Challenge 1

Create a new column in the `gapminder` data frame that contains the population in millions (ie. divide the population column by 1,000,000).

Solution to Challenge 1

```
mutate(gapminder, pop_m = pop/1e6)
```

```
# A tibble: 1,704 x 7
   country     continent  year lifeExp      pop gdpPercap pop_m
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl> <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.  8.43
 2 Afghanistan Asia       1957    30.3  9240934      821.  9.24
 3 Afghanistan Asia       1962    32.0 10267083      853. 10.3
 4 Afghanistan Asia       1967    34.0 11537966      836. 11.5
 5 Afghanistan Asia       1972    36.1 13079460      740. 13.1
 6 Afghanistan Asia       1977    38.4 14880372      786. 14.9
 7 Afghanistan Asia       1982    39.9 12881816      978. 12.9
 8 Afghanistan Asia       1987    40.8 13867957      852. 13.9
 9 Afghanistan Asia       1992    41.7 16317921      649. 16.3
10 Afghanistan Asia       1997    41.8 22227415      635. 22.2
# … with 1,694 more rows
```

You are not limited to mathematical operators in creating new columns. Any function that produces a value for each row can be used:

```
# Take the logarithm of the population value
mutate(gapminder, log_pop = log(pop))
```

```
# A tibble: 1,704 x 7
   country     continent  year lifeExp      pop gdpPercap log_pop
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>   <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.    15.9
 2 Afghanistan Asia       1957    30.3  9240934      821.    16.0
 3 Afghanistan Asia       1962    32.0 10267083      853.    16.1
 4 Afghanistan Asia       1967    34.0 11537966      836.    16.3
 5 Afghanistan Asia       1972    36.1 13079460      740.    16.4
 6 Afghanistan Asia       1977    38.4 14880372      786.    16.5
 7 Afghanistan Asia       1982    39.9 12881816      978.    16.4
 8 Afghanistan Asia       1987    40.8 13867957      852.    16.4
 9 Afghanistan Asia       1992    41.7 16317921      649.    16.6
10 Afghanistan Asia       1997    41.8 22227415      635.    16.9
# … with 1,694 more rows
```

```
# Abbreviate the country name
# str_sub() takes a subset of a string from the given coordinates
mutate(gapminder, country_abbr = str_sub(country, start = 1, end = 4))
```

```
# A tibble: 1,704 x 7
   country     continent  year lifeExp      pop gdpPercap country_abbr
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl> <chr>
 1 Afghanistan Asia       1952    28.8  8425333      779. Afgh
 2 Afghanistan Asia       1957    30.3  9240934      821. Afgh
 3 Afghanistan Asia       1962    32.0 10267083      853. Afgh
 4 Afghanistan Asia       1967    34.0 11537966      836. Afgh
 5 Afghanistan Asia       1972    36.1 13079460      740. Afgh
 6 Afghanistan Asia       1977    38.4 14880372      786. Afgh
 7 Afghanistan Asia       1982    39.9 12881816      978. Afgh
 8 Afghanistan Asia       1987    40.8 13867957      852. Afgh
 9 Afghanistan Asia       1992    41.7 16317921      649. Afgh
10 Afghanistan Asia       1997    41.8 22227415      635. Afgh
# … with 1,694 more rows
```

Challenge 2

Using the function `str_length()` which returns the length of a string, create a new column that contains the number of letters in the country's name.

Solution to Challenge 2

```
mutate(gapminder, num_letters = str_length(country))
```

```
# A tibble: 1,704 x 7
   country     continent  year lifeExp      pop gdpPercap num_letters
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>       <int>
 1 Afghanistan Asia       1952    28.8  8425333      779.          11
 2 Afghanistan Asia       1957    30.3  9240934      821.          11
 3 Afghanistan Asia       1962    32.0 10267083      853.          11
 4 Afghanistan Asia       1967    34.0 11537966      836.          11
 5 Afghanistan Asia       1972    36.1 13079460      740.          11
 6 Afghanistan Asia       1977    38.4 14880372      786.          11
 7 Afghanistan Asia       1982    39.9 12881816      978.          11
 8 Afghanistan Asia       1987    40.8 13867957      852.          11
 9 Afghanistan Asia       1992    41.7 16317921      649.          11
10 Afghanistan Asia       1997    41.8 22227415      635.          11
# … with 1,694 more rows
```

In fact, anything that produces a vector of the same length as the data frame can be used. It does not even have to reference data within the data frame.

```
# 1704 rows in gapminder
index_numbers <- 1:1704

mutate(gapminder, row_num = index_numbers)
```

```
# A tibble: 1,704 x 7
   country     continent  year lifeExp      pop gdpPercap row_num
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>   <int>
 1 Afghanistan Asia       1952    28.8  8425333      779.       1
 2 Afghanistan Asia       1957    30.3  9240934      821.       2
 3 Afghanistan Asia       1962    32.0 10267083      853.       3
 4 Afghanistan Asia       1967    34.0 11537966      836.       4
 5 Afghanistan Asia       1972    36.1 13079460      740.       5
 6 Afghanistan Asia       1977    38.4 14880372      786.       6
 7 Afghanistan Asia       1982    39.9 12881816      978.       7
 8 Afghanistan Asia       1987    40.8 13867957      852.       8
 9 Afghanistan Asia       1992    41.7 16317921      649.       9
10 Afghanistan Asia       1997    41.8 22227415      635.      10
# … with 1,694 more rows
```

Multiple mutations

You can perform multiple mutations at the same time by separating them with a comma.

```
mutate(gapminder, gdp = gdpPercap * pop, log_pop = log(pop))
```

```
# A tibble: 1,704 x 8
   country    continent  year lifeExp      pop gdpPercap        gdp log_pop
   <chr>      <chr>     <dbl>   <dbl>    <dbl>     <dbl>      <dbl>   <dbl>
 1 Afghanist… Asia       1952    28.8  8425333      779.    6.57e 9    15.9
 2 Afghanist… Asia       1957    30.3  9240934      821.    7.59e 9    16.0
 3 Afghanist… Asia       1962    32.0 10267083      853.    8.76e 9    16.1
 4 Afghanist… Asia       1967    34.0 11537966      836.    9.65e 9    16.3
 5 Afghanist… Asia       1972    36.1 13079460      740.    9.68e 9    16.4
 6 Afghanist… Asia       1977    38.4 14880372      786.    1.17e10    16.5
 7 Afghanist… Asia       1982    39.9 12881816      978.    1.26e10    16.4
 8 Afghanist… Asia       1987    40.8 13867957      852.    1.18e10    16.4
 9 Afghanist… Asia       1992    41.7 16317921      649.    1.06e10    16.6
10 Afghanist… Asia       1997    41.8 22227415      635.    1.41e10    16.9
# … with 1,694 more rows
```

One useful feature is that you can refer to your created columns later on in the same `mutate()` call.

```
mutate(
  gapminder,
  gdp = gdpPercap * pop,
  log_gdp = log(gdp)
)
```

```
# A tibble: 1,704 x 8
   country     continent  year lifeExp      pop gdpPercap        gdp log_gdp
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>      <dbl>   <dbl>
 1 Afghanist… Asia       1952    28.8  8425333      779.   6.57e 9    22.6
 2 Afghanist… Asia       1957    30.3  9240934      821.   7.59e 9    22.7
 3 Afghanist… Asia       1962    32.0 10267083      853.   8.76e 9    22.9
 4 Afghanist… Asia       1967    34.0 11537966      836.   9.65e 9    23.0
 5 Afghanist… Asia       1972    36.1 13079460      740.   9.68e 9    23.0
 6 Afghanist… Asia       1977    38.4 14880372      786.   1.17e10    23.2
 7 Afghanist… Asia       1982    39.9 12881816      978.   1.26e10    23.3
 8 Afghanist… Asia       1987    40.8 13867957      852.   1.18e10    23.2
 9 Afghanist… Asia       1992    41.7 16317921      649.   1.06e10    23.1
10 Afghanist… Asia       1997    41.8 22227415      635.   1.41e10    23.4
# … with 1,694 more rows
```

Challenge 3

Create a column in the gapminder data showing the life expectancy in days and one for GDP in billions of dollars

Solution to Challenge 3

```
gapminder %>%
  mutate(
    life_exp_days = lifeExp * 365,
    gdp_billion = gdpPercap * pop / 10^9
)
```

```
# A tibble: 1,704 x 8
   country continent  year lifeExp    pop gdpPercap life_exp_days
   <chr>   <chr>     <dbl>   <dbl>  <dbl>     <dbl>         <dbl>
 1 Afghan… Asia       1952    28.8 8.43e6      779.        10512.
 2 Afghan… Asia       1957    30.3 9.24e6      821.        11071.
 3 Afghan… Asia       1962    32.0 1.03e7      853.        11679.
 4 Afghan… Asia       1967    34.0 1.15e7      836.        12417.
 5 Afghan… Asia       1972    36.1 1.31e7      740.        13172.
 6 Afghan… Asia       1977    38.4 1.49e7      786.        14030.
 7 Afghan… Asia       1982    39.9 1.29e7      978.        14547.
 8 Afghan… Asia       1987    40.8 1.39e7      852.        14900.
 9 Afghan… Asia       1992    41.7 1.63e7      649.        15211.
10 Afghan… Asia       1997    41.8 2.22e7      635.        15243.
# … with 1,694 more rows, and 1 more variable: gdp_billion <dbl>
```

Variable creation functions

As mentioned before, any function that can take a vector of inputs and return a vector with the same length as an output can be used in a `mutate()` call. There are many R functions that can be used in this situation, and `dplyr` introduces a number of new functions that may be useful for creating new variables. Some functions that you might find useful are:

Offset functions

To calculate differences between observations, you may be wanting to look at the value immediately before or after it. These can be accessed using `lag()` or `lead()` respectively.

```
mutate(gapminder, life_exp_prev = lag(lifeExp), life_exp_next = lead(lifeExp))
```

```
# A tibble: 1,704 x 8
   country continent  year lifeExp    pop gdpPercap life_exp_prev
   <chr>   <chr>     <dbl>   <dbl>  <dbl>     <dbl>         <dbl>
 1 Afghan… Asia       1952    28.8 8.43e6      779.          NA  
 2 Afghan… Asia       1957    30.3 9.24e6      821.          28.8
 3 Afghan… Asia       1962    32.0 1.03e7      853.          30.3
 4 Afghan… Asia       1967    34.0 1.15e7      836.          32.0
 5 Afghan… Asia       1972    36.1 1.31e7      740.          34.0
 6 Afghan… Asia       1977    38.4 1.49e7      786.          36.1
 7 Afghan… Asia       1982    39.9 1.29e7      978.          38.4
 8 Afghan… Asia       1987    40.8 1.39e7      852.          39.9
 9 Afghan… Asia       1992    41.7 1.63e7      649.          40.8
10 Afghan… Asia       1997    41.8 2.22e7      635.          41.7
# … with 1,694 more rows, and 1 more variable: life_exp_next <dbl>
```

Cumulative computations

Cumulative sums (`cumsum()`), products (`cumprod()`) and means (`cummean()`) are all available to provide running computation of a variable. While not particularly useful for the gapminder data, you can get an idea of how they function:

```
mutate(gapminder, cumulative_life_exp = cumsum(lifeExp))
```

```
# A tibble: 1,704 x 7
   country     continent  year lifeExp     pop gdpPercap cumulative_life_e…
   <chr>       <chr>     <dbl>   <dbl>   <dbl>     <dbl>              <dbl>
 1 Afghanistan Asia       1952    28.8  8.43e6      779.               28.8
 2 Afghanistan Asia       1957    30.3  9.24e6      821.               59.1
 3 Afghanistan Asia       1962    32.0  1.03e7      853.               91.1
 4 Afghanistan Asia       1967    34.0  1.15e7      836.              125. 
 5 Afghanistan Asia       1972    36.1  1.31e7      740.              161. 
 6 Afghanistan Asia       1977    38.4  1.49e7      786.              200. 
 7 Afghanistan Asia       1982    39.9  1.29e7      978.              240. 
 8 Afghanistan Asia       1987    40.8  1.39e7      852.              280. 
 9 Afghanistan Asia       1992    41.7  1.63e7      649.              322. 
10 Afghanistan Asia       1997    41.8  2.22e7      635.              364. 
# … with 1,694 more rows
```

Challenge 4

Using an offset function, create a column showing the **change** in life expectancy between each sample period. Do any countries have no change in life expectancy for a period?

Solution to Challenge 4

```
diff_lifeExp <- mutate(gapminder, life_exp_change = lifeExp - lag(lifeExp))

filter(diff_lifeExp, life_exp_change == 0)
```

```
# A tibble: 1 x 7
  country continent  year lifeExp      pop gdpPercap life_exp_change
  <chr>   <chr>     <dbl>   <dbl>    <dbl>     <dbl>           <dbl>
1 Romania Europe     1967    66.8 19284814     6471.               0
```

Key Points

- Use `mutate()` to create new variables from old ones.
- You can create new variables using any function that returns a vector of the same length as the data frame.
- Use `group_by()` to group your data based on a variable.

Summarise and Grouping

Overview

Teaching: 20 min
Exercises: 20 min
Questions

- How can I make summaries of groups of data?
- How can I count the elements of groups in a dataset?

Objectives

- Identify when grouping is necessary to make summaries
- Be able to to usefully summarise variables in a dataframe

The `summarise()` function lets us create new variables by collapsing a data frame into a single summary statistic. You can use `summarise()` with any function that takes a vector as input and returns a single value as output. For example, what is the average life expectance in the gapminder data?

```
summarise(gapminder, mean_life_exp = mean(lifeExp))
```

```
# A tibble: 1 x 1
  mean_life_exp
          <dbl>
1          59.5
```

On it’s own, this may not seem that exciting. You could just as easily get the same result by using `mean(gapminder$lifeExp)` . Where is becomes more useful however, is that you can use multiple summary functions at the same time.

```
summarise(
    gapminder,
    mean_life_exp = mean(lifeExp),
    sd_life_exp = sd(lifeExp),
    mean_gdp_per_cap = mean(gdpPercap),
    max_gdp_per_cap = max(gdpPercap)
)
```

```
# A tibble: 1 x 4
  mean_life_exp sd_life_exp mean_gdp_per_cap max_gdp_per_cap
          <dbl>       <dbl>            <dbl>           <dbl>
1          59.5        12.9            7215.         113523.
```

Challenge 1

Calculate the mean and median population for the gapminder data

Solution to Challenge 1

```
summarise(gapminder, mean_pop = mean(pop), median_pop = median(pop))
```

```
# A tibble: 1 x 2
   mean_pop median_pop
      <dbl>      <dbl>
1 29601212.   7023596.
```

and you can get summaries for different groups in conjunction with `group_by()`

```
gapminder_by_country <- group_by(gapminder, country)

summarise(gapminder_by_country, mean_life_exp = mean(lifeExp))
```

```
# A tibble: 142 x 2
   country     mean_life_exp
   <chr>               <dbl>
 1 Afghanistan          37.5
 2 Albania              68.4
 3 Algeria              59.0
 4 Angola               37.9
 5 Argentina            69.1
 6 Australia            74.7
 7 Austria              73.1
 8 Bahrain              65.6
 9 Bangladesh           49.8
10 Belgium              73.6
# … with 132 more rows
```

Challenge 2

Adjust your answer to Challenge 1 to show the mean and median population for each continent.

Solution to Challenge 2

```
gapminder_by_country <- group_by(gapminder, country)

summarise(gapminder_by_country, mean_pop = mean(pop), median_pop = median(pop))
```

```
# A tibble: 142 x 3
   country      mean_pop median_pop
   <chr>           <dbl>      <dbl>
 1 Afghanistan 15823715.  13473708.
 2 Albania      2580249.   2644572.
 3 Algeria     19875406.  18593278.
 4 Angola       7309390.   6589530.
 5 Argentina   28602240.  28162601
 6 Australia   14649312.  14629150
 7 Austria      7583298.   7571522.
 8 Bahrain       373913.    337688.
 9 Bangladesh  90755395.  86751356
10 Belgium      9725119.   9839052.
# … with 132 more rows
```

Sorting your results

If you need to sort your resulting data frame by a particular variable, use `arrange()` . This function takes a data frame and a set of column names and it rearranges the rows so that the specified columns are in order.

```
arrange(gapminder, gdpPercap)
```

```
# A tibble: 1,704 x 6
   country          continent  year lifeExp      pop gdpPercap
   <chr>            <chr>     <dbl>   <dbl>    <dbl>     <dbl>
 1 Congo, Dem. Rep. Africa     2002    45.0 55379852      241.
 2 Congo, Dem. Rep. Africa     2007    46.5 64606759      278.
 3 Lesotho          Africa     1952    42.1   748747      299.
 4 Guinea-Bissau    Africa     1952    32.5   580653      300.
 5 Congo, Dem. Rep. Africa     1997    42.6 47798986      312.
 6 Eritrea          Africa     1952    35.9  1438760      329.
 7 Myanmar          Asia       1952    36.3 20092996      331
 8 Lesotho          Africa     1957    45.0   813338      336.
 9 Burundi          Africa     1952    39.0  2445618      339.
10 Eritrea          Africa     1957    38.0  1542611      344.
# … with 1,694 more rows
```

```
# Use desc() to sort from highest to lowest
arrange(gapminder, desc(gdpPercap))
```

```
# A tibble: 1,704 x 6
   country   continent  year lifeExp     pop gdpPercap
   <chr>     <chr>     <dbl>   <dbl>   <dbl>     <dbl>
 1 Kuwait    Asia       1957    58.0  212846   113523.
 2 Kuwait    Asia       1972    67.7  841934   109348.
 3 Kuwait    Asia       1952    55.6  160000   108382.
 4 Kuwait    Asia       1962    60.5  358266    95458.
 5 Kuwait    Asia       1967    64.6  575003    80895.
 6 Kuwait    Asia       1977    69.3 1140357    59265.
 7 Norway    Europe     2007    80.2 4627926    49357.
 8 Kuwait    Asia       2007    77.6 2505559    47307.
 9 Singapore Asia       2007    80.0 4553009    47143.
10 Norway    Europe     2002    79.0 4535591    44684.
# … with 1,694 more rows
```

Challenge 3

Calculate the average life expectancy per country. Which has the shortest average life expectancy and which has the longest average life expectancy?

Solution to Challenge 3

```
summarised_life_exp <- summarise(gapminder_by_country, mean_life_exp = mean(lifeExp))

arrange(summarised_life_exp, mean_life_exp)
```

```
# A tibble: 142 x 2
   country           mean_life_exp
   <chr>                     <dbl>
 1 Sierra Leone               36.8
 2 Afghanistan                37.5
 3 Angola                     37.9
 4 Guinea-Bissau              39.2
 5 Mozambique                 40.4
 6 Somalia                    41.0
 7 Rwanda                     41.5
 8 Liberia                    42.5
 9 Equatorial Guinea          43.0
10 Guinea                     43.2
# … with 132 more rows
```

```
arrange(summarised_life_exp, desc(mean_life_exp))
```

```
# A tibble: 142 x 2
   country     mean_life_exp
   <chr>               <dbl>
 1 Iceland              76.5
 2 Sweden               76.2
 3 Norway               75.8
 4 Netherlands          75.6
 5 Switzerland          75.6
 6 Canada               74.9
 7 Japan                74.8
 8 Australia            74.7
 9 Denmark              74.4
10 France               74.3
# … with 132 more rows
```

If you provide multiple variables to sort by, `arrange()` will initially sort by the first variable, with any ties broken by the next variable and so on.

Challenge 4

Create a new data frame called `summarised_gdp` which calculates the `mean_gdp_per_cap` per continent for each year. Remember that `group_by()` can be given multiple grouping variables.

Observe the result when you run each of the following two lines. What are the differences and when might you use one over the other?

```
arrange(summarised_gdp, desc(year), desc(mean_gdp_per_cap))
```

```
arrange(summarised_gdp, desc(mean_gdp_per_cap), desc(year))
```

Solution to Challenge 4

```
gap_by_cont_year <- group_by(gapminder, continent, year)

summarised_gdp <- summarise(gap_by_cont_year, mean_gdp_per_cap = mean(gdpPercap))

arrange(summarised_gdp, desc(year), desc(mean_gdp_per_cap))
```

```
# A tibble: 60 x 3
# Groups:   continent [5]
   continent  year mean_gdp_per_cap
   <chr>     <dbl>            <dbl>
 1 Oceania    2007           29810.
 2 Europe     2007           25054.
 3 Asia       2007           12473.
 4 Americas   2007           11003.
 5 Africa     2007            3089.
 6 Oceania    2002           26939.
 7 Europe     2002           21712.
 8 Asia       2002           10174.
 9 Americas   2002            9288.
10 Africa     2002            2599.
# … with 50 more rows
```

```
arrange(summarised_gdp, desc(mean_gdp_per_cap), desc(year))
```

```
# A tibble: 60 x 3
# Groups:   continent [5]
   continent  year mean_gdp_per_cap
   <chr>     <dbl>            <dbl>
 1 Oceania    2007           29810.
 2 Oceania    2002           26939.
 3 Europe     2007           25054.
 4 Oceania    1997           24024.
 5 Europe     2002           21712.
 6 Oceania    1992           20894.
 7 Oceania    1987           20448.
 8 Europe     1997           19077.
 9 Oceania    1982           18555.
10 Oceania    1977           17284.
# … with 50 more rows
```

Counting things

A very common summary operation is to count the number of observations. The `n()` function will help simplify this process. `n()` will return the number of rows in the data frame (or in the group if the data frame is grouped).

```
summarise(gapminder, num_rows = n())
```

```
# A tibble: 1 x 1
  num_rows
     <int>
1     1704
```

```
summarise(gapminder_by_country, num_rows = n())
```

```
# A tibble: 142 x 2
   country     num_rows
   <chr>          <int>
 1 Afghanistan       12
 2 Albania           12
 3 Algeria           12
 4 Angola            12
 5 Argentina         12
 6 Australia         12
 7 Austria           12
 8 Bahrain           12
 9 Bangladesh        12
10 Belgium           12
# … with 132 more rows
```

The `n()` function can be very useful if we need to use the number of observations in calculations. For instance, if we wanted to get the standard error of the population per country:

```
#standard error = standard deviation / square root of the number of samples
summarise(gapminder_by_country, se_pop = sd(pop) / sqrt(n()) )
```

```
# A tibble: 142 x 2
   country        se_pop
   <chr>           <dbl>
 1 Afghanistan  2053803.
 2 Albania       239192.
 3 Algeria      2486462.
 4 Angola        771421.
 5 Argentina    2178518.
 6 Australia    1130222.
 7 Austria       126342.
 8 Bahrain        60880.
 9 Bangladesh  10020393.
10 Belgium       150295.
# … with 132 more rows
```

Challenge 5

Let’s try to put together all three functions introduced here. Produce a data frame that summarises the number of rows for each continent, sorted from highest to lowest. Use `group_by()` , `summarise()` , and `arrange()` in that order to achieve it.

Solution to Challenge 5

```
gap_by_cont <- group_by(gapminder, continent)

count_by_cont <- summarise(gap_by_cont, num_rows = n())

arrange(count_by_cont, desc(num_rows))
```

```
# A tibble: 5 x 2
  continent num_rows
  <chr>        <int>
1 Africa         624
2 Asia           396
3 Europe         360
4 Americas       300
5 Oceania         24
```

Key Points

- `summarise()` creates a new variable that provides a one row summary per group
- `n()` is useful to count rows per group

Adding and Combining Datasets

Overview

Teaching: 30 min
Exercises: 30 min
Questions

- How can I combine multiple datasets?
- How can I merge datasets that have a common variable?

Objectives

- Be able to combine different datasets by row, column or common variable

All the functions we have looked at so far work with a single data frame and modify it in some way. It is common, however, that your data may not be stored in a single, complete, form and instead will be found in a number of places. Perhaps measurements taken in different weeks have been saved to separate places, or maybe you have different files recording observations and metadata. To work with data stored like this it is necessary to learn how to combine and merge different datasets into a single data frame.

Combining data

If you have new data that has the same structure as your old data, it can be added onto the end of your data frame with `bind_rows()` . We have collected the gapminder data for 2012 (../data/gapminder_2012.csv) into a separate file for you to practice this with. Download this file and save it in your data directory as `gapminder_2012.csv` .

```
gapminder_2012 <- read_csv("data/gapminder_2012.csv")
```

```
Parsed with column specification:
cols(
  country = col_character(),
  continent = col_character(),
  year = col_double(),
  lifeExp = col_double(),
  pop = col_double(),
  gdpPercap = col_double()
)
```

```
gapminder_2012
```

```
# A tibble: 134 x 6
   country     continent  year lifeExp       pop gdpPercap
   <chr>       <chr>     <dbl>   <dbl>     <dbl>     <dbl>
 1 Afghanistan Asia       2012    57.2  30700000      1840
 2 Albania     Europe     2012    77     2920000     10400
 3 Algeria     Africa     2012    76.8  37600000     13200
 4 Angola      Africa     2012    61.7  25100000      6000
 5 Argentina   Americas   2012    76.1  42100000     19200
 6 Australia   Oceania    2012    82.3  22800000     42600
 7 Austria     Europe     2012    80.9   8520000     44400
 8 Bahrain     Asia       2012    76.3   1300000     41500
 9 Bangladesh  Asia       2012    71.3 156000000      2710
10 Belgium     Europe     2012    80.3  11100000     41000
# … with 124 more rows
```

Challenge 1

Combine the 2012 data with your gapminder data using `bind_rows(gapminder, gapminder_2012)`. Explore the resulting data frame to see the effect of `bind_rows()`.

Solution to Challenge 1

```
combined_gapminder <- bind_rows(gapminder, gapminder_2012)

tail(combined_gapminder)
```

```
# A tibble: 6 x 6
  country       continent  year lifeExp       pop gdpPercap
  <chr>         <chr>     <dbl>   <dbl>     <dbl>     <dbl>
1 United States Americas   2012    78.9 313000000     50500
2 Uruguay       Americas   2012    76.5   3400000     18500
3 Venezuela     Americas   2012    75.3  29900000     17700
4 Vietnam       Asia       2012    73.6  90500000      4910
5 Zambia        Africa     2012    54.5  14700000      3510
6 Zimbabwe      Africa     2012    54.1  14700000      1850
```

The columns are matched by name, so you need to make sure that both data frames are named consistently. If the names do not match, the data frames will still be bound together but any missing data will be replaced with `NA` s

```
renamed_2012 <- rename(gapminder_2012, population = pop)

mismatched_names <- bind_rows(gapminder, renamed_2012)

mismatched_names
```

```
# A tibble: 1,838 x 7
   country     continent  year lifeExp      pop gdpPercap population
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>      <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.         NA
 2 Afghanistan Asia       1957    30.3  9240934      821.         NA
 3 Afghanistan Asia       1962    32.0 10267083      853.         NA
 4 Afghanistan Asia       1967    34.0 11537966      836.         NA
 5 Afghanistan Asia       1972    36.1 13079460      740.         NA
 6 Afghanistan Asia       1977    38.4 14880372      786.         NA
 7 Afghanistan Asia       1982    39.9 12881816      978.         NA
 8 Afghanistan Asia       1987    40.8 13867957      852.         NA
 9 Afghanistan Asia       1992    41.7 16317921      649.         NA
10 Afghanistan Asia       1997    41.8 22227415      635.         NA
# … with 1,828 more rows
```

```
tail(mismatched_names)
```

```
# A tibble: 6 x 7
  country       continent  year lifeExp   pop gdpPercap population
  <chr>         <chr>     <dbl>   <dbl> <dbl>     <dbl>      <dbl>
1 United States Americas   2012    78.9    NA     50500  313000000
2 Uruguay       Americas   2012    76.5    NA     18500    3400000
3 Venezuela     Americas   2012    75.3    NA     17700   29900000
4 Vietnam       Asia       2012    73.6    NA      4910   90500000
5 Zambia        Africa     2012    54.5    NA      3510   14700000
6 Zimbabwe      Africa     2012    54.1    NA      1850   14700000
```

Merging data

If you are instead looking to merge data sets based on some shared variable, there are a number of `join` s that are useful. The tidyexplain (https://github.com/gadenbuie/tidyexplain) package has a number of animations that will help us understand what is happening with each join.

Let's start with two simple data frames that have one shared column indicating a shared ID:

The first join we will look at is an `inner_join()` . This function takes two data frames as input and merges them based on their shared column. Only rows with data in *both* data frames are kept.

The opposite of an `inner_join()` is a `full_join()` . This function keeps all rows from both data frames, filling in any missing data with `NA` .

The final join we will look at is a `left_join()` . This uses the first data frame as a reference and merges in data from the second data frame. Any rows that are in the left data frame but not the right are filled in with `NA` . Any rows in the right data frame but not the left are ignored.

For the sake of completeness, there are also functions for a `right_join()` , `semi_join()` , and `anti_join()` . But we will not go into how they work because they are far less commonly needed than the three above.

Challenge 2

Create the following two data frames:

```
df1 <- tibble(sample = c(1, 2, 3), measure1 = c(4.2, 5.3, 6.1))
df2 <- tibble(sample = c(1, 3, 4), measure2 = c(7.8, 6.4, 9.0))
```

Predict the result of running `inner_join()` , `full_join()` , and `left_join()` on these two data frames. Perform the joins to see if you are correct.

Solution to Challenge 2

```
inner_join(df1, df2)
```

```
Joining, by = "sample"
```

```
# A tibble: 2 x 3
  sample measure1 measure2
   <dbl>    <dbl>    <dbl>
1      1      4.2      7.8
2      3      6.1      6.4
```

```
full_join(df1, df2)
```

```
Joining, by = "sample"
```

```
# A tibble: 4 x 3
  sample measure1 measure2
   <dbl>    <dbl>    <dbl>
1      1      4.2      7.8
2      2      5.3     NA
3      3      6.1      6.4
4      4     NA        9
```

```
# df1 as left data frame
left_join(df1, df2)
```

```
Joining, by = "sample"
```

```
# A tibble: 3 x 3
  sample measure1 measure2
   <dbl>    <dbl>    <dbl>
1      1      4.2      7.8
2      2      5.3     NA
3      3      6.1      6.4
```

```
# df2 as left data frame
left_join(df2, df1)
```

```
Joining, by = "sample"
```

```
# A tibble: 3 x 3
  sample measure2 measure1
   <dbl>    <dbl>    <dbl>
1      1      7.8      4.2
2      3      6.4      6.1
3      4      9       NA
```

By default, the `join` functions will join based on any shared column names between the two data frames (here just the `sample` column). You may have noticed the helpful message providing information about how the join was performed: `Joining, by = "sample"`. You can control which columns are used to merge with the `by` argument.

```
full_join(df1, df2, by = c("sample"))
```

```
# A tibble: 4 x 3
  sample measure1 measure2
   <dbl>    <dbl>    <dbl>
1      1      4.2      7.8
2      2      5.3     NA
3      3      6.1      6.4
4      4     NA        9
```

This may be useful if the data frames share a number of column names, bu only some of them should be used for merging. You can also use it to merge on a column even if the names don't match between the data frames. In this case you need to specify `by = c("left_name" = "right_name")`:

```
df3 = tibble(ID = c(1, 2, 4), measure3 = c(4.7, 3.5, 9.3))

# Can't do it automatically
full_join(df1, df3)
```

```
Error: `by` required, because the data sources have no common variables
```

```
# name in df1 is "sample", name in df3 is "ID"
full_join(df1, df3, by = c("sample" = "ID"))
```

```
# A tibble: 4 x 3
  sample measure1 measure3
   <dbl>    <dbl>    <dbl>
1      1      4.2      4.7
2      2      5.3      3.5
3      3      6.1     NA
4      4     NA        9.3
```

Realistic data

For a more realistic example of joins, let's turn back to the gapminder data. We have some additional data on countries sex ratios (../data/gapminder_sex_ratios.csv) that you should download and save to your data folder.

Read it in and let's join it to our existing data.

```
gapminder_sr <- read_csv("data/gapminder_sex_ratios.csv")
```

```
Parsed with column specification:
cols(
  country = col_character(),
  year = col_double(),
  sex_ratio = col_double()
)
```

```
gapminder_sr
```

```
# A tibble: 1,722 x 3
   country     year sex_ratio
   <chr>      <dbl>     <dbl>
 1 Burundi     1952      91.9
 2 Comoros     1952      98.8
 3 Djibouti    1952      98.6
 4 Eritrea     1952      98.2
 5 Ethiopia    1952      98.6
 6 Kenya       1952     102. 
 7 Madagascar  1952     106. 
 8 Malawi      1952      92.3
 9 Mauritius   1952      99.2
10 Mozambique  1952      95.6
# … with 1,712 more rows
```

Challenge 3

Which columns are shared between `gapminder` and `gapminder_sr` ?

Compare the output of `left` , `full` and `inner` joins for these two datasets. What are the differences? What are they due to?

Solution to Challenge 3

The two datasets share the `country` and `year` columns.

The output from `outer_join` has the most rows, because it is keeping all the rows from both dataframes. `inner_join` is only including those rows that match. and has the least rows. The `left_join` output will depend on which data frame was provided first and has the same number of rows as that data frame.

Key Points

- `bind_rows` combines datasets that share the same variables
- The `join` family of functions provide a complete range of methods to merge datasets that share common variables

Putting it all together

Overview

Teaching: 25 min
Exercises: 25 min
Questions

- How can I use multiple data verbs together?
- How should I plan my data analysis?

Objectives

- Break down an analysis into several small steps
- Understand approaches to linking analysis steps together.
- Use the pipe (`%>%`) to chain together multiple functions in a readable format.

Now that we have learned some data manipulation verbs you might begin to see how they could be used together. By combining several small and simple steps you can start to perform complex manipulations.

Challenge 1

Using both `select()` and `filter()` , create a data frame containing only the `country` , `year` , and `pop` columns for Australian data.

Your solution to this challenge probably took one of two approaches:

Using an intermediate variable

```
filtered_data <- filter(data, ...)

final_data <- select(filtered_data, ...)
```

Nesting function calls

```
final_data <- select(filter(data, ...), ...)
```

Each of these are perfectly acceptable ways to solve the problem, but can lead to code that is difficult to read and understand. In the case of the intermediate variables, you need to come up with a name for each one, and if you aren't very careful about keeping the connected lines of code together it can become difficult to track which variable is being used in each function. The nested version removes some of these problems, but it's hard enough to read with only two simple steps.

Piping in R

From the magrittr (https://magrittr.tidyverse.org) package in the tidyverse we have another approach for chaining a sequence of functions. This is the pipe operator (`%>%`), which works in a similar way to the unix pipe.

Shortcuts

Typing `%>%` in everytime you want a pipe is a bit awkward, so RStudio has a shortcut to help. Use Ctrl + Shift + M (Cmd + Shift + M on a Mac) to insert a pipe into your code.

Using the pipe

The pipe works by taking the output of it's left hand side and inserting it as the first argument to the function on it's right hand side.

This means that the following

```
some_function(data)
```

could be rewritten using a pipe as

```
data %>% some_function()
```

Other function arguments are left untouched so

```
#Standard form
some_function(data, first_arg, second_arg)

#Piped form
data %>% some_function(first_arg, second_arg)
```

are equivalent.

Challenge 2

Rewrite the following line from the previous lesson in a piped form. Does it give you the same output?

```
filter(gapminder, country == "Australia", year >= 1997)
```

Solution to Challenge 2

```
gapminder %>%
    filter(country == "Australia", year >= 1997)
```

```
# A tibble: 3 x 6
  country   continent  year lifeExp      pop gdpPercap
  <chr>     <chr>     <dbl>   <dbl>    <dbl>     <dbl>
1 Australia Oceania    1997    78.8 18565243    26998.
2 Australia Oceania    2002    80.4 19546792    30688.
3 Australia Oceania    2007    81.2 20434176    34435.
```

Combining

The real value of the pipe comes when there are multiple steps to complete. To rewrite our example above using pipes:

```
#Nested form
final_data <- select(filter(data, ...), ...)

#Piped form
final_data <- data %>%
                filter(...) %>%
                select(...)
```

This can be read as a series of instructions. Take the data frame `data`, *then* `filter` it to keep some rows, *then* `select` some columns.

For many people, the piped form is one that is easier to read and understand, as well as easier to write.

Challenge 3

Take your answer to Challenge 1 and rewrite it in a piped form.

Now, imagine that you have decided later more steps are required. Add a step renaming the `pop` column to `population` for both forms. Do you find one form easier to work with than the other?

Solution to Challenge 3

For the piped version:

```
gapminder %>%
    select(country, year, pop) %>%
    filter(country == "Australia")
```

```
# A tibble: 12 x 3
   country    year      pop
   <chr>     <dbl>    <dbl>
 1 Australia  1952  8691212
 2 Australia  1957  9712569
 3 Australia  1962 10794968
 4 Australia  1967 11872264
 5 Australia  1972 13177000
 6 Australia  1977 14074100
 7 Australia  1982 15184200
 8 Australia  1987 16257249
 9 Australia  1992 17481977
10 Australia  1997 18565243
11 Australia  2002 19546792
12 Australia  2007 20434176
```

And adding a rename step:

```
gapminder %>%
    select(country, year, pop) %>%
    filter(country == "Australia") %>%
    rename(population = pop)
```

```
# A tibble: 12 x 3
   country    year population
   <chr>     <dbl>      <dbl>
 1 Australia  1952    8691212
 2 Australia  1957    9712569
 3 Australia  1962   10794968
 4 Australia  1967   11872264
 5 Australia  1972   13177000
 6 Australia  1977   14074100
 7 Australia  1982   15184200
 8 Australia  1987   16257249
 9 Australia  1992   17481977
10 Australia  1997   18565243
11 Australia  2002   19546792
12 Australia  2007   20434176
```

Common pitfalls

When using pipes to construct a sequential analysis, there are a few problems that can trip people up.

Failing to complete the pipe

When constructing a sequence by adding or removing steps, a very common problem is that you will leave a pipe command trailing at the end of your code block.

```
#Pipe to nowhere
final_data <- data %>%
                filter(...) %>%
                select(...) %>%
```

In this case, your code will not complete and R will pause and wait for the rest of the pipe before continuing. This is easy to spot in your console because the input indicator will change from a `>` to a `+`, telling you that R is waiting for you to complete a command. Just hit the Esc key to cancel the command, and fix up your pipe before running it again.

Forgetting to assign the output

This has been covered before, but is something that often gets forgotten again once you start using pipes. If the final output of your pipe is not saved into a variable, it will just get printed to the screen and then lost.

So instead of

```
gapminder %>%
  select(country, year, pop) %>%
  filter(country == "Australia") %>%
  rename(population = pop)
```

make sure to save the output if you need access to it later

```
aust_data <- gapminder %>%
  select(country, year, pop) %>%
  filter(country == "Australia") %>%
  rename(population = pop)
```

Forgetting to pass some data in to the pipe

Some people have the opposite problem where they focus so much on getting each step of the pipe right that they forget that it needs some data to work on.

Try running the pipe above without any data:

```
select(country, year, pop) %>%
  filter(country == "Australia") %>%
  rename(population = pop)
```

```
Error in select(country, year, pop): object 'country' not found
```

It throws an error because `select()` in the first step is expecting it's first argument to be a data frame that it can work on. Instead, it finds `country`, which is not a data frame.

You could fix this by providing the data frame directly to select

```
select(gapminder, country, year, pop) %>%
  filter(country == "Australia") %>%
  rename(population = pop)
```

But it is easier to know exactly what data is going into a pipe if you put it on it's own line at the start

```
gapminder %>%
  select(country, year, pop) %>%
  filter(country == "Australia") %>%
  rename(population = pop)
```

Overly long pipes

Using pipes can make your code more understandable, but there is a limit to their effectiveness. If your pipe has too many steps it becomes harder to read through and keep in your memory all the steps that have been applied to the data.

It also makes identifying errors harder. Imagine a pipe with 30 steps that is giving you the wrong output or throwing an error. Working out which of those 30 steps is the cause of the problem can become time consuming and difficult. If you find yourself writing very long pipes, consider if there are any logical ways to break it up into a series of smaller pipes.

Style questions

There are a couple of guidelines about style you can use when writing pipes to make them more understandable

Separate each step in a pipe

Each step of the pipe should be on a separate line, and all steps after the first should be indented. This helps to clearly identify the elements of the pipe and read through them in a step-by-step fashion.

Comment sensibly

You *can* add comments into the middle of a pipe chain

```
gapminder %>%
  rename(gdpPerCap = gdpPercap) %>% # Inconsistent capitalisation annoys me
  select(country, year, gdpPerCap) %>%
  # Data version of #MapsWithoutNZ
  filter(country != "New Zealand")
```

but consider the effect those comments have on the legibility of your code. It may be best to add your comment at the start of the pipe instead. Or to break the pipe up so that the complicated parts needing comments are separated from the simpler steps.

Constructing an analysis

We have now learned a number of tools for manipulating data, as well as a convenient way to link them together. When it comes time to put them together, it can help to start at the end. Decide on what your end goal is, and then work backwards step by step to figure out how to achieve it. As an example using the gapminder data, suppose a colleague asked you which country had the sixth highest population in 1972. In order to answer that, you would need a list of the countries ordered by their population in 1972. To get such a list, you would first need to extract just the data from 1972 from the complete gapminder set. And to extract just the 1972 data (assuming you are doing this analysis in R), you would first need to import the data from somewhere.

So this very basic analysis has a number of steps to complete:

- Read gapminder data into R
- Keep only the data from 1972
- Sort this data by population size
- Get the sixth highest population size
- Look at the country with that population

Challenge 4

Using this process, what steps might you need to determine which countries are in the top ten life expectancy lists for both 1987 and 2007?

From design to implementation

Once you have sketched out an analysis, you will have a series of small, self contained steps to complete. This maps well onto the tidyverse philosophy (https://tidyverse.tidyverse.org/articles/manifesto.html) that each function should be as simple as possible, and do one thing well. Ideally each of your analysis steps will correspond to one (or a small number) of tidyverse verbs that we have covered. For example, we could take our analysis from before and add in some tidyverse verbs without losing the descriptiveness.

- `read_csv` the gapminder data into R
- `filter` only the data from 1972
- `arrange` this data by population size
- `filter` the sixth highest population size
- `select` the country variable with that population

As you become familiar with these functions and the 'grammar' of the tidyverse, it will become easier to see how to link them together step by step to complete an analysis. Eventually this process will become second nature and you will easily be able to break down complex analyses into a series of small steps that can solved using a handful of simple functions.

Challenge 5

Try and map some tidyverse verbs onto the steps you identified in Challenge 4. How many verbs did you need for each step?

Now, see if you can implement them in code to answer the question.

Key Points

- Data analyses can be broken down into discrete stages
- Most data analysis stages fit into a small number of types
- Pipes pass their left hand side through as the first argument of the right hand side.
- Pipes make your code more readable, but be careful of going overboard.

Gather & Spread

Overview

Teaching: 30 min
Exercises: 15 min
Questions

- How can I change the format of dataframes?

Objectives

- To be understand the concepts of 'long' and 'wide' data formats and be able to convert between them with tidyr.

Having had some exposure to working with tidyverse functions, you might now understand why it is beneficial to work with tidy data (../04-Tidy-Data/index.html). These data manipulation functions are all designed to work easily with tidy data, leaving you with more time to actually try to answer questions with the data.

Unfortunately, real data sets can come in all shapes and sizes and may not be structured for you to efficiently analyse it. To help tidy our data up in preparation for analysis, we will turn to the `tidyr` package from the tidyverse.

Tidy data review

Before getting into a real example, we'll have a look at a toy dataset. The following dataframe has two weight measurements for three different cows. Have a look at the way the data is structured. Is it tidy?

```
cows <- data_frame(id = c(1, 2, 3),
                   weight1 = c(203, 227, 193),
                   weight2 = c(365, 344, 329))
cows
```

```
# A tibble: 3 x 3
     id weight1 weight2
  <dbl>   <dbl>   <dbl>
1     1     203     365
2     2     227     344
3     3     193     329
```

What might make this dataset tidy?

Table massaging with `gather()` and `spread()`

To make this dataframe tidy, we may need to have only one weight variable, and another variable describing whether it is measurement 1 or 2. The `gather()` function from `tidyr` can do this conversion for us. You can think of `gather()` as scooping all the data up into one big tall pile.

```
cows_long <- cows %>%
  gather(measurement, weight, weight1, weight2)

cows_long
```

```
# A tibble: 6 x 3
     id measurement weight
  <dbl> <chr>        <dbl>
1     1 weight1        203
2     2 weight1        227
3     3 weight1        193
4     1 weight2        365
5     2 weight2        344
6     3 weight2        329
```

`gather()` works by converting the data into a set of key-value pairs, in which the *key* describes what the data is, and the *value* records the actual data. For example, `weight1 - 203` is the key-value pair for the first row of the gathered table.

The call to `gather()` therefore works in two parts. First we specify the names of the new key-value pair columns (`measurement` and `weight` in this case). Then we tell it which columns should be gathered (`weight1` and `weight2` here). `gather()` then takes the contents of those columns and places them into the new *value* column, and labels them with their column name as the *key*.

This transformation may be easier to see in action, so let's have a look at the way `gather()` has worked on the original data.

As well as going from wide to long, we can go back the other way. Instead of `gather()` , we need to `spread()` the data.

```
cows_long %>%
  spread(measurement, weight)
```

```
# A tibble: 3 x 3
     id weight1 weight2
  <dbl>   <dbl>   <dbl>
1     1     203     365
2     2     227     344
3     3     193     329
```

This works in reverse, taking the names of out key-value columns and spreading them out. Each unique value in the key column is given a separate column of it's own. And the content of that column is taken from the specified value column.

Realistic data

Until now, we've been using the original gapminder dataset which comes pre-tidied, but 'real' data (i.e. our own research data) will never be so well organised. Here let's start with the wide format version of the gapminder dataset.

> Download the wide version of the gapminder data from here (../data/gapminder_wide.csv) and save it in your project's `data` directory.

Read in the wide gapminder and have a look at it.

```
gap_wide <- read_csv("data/gapminder_wide.csv")

gap_wide
```

```
# A tibble: 142 x 38
   continent country gdpPercap_1952 gdpPercap_1957 gdpPercap_1962
   <chr>     <chr>            <dbl>          <dbl>          <dbl>
 1 Africa    Algeria          2449.          3014.          2551.
 2 Africa    Angola           3521.          3828.          4269.
 3 Africa    Benin            1063.           960.           949.
 4 Africa    Botswa…           851.           918.           984.
 5 Africa    Burkin…           543.           617.           723.
 6 Africa    Burundi           339.           380.           355.
 7 Africa    Camero…          1173.          1313.          1400.
 8 Africa    Centra…          1071.          1191.          1193.
 9 Africa    Chad             1179.          1308.          1390.
10 Africa    Comoros          1103.          1211.          1407.
# … with 132 more rows, and 33 more variables: gdpPercap_1967 <dbl>,
#   gdpPercap_1972 <dbl>, gdpPercap_1977 <dbl>, gdpPercap_1982 <dbl>,
#   gdpPercap_1987 <dbl>, gdpPercap_1992 <dbl>, gdpPercap_1997 <dbl>,
#   gdpPercap_2002 <dbl>, gdpPercap_2007 <dbl>, lifeExp_1952 <dbl>,
#   lifeExp_1957 <dbl>, lifeExp_1962 <dbl>, lifeExp_1967 <dbl>,
#   lifeExp_1972 <dbl>, lifeExp_1977 <dbl>, lifeExp_1982 <dbl>,
#   lifeExp_1987 <dbl>, lifeExp_1992 <dbl>, lifeExp_1997 <dbl>,
#   lifeExp_2002 <dbl>, lifeExp_2007 <dbl>, pop_1952 <dbl>,
#   pop_1957 <dbl>, pop_1962 <dbl>, pop_1967 <dbl>, pop_1972 <dbl>,
#   pop_1977 <dbl>, pop_1982 <dbl>, pop_1987 <dbl>, pop_1992 <dbl>,
#   pop_1997 <dbl>, pop_2002 <dbl>, pop_2007 <dbl>
```

You can see that wide format has many more columns than our original gapminder data because each metric has a separate column for each year of measurement.

From wide to long

The first step towards recreating the nice and tidy gapminder format is to convert this data from a wide to a long format. The `tidyr` function `gather()` will ‘gather’ your observation variables into a single variable.

```
gap_long <- gap_wide %>%
    gather(obstype_year, obs_values, starts_with('pop'),
           starts_with('lifeExp'), starts_with('gdpPercap'))
gap_long
```

```
# A tibble: 5,112 x 4
   continent country                  obstype_year obs_values
   <chr>     <chr>                    <chr>             <dbl>
 1 Africa    Algeria                  pop_1952        9279525
 2 Africa    Angola                   pop_1952        4232095
 3 Africa    Benin                    pop_1952        1738315
 4 Africa    Botswana                 pop_1952         442308
 5 Africa    Burkina Faso             pop_1952        4469979
 6 Africa    Burundi                  pop_1952        2445618
 7 Africa    Cameroon                 pop_1952        5009067
 8 Africa    Central African Republic pop_1952        1291695
 9 Africa    Chad                     pop_1952        2682462
10 Africa    Comoros                  pop_1952         153936
# … with 5,102 more rows
```

Inside `gather()` we first name the new column for the new ID variable (`obstype_year`), the name for the new amalgamated observation variable (`obs_value`), then the names of the old observation variable. We could have typed out all the observation variables, but since `gather()` can use all the same helper functions (../06-Data-Verbs---select/index.html#select-helper-functions) as the `select()` function can, we can use the `starts_with()` argument to select all variables that starts with the desired character string. Gather also allows the alternative syntax of using the `-` symbol to identify which variables are not to be gathered (i.e. ID variables) so that

```
gap_long <- gap_wide %>%
  gather(obstype_year, obs_values, -continent, -country)
```

is an alternative way of specifying the columns to gather in the wide data set. The best method for any particular data set will depend on how many columns you need to gather and how they are named.

Separating column data

Now, the `obstype_year` column actually contains two pieces of information, the observation type (`pop` , `lifeExp` , or `gdpPercap`) and the `year` . We can use the `separate()` function to split the character strings into multiple variables.

```
gap_separated <- gap_long %>%
  separate(obstype_year, into = c('obs_type', 'year'), sep = "_")

gap_separated
```

```
# A tibble: 5,112 x 5
   continent country                  obs_type year  obs_values
   <chr>     <chr>                    <chr>    <chr>      <dbl>
 1 Africa    Algeria                  pop      1952     9279525
 2 Africa    Angola                   pop      1952     4232095
 3 Africa    Benin                    pop      1952     1738315
 4 Africa    Botswana                 pop      1952      442308
 5 Africa    Burkina Faso             pop      1952     4469979
 6 Africa    Burundi                  pop      1952     2445618
 7 Africa    Cameroon                 pop      1952     5009067
 8 Africa    Central African Republic pop      1952     1291695
 9 Africa    Chad                     pop      1952     2682462
10 Africa    Comoros                  pop      1952      153936
# … with 5,102 more rows
```

You provide `separate()` with a column name containing the values to split, the names of the columns you would like to separate them into, and where to split the values (by default it splits on any non alphanumeric character).

Challenge 1

If you look at the table above, you will see that `separate()` creates character columns out of both the `obs_type` and `year` columns, but our original gapminder has year as a numeric. How would you fix this discrepancy?

Hint: Read through the arguments for `separate` with `?separate`

Solution to Challenge 1

The `convert` argument will try to convert character strings to other data types when set to `TRUE`

```
gap_separated <- gap_long %>%
  separate(obstype_year, into = c('obs_type', 'year'), sep = "_", convert = TRUE)

gap_separated
```

```
# A tibble: 5,112 x 5
   continent country                  obs_type  year obs_values
   <chr>     <chr>                    <chr>    <int>      <dbl>
 1 Africa    Algeria                  pop       1952    9279525
 2 Africa    Angola                   pop       1952    4232095
 3 Africa    Benin                    pop       1952    1738315
 4 Africa    Botswana                 pop       1952     442308
 5 Africa    Burkina Faso             pop       1952    4469979
 6 Africa    Burundi                  pop       1952    2445618
 7 Africa    Cameroon                 pop       1952    5009067
 8 Africa    Central African Republic pop       1952    1291695
 9 Africa    Chad                     pop       1952    2682462
10 Africa    Comoros                  pop       1952     153936
# … with 5,102 more rows
```

The opposite to `separate()` is `unite()`. Use this when you are wanting to *combine* columns into one long string. You provide `unite()` with the name to give the newly combined column, and the names of the columns to combine.

```
gap_separated %>%
  unite(obstype_year, obs_type, year)
```

```
# A tibble: 5,112 x 4
   continent country                  obstype_year obs_values
   <chr>     <chr>                    <chr>             <dbl>
 1 Africa    Algeria                  pop_1952        9279525
 2 Africa    Angola                   pop_1952        4232095
 3 Africa    Benin                    pop_1952        1738315
 4 Africa    Botswana                 pop_1952         442308
 5 Africa    Burkina Faso             pop_1952        4469979
 6 Africa    Burundi                  pop_1952        2445618
 7 Africa    Cameroon                 pop_1952        5009067
 8 Africa    Central African Republic pop_1952        1291695
 9 Africa    Chad                     pop_1952        2682462
10 Africa    Comoros                  pop_1952         153936
# … with 5,102 more rows
```

From long to tidy

The final step in recreating the gapminder data structure is to spread our observation variables out from the long format we have created.

Challenge 2

Spread the `gap_separated` data above to create a new data frame that has the same dimensions as the original `gapminder` data.

Solution to Challenge 2

```
gap_orig <- gap_separated %>%
  spread(obs_type, obs_values)

gap_orig
```

```
# A tibble: 1,704 x 6
   continent country  year gdpPercap lifeExp      pop
   <chr>     <chr>   <int>     <dbl>   <dbl>    <dbl>
 1 Africa    Algeria  1952     2449.    43.1  9279525
 2 Africa    Algeria  1957     3014.    45.7 10270856
 3 Africa    Algeria  1962     2551.    48.3 11000948
 4 Africa    Algeria  1967     3247.    51.4 12760499
 5 Africa    Algeria  1972     4183.    54.5 14760787
 6 Africa    Algeria  1977     4910.    58.0 17152804
 7 Africa    Algeria  1982     5745.    61.4 20033753
 8 Africa    Algeria  1987     5681.    65.8 23254956
 9 Africa    Algeria  1992     5023.    67.7 26298373
10 Africa    Algeria  1997     4797.    69.2 29072015
# … with 1,694 more rows
```

```
dim(gap_orig)
```

```
[1] 1704    6
```

```
dim(gapminder)
```

```
[1] 1704    6
```

We're almost there, the original was sorted by `country`, `continent`, then `year`. So finish everything off with an `arrange()` and then a `select()` to get the columns in the right order.

```
gap_orig <- gap_orig %>%
  arrange(country, continent, year) %>%
  select(country, continent, year, lifeExp, pop, gdpPercap)

gap_orig
```

```
# A tibble: 1,704 x 6
   country     continent  year lifeExp      pop gdpPercap
   <chr>       <chr>     <int>   <dbl>    <dbl>     <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.
 2 Afghanistan Asia       1957    30.3  9240934      821.
 3 Afghanistan Asia       1962    32.0 10267083      853.
 4 Afghanistan Asia       1967    34.0 11537966      836.
 5 Afghanistan Asia       1972    36.1 13079460      740.
 6 Afghanistan Asia       1977    38.4 14880372      786.
 7 Afghanistan Asia       1982    39.9 12881816      978.
 8 Afghanistan Asia       1987    40.8 13867957      852.
 9 Afghanistan Asia       1992    41.7 16317921      649.
10 Afghanistan Asia       1997    41.8 22227415      635.
# … with 1,694 more rows
```

```
gapminder
```

```
# A tibble: 1,704 x 6
   country     continent  year lifeExp      pop gdpPercap
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.
 2 Afghanistan Asia       1957    30.3  9240934      821.
 3 Afghanistan Asia       1962    32.0 10267083      853.
 4 Afghanistan Asia       1967    34.0 11537966      836.
 5 Afghanistan Asia       1972    36.1 13079460      740.
 6 Afghanistan Asia       1977    38.4 14880372      786.
 7 Afghanistan Asia       1982    39.9 12881816      978.
 8 Afghanistan Asia       1987    40.8 13867957      852.
 9 Afghanistan Asia       1992    41.7 16317921      649.
10 Afghanistan Asia       1997    41.8 22227415      635.
# … with 1,694 more rows
```

Great work! We’ve taken a messy data set and tidied it into a form that works well with all the tidyverse functions you have been using up until now.

Challenge 3

Practice your `gather` ing skills with the inbuilt `tidyr::table4a` data frame, which contains the number of TB cases recorded by the WHO in three countries in 1999 and 2000. Is the data frame untidy, and can you tidy it with `gather()` ?

```
tidyr::table4a
```

```
# A tibble: 3 x 3
  country     `1999` `2000`
* <chr>        <int>  <int>
1 Afghanistan    745   2666
2 Brazil       37737  80488
3 China       212258 213766
```

Solution to Challenge 3

```
gather(tidyr::table4a, key = year, value = TB_cases, -country)
```

```
# A tibble: 6 x 3
  country     year  TB_cases
  <chr>       <chr>    <int>
1 Afghanistan 1999       745
2 Brazil      1999     37737
3 China       1999    212258
4 Afghanistan 2000      2666
5 Brazil      2000     80488
6 China       2000    213766
```

Challenge 4

Practice your `spread` ing skills with the inbuilt `tidyr::table2` data frame, which contains the number of TB cases recorded by the WHO in three countries in 1999 and 2000 as well as their population. Is the data frame untidy, and can you tidy it with `spread()` ?

```
tidyr::table2
```

```
# A tibble: 12 x 4
   country      year type            count
   <chr>       <int> <chr>           <int>
 1 Afghanistan  1999 cases             745
 2 Afghanistan  1999 population   19987071
 3 Afghanistan  2000 cases            2666
 4 Afghanistan  2000 population   20595360
 5 Brazil       1999 cases           37737
 6 Brazil       1999 population  172006362
 7 Brazil       2000 cases           80488
 8 Brazil       2000 population  174504898
 9 China        1999 cases          212258
10 China        1999 population 1272915272
11 China        2000 cases          213766
12 China        2000 population 1280428583
```

Solution to Challenge 4

```
spread(tidyr::table2, key = type, value = count)
```

```
# A tibble: 6 x 4
  country      year  cases population
  <chr>       <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

Key Points

- Use the tidyr package to change the layout of dataframes.
- Use `gather()` to go from wide to long format.
- Use `spread()` to go from long to wide format.

Cleaning Data

Overview

Teaching: 15 min
Exercises: 30 min
Questions

- How can I process data that is untidy/inconsistent/missing parts?

Objectives

- Identify inconsistencies in data
- Appropriately fix errors in data without overwriting the original
- Reshape and manipulate data to make it tidy
- Set data to missing where necessary

Real world data sets rarely come in a form ready for analysis. Even the untidy example in the previous lesson was unusual because it was a complete data set with no missing values and informative column names. Using `gather()` / `spread()` and `separate()` / `unite()` to reshape your data will solve a lot of problems, but they can not fix all the problems you will encounter.

A few other situations you may encounter when trying to clean your data up for analysis are:

Handling of missing data

Sometimes, missing data is not recorded as blank or `NA` , but as some other value. As discussed in the data import (../17-Additional-content---Reading-Data-In/index.html) lesson, you can define the missing data values on import with the `na` argument. But in some situations it might not be possible because you are importing from a very specialised data format or using a data frame provided by someone else.

In this situation, we can recode values to be `NA` with the `na_if()` . You provide `na_if()` with a vector of values to check, and then the value that needs to be recoded as `NA` . This works well within a `mutate()` call.

```
# -1 should be NA
missing <- data_frame(id = c("a", "b", "c"), value = c(10, -1, 20))

missing
```

```
# A tibble: 3 x 2
  id    value
  <chr> <dbl>
1 a        10
2 b        -1
3 c        20
```

```
missing %>%
  mutate(value = na_if(value, -1))
```

```
# A tibble: 3 x 2
  id    value
  <chr> <dbl>
1 a        10
2 b        NA
3 c        20
```

Filling in implied values

When recording a number of measurements, repeated values may be left blank for convenience, with the implication that the first value is carried through the blank spaces. For example:

```
implied_id <- data_frame(
  id = c("a", NA, NA, "b", NA, NA),
  value = c(12, 23, 18, 34, 23, 16)
)

implied_id
```

```
# A tibble: 6 x 2
  id    value
  <chr> <dbl>
1 a        12
2 <NA>     23
3 <NA>     18
4 b        34
5 <NA>     23
6 <NA>     16
```

Here, the `id` variable is recorded in the first row, but then left blank until an observation with a new id is recorded. To complete the data frame, you will need to `fill()` the `id` column, which replaces blank entries with the last recorded value.

```
implied_id %>%
  fill(id)
```

```
# A tibble: 6 x 2
  id    value
  <chr> <dbl>
1 a        12
2 a        23
3 a        18
4 b        34
5 b        23
6 b        16
```

Dealing with outliers/errors

Your data might have values that look like they have been recorded incorrectly or are outliers. It can be appropriate to remove or alter these values in some situations. But just editing your raw data is a poor choice because it does not leave any record of what you have done.

A conservative choice in this situation is to create a new variable containing your modifications, which makes explicit what your criteria are. The `case_when()` function may be useful for this. `case_when()` is a fast and powerful way of specifying an outcome in response to a set of particular conditions. For example, assume we have the following data frame, but know that the measurement system can only record values between 0 and 30. We might decide that any measurements outside these values should be set to either 0 or 30 (Is this a good way of dealing with these errors?).

```
outliers <- data_frame(
  id = c("a", "b", "c", "d", "e"),
  value = c(15, -2, 19, 9, 35)
)

outliers
```

```
# A tibble: 5 x 2
  id    value
  <chr> <dbl>
1 a        15
2 b        -2
3 c        19
4 d         9
5 e        35
```

```
# Clamp values outside of the range 0-30 to the closest limit
outliers %>%
  mutate(clean_value = case_when(
    value < 0 ~ 0,
    value > 30 ~ 30,
    TRUE ~ value
  )
)
```

```
# A tibble: 5 x 3
  id    value clean_value
  <chr> <dbl>       <dbl>
1 a        15          15
2 b        -2           0
3 c        19          19
4 d         9           9
5 e        35          30
```

Each part of the `case_when()` call has two elements separated by a `~`. The left hand side is a *condition* that evaluates to `TRUE` or `FALSE`. The right hand side is the *output* that `case_when` should return when the condition is `TRUE`. These conditions are evaluated from top to bottom and only return the output from the first condition that is `TRUE`.

So the code above can be read as:

- Take the `outliers` data frame, then
- `mutate` it to create a new column called `clean_value`
- If the `value` column is less than zero, `clean_value` should be zero
- Otherwise, if the `value` column is greater than 30, `clean_value` should be 30
- Otherwise, if `TRUE` is `TRUE` (ie. always), `clean_value` should be the same as `value`

Key Points

- Real world data is messy
- It takes time and care to prepare it for analysis and visualisation

Writing Data

Overview

Teaching: 15 min
Exercises: 5 min
Questions

- How do I write data to disk after working in R?

Objectives

- Know when and how to write data

At some point, you will probably want to write some data out from R when you have finished your analysis. Just as with the `read_csv` function (or other `read_XXX` functions we covered in the data import (../17-Additional-content---Reading-Data-In/index.html) lesson), there are various `write_XXX` functions to write data frames out in different formats. It is usually best to write your data into a plain text format, particularly if you may need to use it again in later analysis.

What to write

Not all data needs to be written to a file when you have finished an analysis. Any data from intermediate steps in your analysis can always be recreated by running your code again, so does not need to be saved. Only keep the final results of your analysis, ie. the figures or tables you would show to other people to explain what you have found.

Where to keep it

Where your data should be stored will depend on how large it is. For small to moderately sized data, use a relative path to save it within your project's structure. This allows your entire project - data, results, and the code to create the results - to be portable and shared easily if needed. It's a good practice to keep your raw data and your modified data distinct, so make sure you set up a new folder to store your results.

Never overwrite your raw data with modified data

For large data, you will likely have a fixed location for storage. This could be a hard drive or a cloud storage server. Use an absolute path to make sure your results are being written to the correct location. But consider if saving the storage path as a variable is of benefit should you need to change the storage location in the future.

Challenge 1

Create a new folder called `processed_data` in your project folder. Write just the Australian gapminder data to a csv file in this folder. Open the created file in a text editor to confirm that it has written correctly.

Solution to Challenge 1

```
aust_data <- gapminder %>%
  filter(country == "Australia")

write_csv(aust_data, path = "processed_data/aust_gapminder.csv")
```

Key Points

- Intermediate data objects do not need to be written to disk
- Write data in an appropriate format
- Write data to the most useful location

Reproducibility

Overview

Teaching: 10 min
Exercises: 45 min
Questions

- How can I ensure my code is reproducible?

Objectives

- Understand the value of reproducibility
- Identify the key considerations to enable reproducibility

A core concept of reproducible research is that all steps from raw data to results are available and well documented so that others can make use of them. The explanation of research methods in published papers is rarely detailed enough to recreate the results, and so there is no easy way to confirm the published results, or apply the methods in a new context.

Taking the first step towards making your research more reproducible is easy when your analysis is recorded in code. Just collect your analysis code into a script and make it available with your raw data and results. Some people have an aversion to letting other people look at their code, but give it a try as there are a lot of potential benefits.

Benefits of reproducibility

- It makes it easy for your methods to be reused in the future. This may be a colleague trying to apply your approach in their own project or yourself in a year or two trying to compare newly collected data with your previous work.
- You are more likely to catch mistakes in your analysis in the process of documenting it for others. Since each step in the process is recorded, it also makes it easy to trace back to find when the mistake was introduced and correct it.
- Other people will be able to spot mistakes in your code. More people looking at your code means more opportunity to carch and correct errors, ensuring that your results are accurate.

Reproducibility with R

Besides recording your analysis in a script, there are a few other considerations to increase the reproducibility of your research.

Firstly, your script will run with the R environment that launched it. Use the RStudio Environment pane to explore your current environment after completing these lessons. Or use `ls()` to view the data in your environment, and `loadedNamespaces()` to view the packages that are loaded in the environment.

Challenge 1

Create a fresh R session (in RStudio select “Session” > “New Session”) and compare the environments. What effect might this have on a script?

To make sure that your script is self-contained, always test that it performs as expected in a fresh session.

Secondly, make sure that your analysis is documented clearly. This might mean adding short comments to your code as you write it, or for longer form documentation you can create a plain text README file in your project’s directory (eg. to explain the experimental design). An alternative approach is to use an R markdown (https://rmarkdown.rstudio.com) document, which lets you mix together text and code more easily.

When documenting your process, it is often more important to record the *decisions* that you make rather than just describing what your code is doing. Explaining *why* your analysis is taking a particular approach is far more valuable for other people to understand it.

Finally, record precisely which software versions were used to produce your results. Software can be updated and may not always produce the same results as older versions. Record the version of R you are using, as well as the versions of any packages you use. This information can be found with `sessionInfo()`

Putting it all together

As a final challenge, you will complete an end to end data analysis task in a reproducible fashion.

Create a new project and download two data files from the Bureau of Meterology, one containing meterological information (../data/BOM_data.csv), and one containing metadata about weather stations ((../data/BOM_stations.csv))

Take some time to explore the data files and understand what they contain, then write a script that answers the following questions:

- **1**: For each station, how many days have a minimum temperature, a maximum temperature and a rainfall measurement recorded?
- **2**: Which month saw the lowest difference between minimum and maximum temperatures in a day? And which state saw the highest?
- **3**: Which state had the lowest average monthly minimum temperature after excluding sites more than 500m above sea level.
- **4**: Design your own question. What is a question you had after exploring the contents of the data? Or was there something that surprised you when working with the data. Write the data frame that answers your question 4 out to a file.

Once you are finished, swap scripts with another person, along with any instructions they will need to make sure it works. Can you get their script to run?

Compare your code for the first three questions. Were there any instances where you took different approaches to solve the same problem?

Key Points

- A script is a discrete unit of analysis
- A script will be run in the context of an environment
- Software (and compute) dependencies need to be considered

Data

Overview

Teaching: 40 min
Exercises: 15 min
Questions

- What are the basic data types in R?
- How can I collect data together?
- How can I store data of different types?

Objectives

- To be aware of the different types of data.
- To understand vectors and how to create them
- To be aware of lists and how they differ from vectors

Data Types

For R to know how to deal with values, it needs to know what **type** a value is. As a way of thinking about this, imagine you are given the instruction: `3 + 4` . Fairly straightforward. Now imagine you are given the instruction: `3 + green` . This is much harder to deal with, because 3 is a number, and green is a word. For a programming language to be predictable, it needs to be able to deal with and understand values of different types.

In R, there are 5 main types: `double` , `integer` , `complex` , `logical` and `character` .

We can ask R what type a particular value (or object) is with the `typeof` function:

```
typeof(3.14)
```

```
[1] "double"
```

```
typeof(1L) # The L suffix forces the number to be an integer, since by default R uses float numbe
rs
```

```
[1] "integer"
```

```
typeof(1+1i)
```

```
[1] "complex"
```

```
typeof(TRUE)
```

```
[1] "logical"
```

```
typeof('banana')
```

```
[1] "character"
```

No matter how complicated our analyses become, all values in R is interpreted as one of these basic data types. This strictness has some really important consequences, and can be the cause of some confusing errors.

Dates

Dates and times are another special data type it's good to be aware of in R. Working with dates and times is its own detailed topic, but we'll cover them very briefly here so you're aware of some of the options.

The `lubridate` package makes working with dates and times easier (it is also hands down the best package name out there).

To get the current date or date-time:

```
today()
```

```
[1] "2019-04-29"
```

```
now()
```

```
[1] "2019-04-29 16:42:47 AEST"
```

When working with dates, you can specify the format when reading a string:

```
ymd("2019-02-13") #year month day
```

```
[1] "2019-02-13"
```

```
mdy("February 2nd, 2019") #month day year
```

```
[1] "2019-02-02"
```

```
dmy("13-Feb-2019") #day month year
```

```
[1] "2019-02-13"
```

Date-time can be created:

```
ymd_hms("2019-02-13 20:11:23") #year month day hour minute second
```

```
[1] "2019-02-13 20:11:23 UTC"
```

```
mdy_hm("02-13-2018 08:02") #month day year hour minute
```

```
[1] "2018-02-13 08:02:00 UTC"
```

Collections

So far, we've been creating and working with values in isolation (`a <- 5`). But this is very rarely how we work with data. More typically values exist in relation to other values in a group. And those groups often relate to other groups.

R provides structures for managing these groups, or collections of data. The two basic types we will work with are vectors and lists.

Vectors

A vector is a collection of values in a particular order. A critical distinguising feature of the values in a vector is that they **must** be of the same type.

We can create a vector with the `vector` function:

```
my_vector <- vector(length = 3)
my_vector
```

```
[1] FALSE FALSE FALSE
```

To emphasise, *everything in a vector must be the same basic data type*. If you don't choose the datatype, it will default to `logical` ; or, you can declare an empty vector of whatever type you like.

```
another_vector <- vector(mode='character', length=3)
another_vector
```

```
[1] "" "" ""
```

You can also make vectors with explicit contents with the combine function (`c`):

```
combine_vector <- c(2,6,3)
combine_vector
```

```
[1] 2 6 3
```

Given what we've learned so far, what do you think the following will produce?

```
quiz_vector <- c(2,6,'3')
```

This is something called *type coercion*, and it is the source of many surprises and the reason why we need to be aware of the basic data types and how R will interpret them. When R encounters a mix of types (in this case, numeric and character) to be combined into a single vector, it will force them all to be the same type. Consider:

```
coercion_vector <- c('a', TRUE)
coercion_vector
```

```
[1] "a"    "TRUE"
```

```
another_coercion_vector <- c(0, TRUE)
another_coercion_vector
```

```
[1] 0 1
```

The coercion rules go: `logical` -> `integer` -> `numeric` -> `complex` -> `character` , where -> can be read as *are transformed into*. You can try to force coercion against this flow using the `as.` functions:

```
character_vector_example <- c('0','2','4')
character_vector_example
```

```
[1] "0" "2" "4"
```

```
character_coerced_to_numeric <- as.numeric(character_vector_example)
character_coerced_to_numeric
```

```
[1] 0 2 4
```

```
numeric_coerced_to_logical <- as.logical(character_coerced_to_numeric)
numeric_coerced_to_logical
```

```
[1] FALSE  TRUE  TRUE
```

As you can see, some surprising things can happen when R forces one basic data type into another! Nitty-gritty of type coercion aside, the point is: if your data doesn't look like what you thought it was going to look like, type coercion may well be to blame

The combine function, `c()` , will also append things to an existing vector:

```
ab_vector <- c('a', 'b')
ab_vector
```

```
[1] "a" "b"
```

```
combine_example <- c(ab_vector, 'SWC')
combine_example
```

```
[1] "a"   "b"   "SWC"
```

You can also make series of numbers:

```
my_series <- 1:10
my_series
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(10)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
seq(1,10, by=0.1)
```

```
 [1]  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0  2.1  2.2  2.3
[15]  2.4  2.5  2.6  2.7  2.8  2.9  3.0  3.1  3.2  3.3  3.4  3.5  3.6  3.7
[29]  3.8  3.9  4.0  4.1  4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9  5.0  5.1
[43]  5.2  5.3  5.4  5.5  5.6  5.7  5.8  5.9  6.0  6.1  6.2  6.3  6.4  6.5
[57]  6.6  6.7  6.8  6.9  7.0  7.1  7.2  7.3  7.4  7.5  7.6  7.7  7.8  7.9
[71]  8.0  8.1  8.2  8.3  8.4  8.5  8.6  8.7  8.8  8.9  9.0  9.1  9.2  9.3
[85]  9.4  9.5  9.6  9.7  9.8  9.9 10.0
```

We can ask a few questions about vectors:

```
sequence_example <- seq(10)
head(sequence_example, n=2)
```

```
[1] 1 2
```

```
tail(sequence_example, n=4)
```

```
[1]  7  8  9 10
```

```
length(sequence_example)
```

```
[1] 10
```

```
class(sequence_example)
```

```
[1] "integer"
```

```
typeof(sequence_example)
```

```
[1] "integer"
```

Finally, you can give names to elements in your vector:

```
my_example <- 5:8
names(my_example) <- c("a", "b", "c", "d")
my_example
```

```
a b c d
5 6 7 8
```

```
names(my_example)
```

```
[1] "a" "b" "c" "d"
```

Challenge 1

Start by making a vector with the numbers 1 through 26. Multiply the vector by 2, and give the resulting vector names A through Z (hint: there is a built in vector called `LETTERS`)

Solution to Challenge 1

```
x <- 1:26
x <- x * 2
names(x) <- LETTERS
```

Lists

Another basic data of grouping values is the `list` . A list is simpler in some ways than the other types, because you can put anything you want in it:

```
list_example <- list(1, "a", TRUE, 1+4i)
list_example
```

```
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

```
another_list <- list(title = "Numbers", numbers = 1:10, data = TRUE )
another_list
```

```
$title
[1] "Numbers"

$numbers
 [1]  1  2  3  4  5  6  7  8  9 10

$data
[1] TRUE
```

Lists can even contain other lists:

```
nested_list <- list(list_example, another_list)
nested_list
```

```
[[1]]
[[1]][[1]]
[1] 1

[[1]][[2]]
[1] "a"

[[1]][[3]]
[1] TRUE

[[1]][[4]]
[1] 1+4i


[[2]]
[[2]]$title
[1] "Numbers"

[[2]]$numbers
 [1]  1  2  3  4  5  6  7  8  9 10

[[2]]$data
[1] TRUE
```

There is no limit to how deeply nested such structures can be.

Because they are so flexible, lists are incredibly powerful, but can be a bit difficult to work with depending on how complex their structure is.

By combining the strictness of vectors with the flexibility of lists, soon we'll see the workhorse of R, the `data.frame` .

Challenge 2

Make a list that contains:

- Today's date
- A character vector of length two containing your name and your favourite colour
- Another list containing the integer 5

Solution to Challenge 2

```
solution_list <- list(today(), c("My name", "Puce"),
                        list(5))
solution_list
```

```
[[1]]
[1] "2019-04-29"

[[2]]
[1] "My name" "Puce"

[[3]]
[[3]][[1]]
[1] 5
```

Key Points

- The basic data types in R are double, integer, complex, logical, and character.
- Vectors are an ordered collection of data of the same type.
- Create vectors with `c()` .
- Lists are an ordered collection of data that can be any type.

Reading Data In

Overview

Teaching: 35 min
Exercises: 10 min
Questions

- How can I get data into R?

Objectives

- Read data in from plain text and Excel files
- Control import parameters to cope with unusual data formats

So far, we have been using datasets that are already available within R. Chances are though that the data you want to work with exists separately, perhaps as a spreadsheet or CSV file. In order to start working with this data, we need to learn how to read it in to R.

Common file types

There are many different ways that data could be stored, but for now we will focus on reading in tabular data. This is data that is already in a data frame (../05-Dataframes/index.html)-like structure and so we will want to read it in to R as a data frame.

Two of the most common ways you might find data like this are:

Plain text files

These are text files where the columns of data are separated by some delimiting character. Examples of these might include

- CSV (**c**omma **s**eparated **v**alue) files that use a comma to delimit the columns

```
carat,cut,color,clarity,depth,table,price,x,y,z
0.23,Ideal,E,SI2,61.5,55,326,3.95,3.98,2.43
0.21,Premium,E,SI1,59.8,61,326,3.89,3.84,2.31
0.23,Good,E,VS1,56.9,65,327,4.05,4.07,2.31
0.29,Premium,I,VS2,62.4,58,334,4.2,4.23,2.63
0.31,Good,J,SI2,63.3,58,335,4.34,4.35,2.75
```

- TSV (**t**ab **s**eparated **v**alue) files, which use a tab instead

```
carat   cut     color   clarity depth   table   price   x       y       z
0.23    Ideal   E       SI2     61.5    55      326     3.95    3.98    2.43
0.21    Premium E       SI1     59.8    61      326     3.89    3.84    2.31
0.23    Good    E       VS1     56.9    65      327     4.05    4.07    2.31
0.29    Premium I       VS2     62.4    58      334     4.2     4.23    2.63
0.31    Good    J       SI2     63.3    58      335     4.34    4.35    2.75
```

Spreadsheet files

All spreadsheet programs (Excel, Calc, Numbers, etc.) will have a way to export data into one of the plain text formats above. This will usually be the best way to get data into R. For excel files (.xls or .xlsx) however, it is possible to read them in directly which we will demonstrate later

Gapminder data

For this section, we will be reading in data from the Gapminder (www.gapminder.org) organisation, which records various statistics for 142 countries betwen 1952 and 2007. This data is available as an R package (https://cran.r-project.org/web/packages/gapminder/index.html), but we have prepared csv (../data/gapminder.csv), tsv (../data/gapminder.tsv), and excel (../data/gapminder.xlsx) versions for you to practice with.

Challenge 1

Download the three versions of the Gapminder above and save them to your project folder.

Open one of the files and describe what statistics are recorded

Solution to Challenge 1

Using the ideas discussed previously about project structure (../01-R-and-RStudio/index.html), we will save the files into a `data` directory within our project. We can then access them with a relative path `data/gapminder.csv` (for the csv example)

Opening a file we can see that there are six columns of data: a country name and continent, the year that the data was recorded, and the life expectancy, population and GDP per capita.

To load this data into R, we will use the `read_csv` function from the `readr` (http://readr.tidyverse.org) package (which will be loaded automatically if you have preciously run `library(tidyverse)` , but here we will load it separately).

```
library(readr)

gapminder_csv <- read_csv("data/gapminder.csv")
```

```
Parsed with column specification:
cols(
  country = col_character(),
  continent = col_character(),
  year = col_double(),
  lifeExp = col_double(),
  pop = col_double(),
  gdpPercap = col_double()
)
```

```
gapminder_csv
```

```
# A tibble: 1,704 x 6
   country     continent  year lifeExp      pop gdpPercap
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.
 2 Afghanistan Asia       1957    30.3  9240934      821.
 3 Afghanistan Asia       1962    32.0 10267083      853.
 4 Afghanistan Asia       1967    34.0 11537966      836.
 5 Afghanistan Asia       1972    36.1 13079460      740.
 6 Afghanistan Asia       1977    38.4 14880372      786.
 7 Afghanistan Asia       1982    39.9 12881816      978.
 8 Afghanistan Asia       1987    40.8 13867957      852.
 9 Afghanistan Asia       1992    41.7 16317921      649.
10 Afghanistan Asia       1997    41.8 22227415      635.
# … with 1,694 more rows
```

Here, we can see that `read_csv` provides us with some information on what it thinks the data types are as it reads the file in (two columns of character data and four columns of double in this case). We can also see that it has imported the data in the tibble format (../05-Dataframes/index.html) discussed previously.

Similarly, for the tsv file there is a `read_tsv` function

Challenge 2

Read in the `gapminder.tsv` file using `read_tsv`

Confirm that reading from csv or tsv files produce the same output (you might find the `all.equal` function useful)

Solution to Challenge 2

```
gapminder_tsv <- read_tsv("data/gapminder.tsv")

gapminder_tsv
```

```
# A tibble: 1,704 x 6
   country     continent  year lifeExp      pop gdpPercap
   <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>
 1 Afghanistan Asia       1952    28.8  8425333      779.
 2 Afghanistan Asia       1957    30.3  9240934      821.
 3 Afghanistan Asia       1962    32.0 10267083      853.
 4 Afghanistan Asia       1967    34.0 11537966      836.
 5 Afghanistan Asia       1972    36.1 13079460      740.
 6 Afghanistan Asia       1977    38.4 14880372      786.
 7 Afghanistan Asia       1982    39.9 12881816      978.
 8 Afghanistan Asia       1987    40.8 13867957      852.
 9 Afghanistan Asia       1992    41.7 16317921      649.
10 Afghanistan Asia       1997    41.8 22227415      635.
# … with 1,694 more rows
```

```
all.equal(gapminder_csv, gapminder_tsv)
```

```
[1] TRUE
```

Control of import parameters

Both `read_csv` and `read_tsv` have a number of parameters that may be needed to help import a file. You can read more about these in the help file for the functions, but some more commonly encountered situations are:

Ignoring comments or metadata from the top of a file

Sometimes files will have metadata information included along with the data. Use `skip` to ignore a set number of lines from the top of the file or `comment` if metadata lines are indicated by a specific character such as `#`.

Eg. a file such as:

```
#Metadata information detailing data collection process
#Not part of the data, but included to explain it
country continent       year    lifeExp pop     gdpPercap
Afghanistan     Asia    1952    28.801  8425333 779.4453145
Afghanistan     Asia    1957    30.332  9240934 820.8530296
Afghanistan     Asia    1962    31.997  10267083        853.10071
```

Can be read in using:

```
read_tsv("data/commented_file.tsv", comment = "#")
read_tsv("data/commented_file.tsv", skip = 2)
```

Both lines above produce the same output

Providing column names

By default, `readr` assumes that the first row of data is a "header" - that is defines the names of the columnes for the data frame. If this is not the case you can either specify the names manually with a character vector, or have the column names generated automatically

Eg.

```
Afghanistan     Asia    1952    28.801  8425333 779.4453145
Afghanistan     Asia    1957    30.332  9240934 820.8530296
Afghanistan     Asia    1962    31.997  10267083        853.10071
```

Can be read in using:

```
read_tsv("data/no_names.tsv", col_names = F)
```

```
# A tibble: 3 x 6
   X1          X2       X3    X4       X5    X6
   <chr>       <chr> <dbl> <dbl>    <dbl> <dbl>
 1 Afghanistan Asia   1952  28.8  8425333  779.
 2 Afghanistan Asia   1957  30.3  9240934  821.
 3 Afghanistan Asia   1962  32.0 10267083  853.
```

or

```
read_tsv("data/no_names.tsv", col_names = c("country", "continent","year","lifeExp","pop","gdpPer
cap"))
```

```
# A tibble: 3 x 6
  country     continent  year lifeExp      pop gdpPercap
  <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>
1 Afghanistan Asia       1952    28.8  8425333      779.
2 Afghanistan Asia       1957    30.3  9240934      821.
3 Afghanistan Asia       1962    32.0 10267083      853.
```

Dealing with missing data

By default, blank columns or ones containing “NA” are considered missing. If your data uses a different value for missing data you will need to specify it.

Eg.

```
country continent       year    lifeExp pop     gdpPercap
Afghanistan     Asia    1952    28.801  8425333 779.4453145
Afghanistan     Asia    1957    30.332  -       820.8530296
Afghanistan     Asia    1962    31.997  10267083        853.10071
```

```
read_tsv("data/missing.tsv")
```

```
# A tibble: 3 x 6
  country     continent  year lifeExp pop      gdpPercap
  <chr>       <chr>     <dbl>   <dbl> <chr>        <dbl>
1 Afghanistan Asia       1952    28.8 8425333       779.
2 Afghanistan Asia       1957    30.3 -             821.
3 Afghanistan Asia       1962    32.0 10267083      853.
```

Using the default parameters, `pop` is incorrectly read in as a character column because missing data is indicated by `-`

```
read_tsv("data/missing.tsv", na = "-")
```

```
# A tibble: 3 x 6
  country     continent  year lifeExp      pop gdpPercap
  <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>
1 Afghanistan Asia       1952    28.8  8425333      779.
2 Afghanistan Asia       1957    30.3       NA      821.
3 Afghanistan Asia       1962    32.0 10267083      853.
```

Now the `pop` data is correctly recognised as a number, and the missing data is correctly recorded as `NA` .

Setting explicit column types

Sometimes, the default column types might not be what you intend. In that case you can set them explicitly by either providing a string representing the column types

> From the `read_tsv` help file: c = character, i = integer, n = number, d = double, l = logical, f = factor, D = date, T = date time, t = time, ? = guess, or _/- to skip the column

Eg. setting `country` to be a factor, `pop` as an integer, and not reading in the `lifeExp` column. The remaining columns are left as ‘guess’, in which case `readr` tries to determine their type automatically.

```
read_tsv("data/gapminder.tsv", col_types = "f??-i?")
```

```
# A tibble: 1,704 x 5
   country     continent  year      pop gdpPercap
   <fct>       <chr>     <dbl>    <int>     <dbl>
 1 Afghanistan Asia       1952  8425333      779.
 2 Afghanistan Asia       1957  9240934      821.
 3 Afghanistan Asia       1962 10267083      853.
 4 Afghanistan Asia       1967 11537966      836.
 5 Afghanistan Asia       1972 13079460      740.
 6 Afghanistan Asia       1977 14880372      786.
 7 Afghanistan Asia       1982 12881816      978.
 8 Afghanistan Asia       1987 13867957      852.
 9 Afghanistan Asia       1992 16317921      649.
10 Afghanistan Asia       1997 22227415      635.
# … with 1,694 more rows
```

Note:

If you choose to use a type string like this, you must provide a type for each colum in the data or R will throw a warning

Or by providing a column specification, like how `read_csv` did to explain it’s default parsing when we first ran it

Importing the same types as above in this format would look like this:

```
# Default is col_guess()
# So we just need to specify the
# columns that are different
gapminder_spec <- cols(
  country = col_factor(),
  pop = col_integer(),
  gdpPercap = col_skip()
)

read_tsv("data/gapminder.tsv", col_types = gapminder_spec)
```

```
# A tibble: 1,704 x 5
   country     continent  year lifeExp      pop
   <fct>       <chr>     <dbl>   <dbl>    <int>
 1 Afghanistan Asia       1952    28.8  8425333
 2 Afghanistan Asia       1957    30.3  9240934
 3 Afghanistan Asia       1962    32.0 10267083
 4 Afghanistan Asia       1967    34.0 11537966
 5 Afghanistan Asia       1972    36.1 13079460
 6 Afghanistan Asia       1977    38.4 14880372
 7 Afghanistan Asia       1982    39.9 12881816
 8 Afghanistan Asia       1987    40.8 13867957
 9 Afghanistan Asia       1992    41.7 16317921
10 Afghanistan Asia       1997    41.8 22227415
# … with 1,694 more rows
```

Knowing how these few parameters work will enable you to read most csv/tsv files you come across into R.

Challenge 4

Read in the gapminder tsv again, but make the `continent` column a factor and the `year` column a date. (Hint: Use the `cols()` specification and "%Y" is the date format you will need)

Check whether the newly imported data is still the same as the csv data imported previously

Solution to Challenge 4

```
spec <- cols(
  continent = col_factor(),
  year = col_date(format = "%Y")
)

gapminder_tsv <- read_tsv("data/gapminder.tsv", col_types = spec)

all.equal(gapminder_csv, gapminder_tsv)
```

```
[1] "Incompatible type for column `country`: x character, y factor" "Incompatible type for colu
mn `year`: x numeric, y Date"
```

Reading excel files

For reading in excel files, we will use the `readxl` (https://readxl.tidyverse.org) package. This package is part of the tidyverse, but is not loaded with `library(tidyverse)` , so we will have to do it ourselves.

```
library(readxl)
```

We can now use the `read_excel` function from `readxl` to read in our data.

```
gapminder_excel <- read_excel("data/gapminder.xlsx")
```

Since `read_excel` uses most of the same options as `read_csv` / `read_tsv` covered above, you already know how to use it. Some excel specific options you may need to use include

Specifying the worksheet to extract

The gapminder data is all in the first sheet in the file (called 'gapminder'). This means that

```
read_excel("data/gapminder.xlsx")
read_excel("data/gapminder.xlsx", sheet = 1)
read_excel("data/gapminder.xlsx", sheet = "gapminder")
```

will all produce the same result.

Extracting data from a specific range

Perhaps you are only interested in data from a particular region of the worksheet. This can be imported using an excel range pattern.

```
read_excel("data/gapminder.xlsx", range = "A1:E4")
```

```
# A tibble: 3 x 5
  country     continent  year lifeExp      pop
  <chr>       <chr>     <dbl>   <dbl>    <dbl>
1 Afghanistan Asia       1952    28.8  8425333
2 Afghanistan Asia       1957    30.3  9240934
3 Afghanistan Asia       1962    32.0 10267083
```

`read_csv` and `read.csv`

You may have noticed that there is also a `read.csv` function. This default behaviour of this function is that all strings are read in as factors, which can be a common source of mistakes for newer R users. For this reason, we will stick with using the tidyverse equivalents where possible.

Regardless of the method chosen. After importing your data, it's always a good choice to check it afterwards using the data frame (../05-Dataframes/index.html) exploration methods

Key Points

- Use `read_csv()` or `read_tsv()` to read in plain text data
- Use `read_excel()` from the `readxl` package to read in Excel files