EARL

ENTERPRISE APPLICATIONS OF THE R LANGUAGE

Functional Programming with Purr

Workshop 5 at the EARL Conference

Date: 11 September 2018
Time: 2pm – 5pm
Room: Tower 3

@ earl-team@mango-solutions.com
+44 (0)1249 705 450
mango-solutions.com

Chapter 1 Introduction to Functional Programming with Purrr

1.1 Introduction to the Training

1.1.1 Course Aims

This course has been designed to provide a practical introduction to functional programming in R. It is assumed that you will have either attended an Introduction to R course or have equivalent experience with the R language.

1.1.2 Course Materials

Items appearing in this material are sometimes given a special appearance to set them apart from regular text. Here's how they look:

```
> This is a section of code                    # This is a comment
```

🔴	A warning, typically describing non-intuitive aspects of the R language
🟠	A tip: additional features of R or "shortcuts" based on user experience
🟢	Exercises to be performed during (or after) the training course

1.1.3 Course Script and Exercise Answers

A great deal of code will be executed within R during the delivery of this training. This includes the answers to each exercise, as well as other code written to answer questions that arise. Following the course, you will be sent a script containing all the code that was executed.

1.3 Recap: Lists

In the world of the tidyverse many R users no longer realise that they are working with lists or what benefit a list can bring. But, they are actually used very widely, for everything form model output, to plot objects to data frames themselves.

A list can be thought of as a container in which to hold other objects. For example, we could have a list that holds 3 character vectors, a numeric matrix and a logical array. Lists provide an efficient way to hold these objects so that they are easily accessible to the user.

```
> myList <- list(A = rnorm(100),
+                B = sample(LETTERS, 10))
```

The two most useful functions for extracting information from a list are `names` and `length`.

```
> length(myList)
[1] 2
> names(myList)
[1] "A" "B"
```

If we want to get information out of a list we can do so in one of two ways. We can either use double square brackets, with the index of the element we want to extract. Or we can use the dollar notation with the name of the element.

```
> myList[[2]]
 [1] "P" "A" "M" "K" "E" "S" "Q" "O" "H" "Z"
> myList$B
 [1] "P" "A" "M" "K" "E" "S" "Q" "O" "H" "Z"
```

A “Data Frame” is simply a list structure with the following additional “rules”:
- Every list element must have a name
- All elements (vectors) must be of the same length

1.4 Recap: Basic Function Writing

Whilst we will see some shortcuts to providing functions that are provided by **purrr** and ideas of functional programming it is still worth being familiar with how we contruct a function.

We define a function with the `function` keyword. This is followed by the arguments to that function and the body of the function – or what we want the function to do with those arguments.

```
> addingFunction <- function(x, y = 0){
+ 
+   x + y
+ 
+ }
> 
> addingFunction(1:10)
 [1]  1  2  3  4  5  6  7  8  9 10
```

Note that:

- Defaults are defined with the arguments (`y = 0`)
- The return value for the function should go on the last line
- Any objects created inside the function will not exist outside of the function after it has been called

1. Using the `gap_split` data in the **repurrrsive** package:
    a. How many elements are in the list?
    b. Do the elements have names?
    c. Extract the data from the United Kingdom. What type of data is it?
2. Write a function that, when given the data and a country name will calculate the mean life expectancy for that country

1.5 What is Functional Programming?

In a nutshell, functional programming puts an emphasis on using functions to describe what we are doing so that we can simplify the code that we write and more clearly understand what is happening. The best way to understand how it changes the way we write code and why it helps us is to consider an example.

Suppose we are analysing the gapminder dataset that we saw above. As part of our analysis we want to find the year in which each country had its maximum life expectancy. We might do this for the first country as:

```
> gap_split[[1]] %>%
+   filter(lifeExp == max(lifeExp)) %>%
+   magrittr::extract2("year")
[1] 2007
```

To obtain this for all countries we would then need to repeat this calculation 141 more times. Copying and pasting this code would almost certainly result in a mistake at some point. Suppose we then wanted to find the year of the maximum population, we would need to edit our code 142 times to switch from life expectancy to population.

Instead, we can write a function that takes our data and performs the calculation for us.

```
> maxYear <- function(data){
+
+  data %>%
+    filter(lifeExp == max(lifeExp)) %>%
+    magrittr::extract2("year")
+
+ }
>
> maxYear(gap_split[[1]])
[1] 2007
```

We could make this function more generic, taking the column that we want to find the maximum of, but this requires the use of techniques for programming in **dplyr** which is beyond the scope of this course.

This makes it very easy for us to now change our function to generate the year that a country had its maximum population but doesn't resolve the problem of potentially making mistakes in the countries. To resolve this, we could use a loop.

```
> years <- vector("numeric", length = length(gap_split))
>
> for(c in seq_along(gap_split)){
+   years[c] <- maxYear(gap_split[[c]])
+ }
```

However, functional programming gives us a much more elegant way to solve this problem. We can use what is known as a functional. This is a special type of function that allows us to provide another function as an argument, and effectively tell it to run the that function on all subsets of the data. For example:

```
> map(gap_split, maxYear)
```

Throughout this course we will see a variety of the features of functional programming and see what this really means for us, and how we can apply those ideas using the **purrr** package.

1.6 Further Reading

For more examples of using the **purrr** package and how it can help us with iteration look at the "Iteration" chapter of R for Data Science by Hadley Wickham and Garrett Grolemund.

For more technical details on functional programming you may be interested in the Functional Programming section of Advanced R Programming.

For a great overview of the core functions in the **purrr** package we would recommend downloading the cheat sheet from the RStudio website.

Chapter 2
Iteration and the purrr package

2.1 Introduction

Whilst there are a number of approaches to allow us to iteratively apply a function to lists in R, the current state of the art is the **purrr** package. This package provides a consistent means of calling the iteration functions that also fits with the wider tidyverse packages.

We will also be using the **repurrrsive** package for some example datasets that have been pre-structured to a form similar to how you may work with data with the **purrr** package.

```
> library(purrr)
> library(repurrrsive)
```

2.2 Iterating over each list element

The core function in the **purrr** package to allow us to iteratively apply some other functionality is the `map` function. This function allows us to pass it a list, tell it to perform some action and return to us a new list with the output of applying the function multiple times.

2.2.1 Extracting Named Elements

As an example, let's suppose we want to extract the life expectancy data for each country from the split gapminder data.

```
> lifeExpectancy <- map(gap_split, "lifeExp")
> length(lifeExpectancy)
[1] 142
> lifeExpectancy[[1]]
[1] 28.801 30.332 31.997 34.020 36.088 38.438 39.854 ...
> lifeExpectancy$`United Kingdom`
 [1] 69.180 70.420 70.760 71.360 72.010 72.760 74.040...
```

In this example we have simply given the map function the name of the column that we want to extract from each data frame and it has returned a new list. The new list has 1 element for each country and this element contains a vector of the life expectancy data.

This is just one of the shortcuts that the map function allows for applying a function. As well as extracting elements by name we can also extract by position/index. Try running the code above but with `4` instead of `"lifeExp"`

2.2.2 Applying Functions

Generally, we will want more flexibility than simply extracting elements from data. Instead of passing a column name we can give a function name to apply to each list element. In the context of functional programming, this makes `map` a functional.

```
> map(lifeExpectancy, max)
$`Afghanistan`
[1] 43.828

$Albania
[1] 76.423

$Algeria
[1] 72.301
...
```

Again, this returns a list. This time the list contains only a single value as the `max` function returns only a single value.

1. Using the split gapminder data:
   a. Find the minimum value of the population for each country
   b. Calculate the variance of the GDP per capita

Extension Questions

2. For each country, extract the value of the population in 1952.
3. Which country had the lowest population in 1952? *(hint: take a look at `which.min`)*

2.2.3 Passing Additional Arguments

Sometimes we will want to pass additional arguments to our functions, for example if there are missing values in our data we may want to use the `na.rm` argument when finding the mean or maximum. We can pass additional arguments by simply naming them after the function name in our `map` call.

```
> map(lifeExpectancy, quantile, probs = c(0.05, 0.95))
$`Afghanistan`
      5%      95%
29.64305 42.89355

$Albania
     5%     95%
57.4575 75.9984
...
```

2.3 Applying Custom Functions

In the previous two sections we were able to calculate the maximum life expectancy for each country, but we had to run the map function twice. Once to extract the correct column and then again to apply the function.

We can simplify this by writing our own function to calculate the maximum life expectancy.

```
> maxLife <- function(data){
>   max(data$lifeExp)
> }
> map(gap_split, maxLife)
$`Afghanistan`
[1] 43.828

$Albania
[1] 76.423
...
```

Whilst this is reasonable, and especially in the case that the function is long, easy to follow, the **purrr** package does allow us to take some shortcuts. We can define a function in **purrr** using a special formula notation.

```
> map(gap_split, ~max(.$lifeExp))
$`Afghanistan`
[1] 43.828

$Albania
[1] 76.423
...
```

This notation looks a little strange at first, but it is a convenient way of not having to define a complete function for simple cases. At the front we use the usual formula notation "~". Inside our simplified function definition, we use the shortcut "`.`" to refer to the list element.

In functional programming, an unnamed function such as this, is known as an anonymous function. The ability to work with unnamed functions is one of the characteristics of functional programming.

1. Write a function that will:
   a. Take a data frame as input
   b. Return the year in which the lowest population value occurred
   c. Returns the year as a single integer value
2. Run this function on the split gapminder data to find the year that each country had its lowest population.
3. Re-write this code to use the **purrr** shortcuts

Extension

4. Which country had its lowest population most recently?

2.4 Simplifying Output

In the examples that we have seen so far, the map function has returned a list, even though for each country we have only returned a single value, that may be more easily interpreted by being a vector of numeric values.

As well as the generic map function there are a collection of functions that make up the map family that allow us to return specific output types.

Function	Return Type
`map`	list
`map_chr`	character vector
`map_dbl`	numeric vector
`map_df`	data frame
`map_int`	integer vector
`map_lgl`	logical vector

```
> map_dbl(gap_split, ~max(.$lifeExp))
Afghanistan                  Albania                  Algeria
     43.828                   76.423                   72.301
     Angola                Argentina                Australia
     42.731                   75.320                   81.235
...
```

A useful feature of these functions is that if the output of the function cannot be represented in the type requested, because it can't be converted to that type, or it is too long an output type, the function will simply error.

```
> map_lgl(gap_split, ~max(.$lifeExp))
Error: Can't coerce element 1 from a double to a logical
> map_dbl(gap_split, ~range(.$lifeExp))
Error: Result 1 is not a length 1 atomic vector
```

This ensures that if something goes wrong with our code, we can be more certain that it won't simply convert to another format, like a list.

●

1. Find the average life expectancy for each country, storing the output in a numeric vector
2. Can you store the output in an integer vector?

Chapter 3
Working with Lists

3.1 Introduction

The **purrr** package not only allows us to apply functions to list elements but it also includes a range of functions to allow us to easily work with lists in ways that we may a data frame. For example, allowing us to filter, transform and join to lists.

```
> library(purrr)
> library(repurrrsive)
```

3.2 Filtering List Elements

So far, we have worked with all the data in our list, but suppose we were only interested in a subset of all the available data. To extract a single list element we can use the `pluck` function. We can use this to extract by name or index and by providing multiple names or postions.

```
> pluck(gap_split, "United Kingdom")
# A tibble: 12 x 6
   country        continent  year lifeExp      pop gdpPercap
   <fct>          <fct>     <int>   <dbl>    <int>     <dbl>
 1 United Kingdom Europe     1952    69.2 50430000     9980.
 2 United Kingdom Europe     1957    70.4 51430000    11283.
 3 United Kingdom Europe     1962    70.8 53292000    12477.
 4 United Kingdom Europe     1967    71.4 54959000    14143.
 5 United Kingdom Europe     1972    72.0 56079000    15895.
 6 United Kingdom Europe     1977    72.8 56179000    17429.
 7 United Kingdom Europe     1982    74.0 56339704    18232.
 8 United Kingdom Europe     1987    75.0 56981620    21665.
 9 United Kingdom Europe     1992    76.4 57866349    22705.
10 United Kingdom Europe     1997    77.2 58808266    26075.
11 United Kingdom Europe     2002    78.5 59912431    29479.
12 United Kingdom Europe     2007    79.4 60776238    33203.
>
> pluck(gap_split, "United Kingdom", "lifeExp")
 [1] 69.180 70.420 70.760 71.360 72.010 72.760 74.040 75.007 76.420
77.218 78.471 79.425
```

This can be useful to focus in on a small section of the data, but what if we wanted to collect all the elements that satisfied some criteria (the equivalent of **dplyr**'s `filter`). For this we can use either `keep` or `discard`.

Both functions allow you to create a logical test, just like in `filter`. The `keep` function will then retain elements that satisfy the criteria, while `discard` will remove elements that satisfy the criteria.

```
> is.europe <- function(data){
+   unique(data$continent) == "Europe"
+ }
>
> europe <- keep(gap_split, is.europe)
> names(europe)
 [1] "Albania"                "Austria"                "Belgium"
 [4] "Bosnia and Herzegovina" "Bulgaria"               "Croatia"
...
>
> otherContinent <- discard(gap_split, is.europe)
> names(otherContinent)
  [1] "Afghanistan"              "Algeria"
  [3] "Angola"                   "Argentina"
  [5] "Australia"                "Bahrain"
...
```

3.3 Joining Lists

Just as with data in a rectangular, data frame structure, we will often have a need to join to a list. The **purrr** package contains some useful functions that will allow us not only to add elements to an existing list, but also specify where in the list they will be added.

The two functions we can use are `append`, for adding after an existing element, and `prepend`, for adding before an existing element.

```
> uk <- pluck(gap_split, "United Kingdom")
> updated_gap <- prepend(gap_split, values = list("UK" = uk))
> names(updated_gap)
[1] "UK"                         "Afghanistan"
[3] "Albania"                    "Algeria"
[5] "Angola"                     "Argentina"
...
```

3.4 Transposing Lists

At times it can be more convenient to work with a transposed version of the list rather than the original version. The `transpose` function lets us invert a list. In the case of the gapminder data this will give us list elements for each column of the data (country, life expectancy, population etc.), each containing list elements for each country.

```
> invert <- transpose(gap_split)
> names(invert)
[1] "country"   "continent" "year"      "lifeExp"   "pop"
"gdpPercap"
> pluck(invert, "lifeExp", "United Kingdom")
 [1] 69.180 70.420 70.760 71.360 72.010 ...
```

1. Write a function to test if the life expentancy for the most recent year is the maximum life expectancy. The function should return TRUE (when life expectancy in 2007 is the maximum) or FALSE.
2. Test your function on the data for Botswana and the data for Denmark.
3. Filter the split gapminder data to return only elements where the life expectancy in 2007 is not it's highest life expectancy.

Extension Questions

4. Use appropriate `map` functions to return the maximum life expectancy for each of these countries and their life expectancy in 2007.

Chapter 4
The Wider Map Family

4.1 Introduction

Once you get into the habit of working with lists and applying functions across them with the map family of functions you will quickly realise you have a need to apply across two or more lists at the same time or for functions that don't specifically produce an output. We can use a wider family of functions included in **purrr** to do this.

```
> library(purrr)
> library(repurrrsive)
> library(tidyr)
```

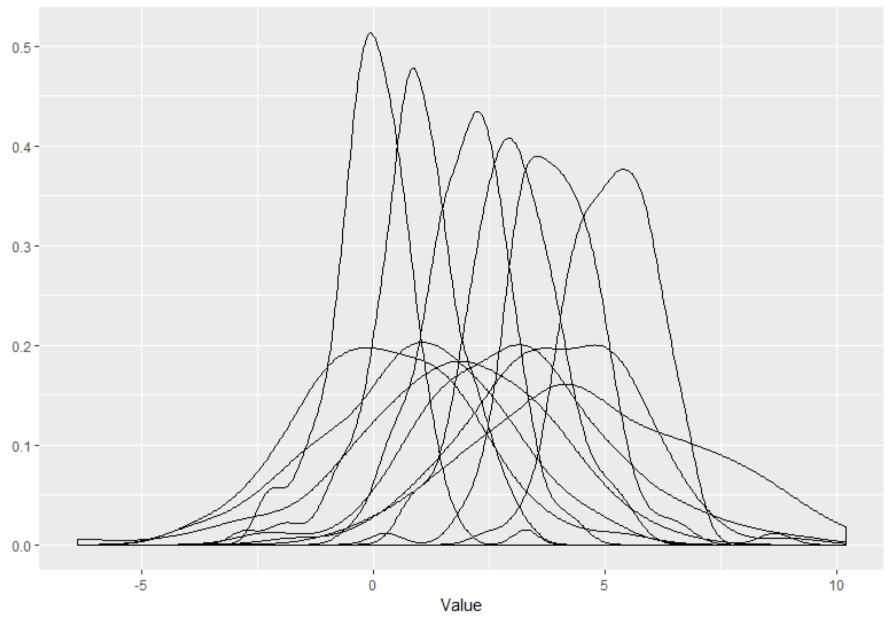
4.2 Applying across Multiple Lists

When it comes to using two or more lists as the inputs for our functions, there are two families of functions for doing this in **purrr**.

Function	Usage
`map2`	Apply a function across 2 list
`pmap`	Apply a function across p lists

Both of these functions have the same range of output options as the `map` function, defined in the same way, e.g. `map2_chr`.

```
> means <- rep(0:5, each = 2)
> sds <- rep(c(1, 2), times = 6)
> means <- set_names(means, nm = LETTERS[1:12])
>
> normData <- map2_df(means, sds, rnorm, n = 100)
>
> gather(normData, Simulation, Value) %>%
+   qplot(Value, data = ., geom = "density", group = Simulation)
```



Until now we have only used lists as the input to `map` functions but we can also provide vectors as we have in this example.

One thing to note when we are applying functions to multiple lists is how we specify each of the lists when we define our own functions using the **purrr** shotcuts. For the `map2` function we can simply use "`.x`" and "`.y`"

```
> map2_df(means, sds, ~rnorm(n = 100, mean = .x, sd = .y))
```

For `pmap` we use "`..p`", where p is the number of the list element.

```
> n <- sample(c(5, 10, 100), 12, replace = TRUE)
>
> pmap(list(means, sds, n), ~rnorm(n = ..3, mean = ..1, sd = ..2))
```

4.3 Side Effects

All of the examples we have considered so far have been applying functions that will return the result of some calculation, such as the maximum. But we may want to use a list to generate other types of output such as graphics. When the output we are interested in is not a value that is returned but some other action, like creating a graphic, printing to the screen or saving files, we consider this to be a side effect.

If we are interested in side effects we use the `walk` functions rather than `map`. Just like `map` there are variants of `walk` for applying over two (`walk2`) or more (`pwalk`) lists. However, as we don't use `walk` for its return values, there are no equivalents to the `map_*` functions.

```
> plotLifeExpectancy <- function(data){
+    country <- unique(data$country)
+    p <- qplot(x = year, y = lifeExp, data = data,
+               main = country, geom = "line")
+    print(p)
+ }
>
> pdf("LifeExpectancyPlots.pdf")
> walk(gap_split, plotLifeExpectancy)
> dev.off()
```

●

1. Write a function that:
    a. Takes a vector of life expectancies for a single country and the name of the country
    b. Prints to the screen the name of the country and the maximum life expectancy (e.g. *"The maximum for United Kingdom is …"*)
2. Create:
    a. a list containing only life expectancy values for all countries in the gapminder data
    b. a list of the country names in the gapminder data
3. Apply your function over the list of life expectancies and countries

4.4 Using the Index

In the examples above we have carefully contructed our problem so that we were able to extract the country name to use in our output, either as a title to a plot or as part of the printed output. We have done this in two ways above, one making use of the fact that the country was repeated in the data and once passing a second list. But this can be quite inconvenient, or impractical. Thankfully there are shortcuts to this in **purrr**, through a series of functions `imap` and `iwalk`.

```
> plotLifeExpectancy <- function(data, country){
+   p <- qplot(x = year, y = lifeExp, data = data,
+              main = country, geom = "line")
+   print(p)
+ }

> pdf("LifeExpectancyPlotsWithCountry.pdf")
> iwalk(gap_split, plotLifeExpectancy)
> dev.off()
```

●

1. Write a function that:
    a. Takes a vector of life expectancies for a single country and the name of the country
    b. Prints to the screen the name of the country and the maximum life expectancy (e.g. *"The maximum for United Kingdom is …"*)
2. Apply your function over the list of life expectancies using the iwalk function.

Chapter 5
Nested Data

5.1 Introduction

There are many instances when we are working with the `map` family of functions that it is to apply functions across sub-groups of data-frames. For example, in the gapminder data we are generally applying functions for each country, where the data for each country is a data frame. Rather than convert our data frame into a list we can instead maintain the data frame structure using nested data frames.

```
> library(tidyverse)
> library(repurrrsive)
> library(modelr)
> library(broom)
```

5.2 Nested Data Frames

A nested data frame is one in which we store data frames within data frames. As an example consider the nested version of the gapminder data frame in the **repurrrsive** package.

```
> gap_nested
# A tibble: 142 x 3
   country     continent data
   <fct>       <fct>     <list>
 1 Afghanistan Asia      <tibble [12 x 4]>
 2 Albania     Europe    <tibble [12 x 4]>
 3 Algeria     Africa    <tibble [12 x 4]>
 4 Angola      Africa    <tibble [12 x 4]>
 5 Argentina   Americas  <tibble [12 x 4]>
 6 Australia   Oceania   <tibble [12 x 4]>
 7 Austria     Europe    <tibble [12 x 4]>
 8 Bahrain     Asia      <tibble [12 x 4]>
 9 Bangladesh  Asia      <tibble [12 x 4]>
10 Belgium     Europe    <tibble [12 x 4]>
# ... with 132 more rows
```

Here you can see that we have only one row for each country. The remaining data for each country is stored in the data column. Rather than containing individual values, this column contains a series of data frames, or specifically `tibbles`.

We can interact with this data frame in the usual way, and use the `unnest` function to extract the data stored in a particular cell.

```
> gap_nested %>%
+   filter(country == "United Kingdom") %>%
+   select(data) %>%
+   unnest()
# A tibble: 12 x 4
    year lifeExp      pop gdpPercap
   <int>   <dbl>    <int>     <dbl>
 1  1952    69.2 50430000     9980.
 2  1957    70.4 51430000    11283.
 3  1962    70.8 53292000    12477.
 4  1967    71.4 54959000    14143.
 5  1972    72.0 56079000    15895.
 6  1977    72.8 56179000    17429.
```

5.3 Mutate and Map

Whilst much of what we have seen so far can still be used when we are working with nested data, the main point to remember is that we will now, generally, want to map over the column that contains the data frames.

```
> map(gap_nested$data, "lifeExp")
[[1]]
 [1] 28.801 30.332 31.997 34.020 ...
```

For this reason, and so that we can keep all of the results with the corresponding rows of the data frame, we typically work with map in combination with mutate.

```
> gap_nested %>%
+   mutate(MaxLife = map_dbl(data, ~max(.$lifeExp)))
# A tibble: 142 x 4
   country     continent data              MaxLife
   <fct>       <fct>     <list>              <dbl>
 1 Afghanistan Asia      <tibble [12 x 4]>    43.8
 2 Albania     Europe    <tibble [12 x 4]>    76.4
 3 Algeria     Africa    <tibble [12 x 4]>    72.3
 4 Angola      Africa    <tibble [12 x 4]>    42.7
 5 Argentina   Americas  <tibble [12 x 4]>    75.3
 6 Australia   Oceania   <tibble [12 x 4]>    81.2
```

●

Note the use of `map_dbl` in the example above. This ensures that a numeric vector is returned so we see the output as numeric values. Using `map` would instead return a list.

●

1. Using the nested gapminder data:
   a. Find the minimum value of the population for each country
   b. Calculate the variance of the GDP per capita

Extension Questions

2. For each country, extract the value of the population in 1952.
3. Which country had the lowest population in 1952?

5.4 Map for Modelling & Simulation

One of the great uses for the `map` family of functions and nested data is in modelling and simulation. In particular in any situation where there is a need to run a model on multiple sets of data, be that subsets of a datasets (like countries in the gapminder data) or bootstrap samples.

```
> gap_model <- gap_nested %>%
+   mutate(model = map(data, ~lm(lifeExp ~ year, data = .x)))
> gap_model
# A tibble: 142 x 4
   country     continent data              model
   <fct>       <fct>     <list>            <list>
 1 Afghanistan Asia      <tibble [12 x 4]> <S3: lm>
 2 Albania     Europe    <tibble [12 x 4]> <S3: lm>
 3 Algeria     Africa    <tibble [12 x 4]> <S3: lm>
 4 Angola      Africa    <tibble [12 x 4]> <S3: lm>
 5 Argentina   Americas  <tibble [12 x 4]> <S3: lm>
 6 Australia   Oceania   <tibble [12 x 4]> <S3: lm>
 7 Austria     Europe    <tibble [12 x 4]> <S3: lm>
 8 Bahrain     Asia      <tibble [12 x 4]> <S3: lm>
 9 Bangladesh  Asia      <tibble [12 x 4]> <S3: lm>
10 Belgium     Europe    <tibble [12 x 4]> <S3: lm>
# ... with 132 more rows
```

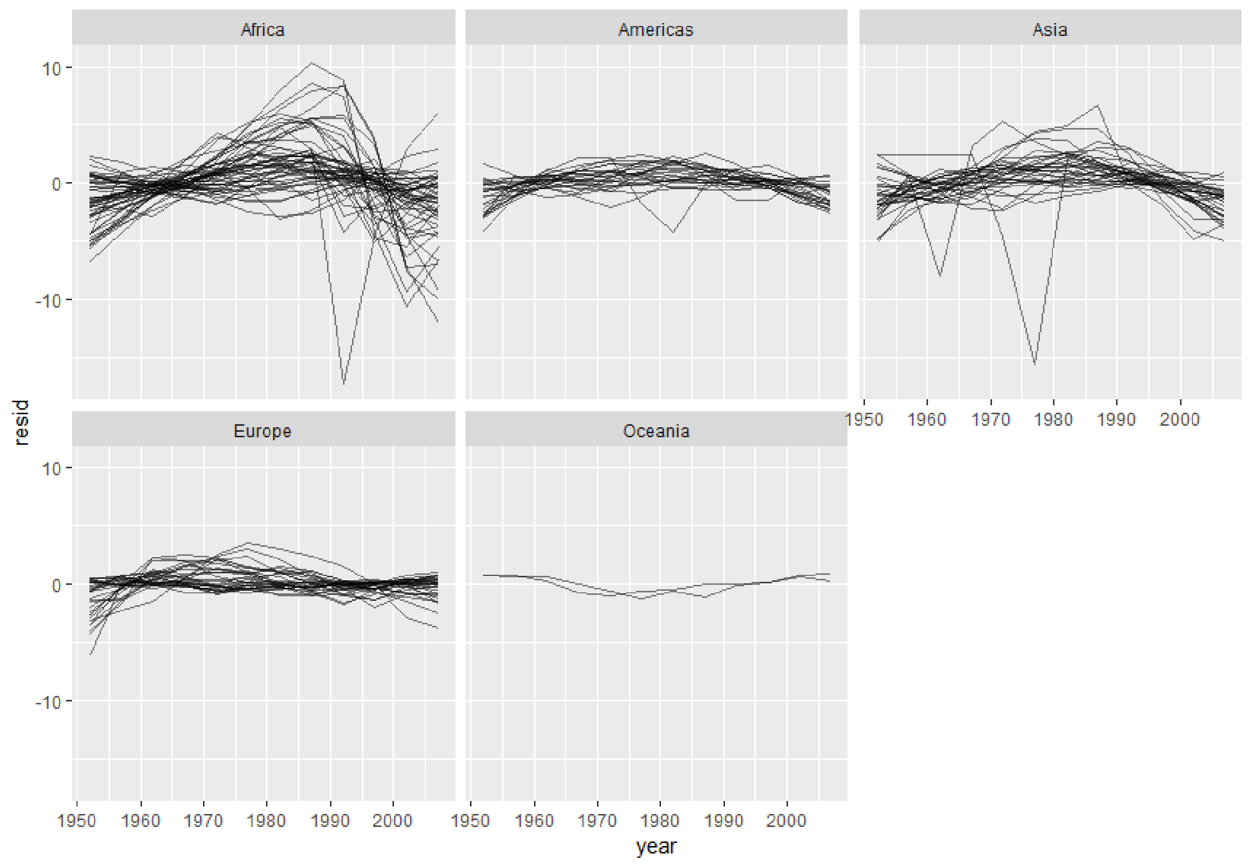
We can see that this returns an updated nested data frame, where the model column now contains the entire linear model fit for the given country. By continuing to make use of the `map` functions we can extract information about the model.

```
> gap_model %>%
+   transmute(country, fit = map(model, glance)) %>%
+   unnest()
# A tibble: 142 x 12
   country r.squared adj.r.squared sigma statistic  p.value    df
   <fct>       <dbl>         <dbl> <dbl>     <dbl>    <dbl> <int>
 1 Afghan~     0.948         0.942 1.22      181.  9.84e- 8     2
 2 Albania     0.911         0.902 1.98      102.  1.46e- 6     2
 3 Algeria     0.985         0.984 1.32      662.  1.81e-10     2
 4 Angola      0.888         0.877 1.41       79.1 4.59e- 6     2
 5 Argent~     0.996         0.995 0.292    2246.  4.22e-13     2
 6 Austra~     0.980         0.978 0.621     481.  8.67e-10     2
 7 Austria     0.992         0.991 0.407    1261.  7.44e-12     2
 8 Bahrain     0.967         0.963 1.64      291.  1.02e- 8     2
 9 Bangla~     0.989         0.988 0.977     930.  3.37e-11     2
10 Belgium     0.995         0.994 0.293    1822.  1.20e-12     2
# ... with 132 more rows, and 5 more variables: logLik <dbl>,
#   AIC <dbl>, BIC <dbl>, deviance <dbl>, df.residual <int>
```

● The packages **modelr** and **broom** provide a series of useful functions for generating tidy model output including metrics, coefficients, residuals and predictions.

Once we have the information that we want we can use `unnest`, as we did above, to get the data back into a format that we can easily use for other tasks, such as visualisation.

```
> gap_fit <- gap_model %>%
+   mutate(residuals = map2(data, model, add_residuals)) %>%
+   unnest(residuals)
>
> ggplot(data = gap_fit, aes(year, resid)) +
+   geom_line(alpha = 0.5, aes(group = country)) +
+   facet_wrap(~continent)
```



5.5 Converting to Nested Data

To create nested data we need to use the `nest` function, from the **tidyr** package. We can use this function alone, but it is generally clearer to see how we are nesting when used in combination with `group_by`.

```
> iris %>%
+   group_by(Species) %>%
+   nest()
# A tibble: 3 x 2
  Species    data
  <fct>      <list>
1 setosa     <tibble [50 x 4]>
2 versicolor <tibble [50 x 4]>
3 virginica  <tibble [50 x 4]>
```

We can group by multiple variables by simply passing more variables to `group_by`, and all remaining columns will be nested.

1. Starting with the `gap_simple` data, convert to a nested data frame, grouping by continent.
2. For each continent, fit a single linear model for life expectancy
3. Add a column containing the model metrics to the nested data