EARL conference

Functional Programming with Purrr.

WORKSHOP 4.

Chapter 1

Introduction To Functional Programming With purrr

Introduction to the Workshop

Workshop 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.

Workshop Materials

Items appearing in this material are sometimes given a special appearance to set them apart from regular text. Here is 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

Workshop 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.

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 from model output, to plot objects to data frames themselves.

A list can be thought of as a container in which objects of different types are held. 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.

```
my_list <- list(
  A = rnorm(100),
  B = sample(LETTERS, 10))
```

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

```
names(my_list)
```

```
#> [1] "A" "B"
```

```
length(my_list)
```

```
#> [1] 2
```

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.

```
my_list[[2]]
```

```
#>  [1] "I" "J" "H" "L" "G" "R" "A" "B" "K" "D"
```

```
my_list$B
```

```
#>  [1] "I" "J" "H" "L" "G" "R" "A" "B" "K" "D"
```

The dollar notation `my_list$B` is a shorthand for element extraction. A hidden pitfall is that the name matching is partial, i.e. the above command might return `my_list$BA` if it existed but `my_list$B` did not – this would not be desirable if an error should have been caught and dealt with. For this reason, it is not recommended to use the dollar notation in a non-interactive environment (e.g. in production).

```
my_list <- list(
  A = rnorm(100),
  BA = sample(LETTERS, 10))

my_list$B
```

```
#>  [1] "W" "B" "P" "D" "N" "G" "J" "E" "X" "A"
```

Recap: Dataframes

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

Recap: Function Writing

Just a quick reminder of how a function is constructed, although you will be introduced to more ways of doing so e.g. a shorthand in purrr, later.

We define a function with the function keyword. This is followed by () and between them the arguments to that function (if needed), and the body of the function – or what we want the function to do.

```
adding_function <- function(x, y = 0){
  x + y
}

my_results <- adding_function(1:10)
```

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.

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.

We'll use the gapminder dataset (made famous by Hans Rosling) from the repurrrsive package to illustrate functional programming. Let's take a quick look

```
library(dplyr)
library(repurrrsive)

repurrrsive::gap_split |> head()
```

```
#> $Afghanistan
#> # A tibble: 12 × 6
#>    country     continent  year lifeExp      pop gdpPercap
#>    <fct>       <fct>     <int>   <dbl>    <int>     <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.
#> 11 Afghanistan Asia       2002    42.1 25268405      727.
#> 12 Afghanistan Asia       2007    43.8 31889923      975.
#>
#> $Albania
#> # A tibble: 12 × 6
#>    country continent  year lifeExp     pop gdpPercap
#>    <fct>   <fct>     <int>   <dbl>   <int>     <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.
#> 11 Albania Europe     2002    75.7 3508512     4604.
#> 12 Albania Europe     2007    76.4 3600523     5937.
#>
#> $Algeria
#> # A tibble: 12 × 6
#>    country continent  year lifeExp      pop gdpPercap
#>    <fct>   <fct>     <int>   <dbl>    <int>     <dbl>
#>  1 Algeria Africa     1952    43.1  9279525     2449.
#>  2 Algeria Africa     1957    45.7 10270856     3014.
#>  3 Algeria Africa     1962    48.3 11000948     2551.
#>  4 Algeria Africa     1967    51.4 12760499     3247.
#>  5 Algeria Africa     1972    54.5 14760787     4183.
#>  6 Algeria Africa     1977    58.0 17152804     4910.
#>  7 Algeria Africa     1982    61.4 20033753     5745.
#>  8 Algeria Africa     1987    65.8 23254956     5681.
#>  9 Algeria Africa     1992    67.7 26298373     5023.
```

```
#> 10 Algeria Africa     1997    69.2 29072015     4797.
#> 11 Algeria Africa     2002    71.0 31287142     5288.
#> 12 Algeria Africa     2007    72.3 33333216     6223.
#>
#> $Angola
#> # A tibble: 12 × 6
#>    country continent  year lifeExp      pop gdpPercap
#>    <fct>   <fct>     <int>   <dbl>    <int>     <dbl>
#>  1 Angola  Africa     1952    30.0  4232095     3521.
#>  2 Angola  Africa     1957    32.0  4561361     3828.
#>  3 Angola  Africa     1962    34    4826015     4269.
#>  4 Angola  Africa     1967    36.0  5247469     5523.
#>  5 Angola  Africa     1972    37.9  5894858     5473.
#>  6 Angola  Africa     1977    39.5  6162675     3009.
#>  7 Angola  Africa     1982    39.9  7016384     2757.
#>  8 Angola  Africa     1987    39.9  7874230     2430.
#>  9 Angola  Africa     1992    40.6  8735988     2628.
#> 10 Angola  Africa     1997    41.0  9875024     2277.
#> 11 Angola  Africa     2002    41.0 10866106     2773.
#> 12 Angola  Africa     2007    42.7 12420476     4797.
#>
#> $Argentina
#> # A tibble: 12 × 6
#>    country   continent  year lifeExp      pop gdpPercap
#>    <fct>     <fct>     <int>   <dbl>    <int>     <dbl>
#>  1 Argentina Americas   1952    62.5 17876956     5911.
#>  2 Argentina Americas   1957    64.4 19610538     6857.
#>  3 Argentina Americas   1962    65.1 21283783     7133.
#>  4 Argentina Americas   1967    65.6 22934225     8053.
#>  5 Argentina Americas   1972    67.1 24779799     9443.
#>  6 Argentina Americas   1977    68.5 26983828    10079.
#>  7 Argentina Americas   1982    69.9 29341374     8998.
#>  8 Argentina Americas   1987    70.8 31620918     9140.
#>  9 Argentina Americas   1992    71.9 33958947     9308.
#> 10 Argentina Americas   1997    73.3 36203463    10967.
#> 11 Argentina Americas   2002    74.3 38331121     8798.
#> 12 Argentina Americas   2007    75.3 40301927    12779.
#>
#> $Australia
#> # A tibble: 12 × 6
#>    country   continent  year lifeExp      pop gdpPercap
#>    <fct>     <fct>     <int>   <dbl>    <int>     <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.
```

This gap_split dataset is actually a list of datasets, one for each country, the individual datasets contain some key figures for each year.

If 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)) |>
  pull("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, not to mention the impracticality. 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.

```
max_year <- function(data){
  data |>
    filter(lifeExp == max(lifeExp)) |>
    pull("year")
}

max_year(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 in which 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] <- max_year(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 function on all subsets of the data. For example:

```
library(purrr)
map(gap_split, max_year) |> head()
```

```
#> $Afghanistan
#> [1] 2007
#>
#> $Albania
#> [1] 2007
#>
#> $Algeria
#> [1] 2007
#>
#> $Angola
#> [1] 2007
#>
#> $Argentina
#> [1] 2007
#>
#> $Australia
#> [1] 2007
```

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.

Functional vs object-orientated: two main programming paradigms, based on the way in which problems are described and solved. Functional programming is generally considered more natural for R. The are two aspects:

1. R has many first-class functions built-in, such as all arithmetic operators (try `-(3, 2)`).
2. Utilities such as purrr have made it intuitive and convenient to write “functional style” code in R which we will explore further.

Chapter 2

Iteration And The purrr Package

Introduction

The core functions in the purrr package are the “map family” of functions. They transform an input by applying a function to each element of a list or atomic vector and returning an object of the same length as the input.

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

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.

```
life_expectancy <- map(gap_split, "lifeExp")
length(life_expectancy)
```

```
#> [1] 142
```

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

```
#>  [1] 28.801 30.332 31.997 34.020 36.088 38.438 39.854 40.822 41.674
#> [10] 41.763 42.129 43.828
```

```
life_expectancy$`United Kingdom`
```

```
#>  [1] 69.180 70.420 70.760 71.360 72.010 72.760 74.040 75.007 76.420
#> [10] 77.218 78.471 79.425
```

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”.

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(life_expectancy, max) |> head()
```

```
#> $Afghanistan
#> [1] 43.828
#>
#> $Albania
#> [1] 76.423
#>
#> $Algeria
#> [1] 72.301
#>
#> $Angola
#> [1] 42.731
#>
#> $Argentina
#> [1] 75.32
#>
#> $Australia
#> [1] 81.235
```

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)

Demonstrated by the below two lines, Hadley Wickham describes one of the benefits of functional programming with purrr as it "weights actions and objects equally":

```
map(life_expectancy, max)
```

```
map(life_expectancy, min)
```

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(life_expectancy, quantile, probs = c(0.05, 0.95)) |> head()
```

```
#> $Afghanistan
#>       5%      95% 
#> 29.64305 42.89355 
#> 
#> $Albania
#>      5%     95% 
#> 57.4575 75.9984 
#> 
#> $Algeria
#>       5%      95% 
#> 44.51140 71.58215 
#> 
#> $Angola
#>      5%     95% 
#> 31.1062 41.7806 
#> 
#> $Argentina
#>      5%     95% 
#> 63.5377 74.7810 
#> 
#> $Australia
#>       5%      95% 
#> 69.78550 80.75925
```

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 like we did earlier.

```
max_life <- function(data){
  max(data$lifeExp)
}
map(gap_split, max_life)|> head()
```

```
#> $Afghanistan
#> [1] 43.828
#>
#> $Albania
#> [1] 76.423
#>
#> $Algeria
#> [1] 72.301
#>
#> $Angola
#> [1] 42.731
#>
#> $Argentina
#> [1] 75.32
#>
#> $Australia
#> [1] 81.235
```

Whilst this is reasonable – and especially in the case that the function is short, the purrr package does allow us to take some shortcuts. We can define a function in purrr explicitly

```
map(gap_split, function(x) { max(x$lifeExp)}) |> head()
```

```
#> $Afghanistan
#> [1] 43.828
#>
#> $Albania
#> [1] 76.423
#>
#> $Algeria
#> [1] 72.301
#>
#> $Angola
#> [1] 42.731
#>
#> $Argentina
#> [1] 75.32
#>
#> $Australia
#> [1] 81.235
```

or use a special formula notation to pass arguments to an existing formula.

```
map(gap_split, ~ max(.$lifeExp)) |> head()
```

```
#> $Afghanistan
#> [1] 43.828
#>
#> $Albania
#> [1] 76.423
#>
#> $Algeria
#> [1] 72.301
#>
#> $Angola
#> [1] 42.731
#>
#> $Argentina
#> [1] 75.32
#>
#> $Australia
#> [1] 81.235
```

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?

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 – it may be more useful to return them in a vector instead.

There is a collection of functions that make up the map family and 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)) |> head()
```

```
#> Afghanistan     Albania     Algeria      Angola   Argentina
#>      43.828      76.423      72.301      42.731      75.320
#>   Australia
#>      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 result in 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 in `stop_bad_type()`:
#> ! Result 1 must be a single double, not a double vector of length 2
```

This reduces the chances of unexpected output, e.g. if anything other than logical values would indicate issues in previous code it would be caught and dealt with properly, instead of masked inside 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

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)
```

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 positions.

```
pluck(gap_split, "United Kingdom")
```

```
#> # A tibble: 12 × 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
#> [10] 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 dplyr::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) |> head()
```

```
#> [1] "Albania"                  "Austria"
#> [3] "Belgium"                  "Bosnia and Herzegovina"
#> [5] "Bulgaria"                 "Croatia"
```

```
other_continent <- discard(gap_split, is_europe)
names(other_continent) |> head()
```

```
#> [1] "Afghanistan" "Algeria"     "Angola"      "Argentina"
#> [5] "Australia"   "Bahrain"
```

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")
# prepend has a before argument that can be used to specify
# which element to put before, default is first
# append has similar after argument
updated_gap <- prepend(gap_split, values = list("UK" = uk))
names(updated_gap) |> head()
```

```
#> [1] "UK"          "Afghanistan" "Albania"     "Algeria"
#> [5] "Angola"      "Argentina"
```

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"
#> [6] "gdpPercap"
```

```
pluck(invert, "lifeExp", "United Kingdom")
```

```
#>  [1] 69.180 70.420 70.760 71.360 72.010 72.760 74.040 75.007 76.420
#> [10] 77.218 78.471 79.425
```

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.

https://r4ds.had.co.nz/vectors.html#lists-of-condiments

Chapter 4

The Wider Map Family

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)
library(ggplot2)
```

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 lists
`pmap`	Apply a function across any number of lists ("p" for parallel)

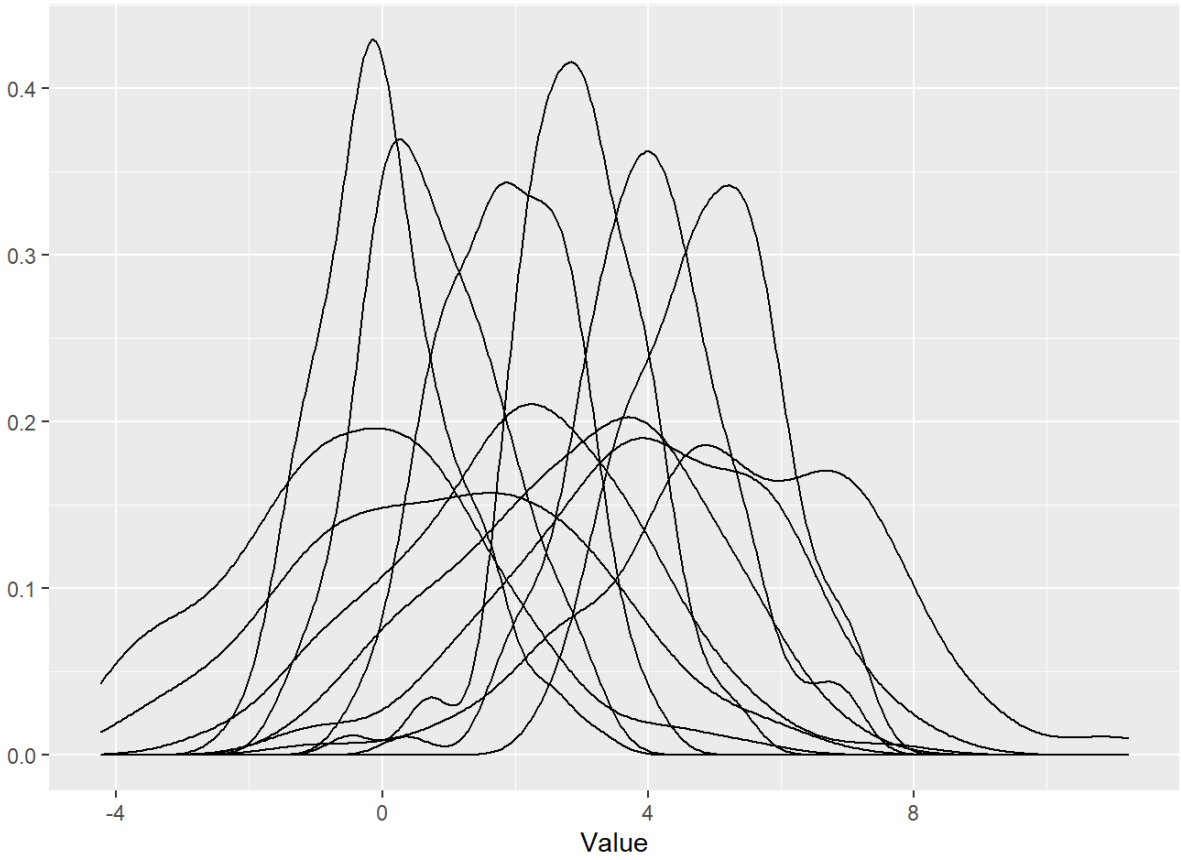
Both of these functions have the same range of output options as the map function, defined in the same way, e.g. map2_chr.

```
means <- rep(0:5, each = 2)
sds <- rep(c(1, 2), times = 6)
means <- set_names(means, nm = LETTERS[1:12])


norm_data <- map2_df(means, sds, rnorm, n = 100)

plot_data <- norm_data  |>
  pivot_longer(
    col = everything(),
    names_to = "Simulation",
    values_to = "Value")

qplot(
    Value,
    data = plot_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))
```

```
#> # A tibble: 100 × 12
#>          A       B       C        D     E     F     G     H     I
#>      <dbl>   <dbl>   <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#>  1 -0.580  -3.68    1.75    3.38    2.18  1.82   2.44  3.44  3.62
#>  2 -0.269   1.20    1.65    2.18    1.12  2.53   4.12  2.03  4.58
#>  3  0.236   0.0192  3.72   -0.557   1.86  3.06   2.88  1.57  4.50
#>  4  0.718  -3.45    1.07   -1.51    1.71  5.53   3.19  7.13  5.12
#>  5  0.0154 -0.846   0.444   0.0814  1.52  1.28   2.51  3.33  2.93
#>  6 -1.61   -1.82   -0.105   0.929   0.969 1.45   2.68  2.78  3.88
#>  7 -0.710  -0.129  -0.0124 -0.0115  1.56  1.98   2.45  5.84  4.51
#>  8  1.64   -0.514   1.34   -0.284   2.05  0.335  3.94  5.37  2.91
#>  9  1.47    1.51    1.08    1.72    2.26  2.39   4.21  4.62  4.87
#> 10  0.926  -6.78    0.736   0.00921 0.970 4.90   3.43  2.11  3.43
#> # … with 90 more rows, and 3 more variables: J <dbl>, K <dbl>,
#> #   L <dbl>
```

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

```
n <- sample(c(5, 10, 20), 12, replace = TRUE)
pmap(list(means, sds, n), ~ rnorm(n = ..3, mean = ..1, sd = ..2))
```

```
#> $A
#>  [1]  0.05306677  1.53943355 -0.37342093  0.78738827  0.03692600
#>  [6] -0.78153767  0.75647820 -1.29213744  0.05891420  1.69704274
#> [11] -0.29830208 -0.32368495 -0.96490302  1.78830119 -0.34392395
#> [16] -0.32899193 -0.45053399 -1.00139455  0.51045692  0.96668043
#>
#> $B
#>  [1] -0.09667314 -1.68887259  5.90863261  2.96781438 -1.54789835
#>  [6] -1.67487147  1.44818175  0.35473912 -0.94812044  0.17037707
#>
#> $C
#>  [1] -0.1133114  0.3607876  3.0855347  0.9049196  1.1766309  0.5386486
#>  [7]  1.9660442  1.2380121  0.8375268 -0.2847679
#>
#> $D
#> [1]  4.3636074  0.4878532 -0.1896942 -2.5406924 -0.4308187
#>
#> $E
#> [1] 2.883182 2.087774 1.766396 3.157384 1.732936
#>
#> $F
#> [1]  1.2426897 -0.1755008  1.2357810 -0.2875697  1.8032261
#>
#> $G
#>  [1] 2.709382 2.556148 3.591702 2.388103 2.774929 3.623110 2.019475
#>  [8] 2.728481 2.638644 1.632811
#>
#> $H
#>  [1] 4.444356 5.361765 5.754598 5.179543 2.293605 6.072980 7.346117
#>  [8] 1.464136 6.633322 5.871938
#>
#> $I
#> [1] 4.356636 2.476217 4.655750 3.336176 4.097865
#>
#> $J
#>  [1] 5.9615715 3.7462881 4.6025603 0.1366557 5.8245424 5.0612313
#>  [7] 6.7198297 4.0106801 3.1902272 2.1303632
#>
#> $K
#>  [1] 4.642229 4.766969 5.654868 6.908149 4.992976 5.652107 4.466870
#>  [8] 5.170035 5.677779 3.977813 4.970971 3.586710 5.237613 5.821791
#> [15] 7.829777 4.154803 5.508486 6.113233 4.970607 5.402652
#>
#> $L
#>  [1] 2.505774 5.563631 5.856329 6.299537 4.348480 2.730328 2.572354
#>  [8] 4.746437 1.903174 5.110017
```

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.

```
plot_life_expectancy <- 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, plot_life_expectancy)
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
```

Using The Index

In the examples above we have carefully constructed 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, once 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.The essential equivalence is `imap(x, f())` is the same as `map2(x, names(x), f())`.

```
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

Introduction

There are many instances when we are working with the map family of functions that we aim 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(dplyr)
library(tidyr)
library(purrr)
library(ggplot2)
library(repurrrsive)
library(modelr)
library(broom)
```

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.

```
#observe the nested version of the gapminder data
gap_nested |> head()
```

```
#> # A tibble: 6 × 3
#>   country     continent data
#>   <fct>       <fct>     <list>
#> 1 Afghanistan Asia      <tibble [12 × 4]>
#> 2 Albania     Europe    <tibble [12 × 4]>
#> 3 Algeria     Africa    <tibble [12 × 4]>
#> 4 Angola      Africa    <tibble [12 × 4]>
#> 5 Argentina   Americas  <tibble [12 × 4]>
#> 6 Australia   Oceania   <tibble [12 × 4]>
```

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(cols = c(data))
```

```
#> # A tibble: 12 × 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.
#>  7  1982    74.0 56339704    18232.
#>  8  1987    75.0 56981620    21665.
#>  9  1992    76.4 57866349    22705.
#> 10  1997    77.2 58808266    26075.
#> 11  2002    78.5 59912431    29479.
#> 12  2007    79.4 60776238    33203.
```

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") |> head()
```

```
#> [[1]]
#>  [1] 28.801 30.332 31.997 34.020 36.088 38.438 39.854 40.822 41.674
#> [10] 41.763 42.129 43.828
#>
#> [[2]]
#>  [1] 55.230 59.280 64.820 66.220 67.690 68.930 70.420 72.000 71.581
#> [10] 72.950 75.651 76.423
#>
#> [[3]]
#>  [1] 43.077 45.685 48.303 51.407 54.518 58.014 61.368 65.799 67.744
#> [10] 69.152 70.994 72.301
#>
#> [[4]]
#>  [1] 30.015 31.999 34.000 35.985 37.928 39.483 39.942 39.906 40.647
#> [10] 40.963 41.003 42.731
#>
#> [[5]]
#>  [1] 62.485 64.399 65.142 65.634 67.065 68.481 69.942 70.774 71.868
#> [10] 73.275 74.340 75.320
#>
#> [[6]]
#>  [1] 69.120 70.330 70.930 71.100 71.930 73.490 74.740 76.320 77.560
#> [10] 78.830 80.370 81.235
```

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(max_life = map_dbl(data, ~ max(.$lifeExp))) |>
  head()
```

```
#> # A tibble: 6 × 4
#>   country     continent data              max_life
#>   <fct>       <fct>     <list>               <dbl>
#> 1 Afghanistan Asia      <tibble [12 × 4]>     43.8
#> 2 Albania     Europe    <tibble [12 × 4]>     76.4
#> 3 Algeria     Africa    <tibble [12 × 4]>     72.3
#> 4 Angola      Africa    <tibble [12 × 4]>     42.7
#> 5 Argentina   Americas  <tibble [12 × 4]>     75.3
#> 6 Australia   Oceania   <tibble [12 × 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?

Map For Modelling And 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 |> head()
```

```
#> # A tibble: 6 × 4
#>   country     continent data              model
#>   <fct>       <fct>     <list>            <list>
#> 1 Afghanistan Asia      <tibble [12 × 4]> <lm>
#> 2 Albania     Europe    <tibble [12 × 4]> <lm>
#> 3 Algeria     Africa    <tibble [12 × 4]> <lm>
#> 4 Angola      Africa    <tibble [12 × 4]> <lm>
#> 5 Argentina   Americas  <tibble [12 × 4]> <lm>
#> 6 Australia   Oceania   <tibble [12 × 4]> <lm>
```

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(cols = c(fit)) |>
  head()
```

```
#> # A tibble: 6 × 13
#>   country     r.squared adj.r.squared sigma statistic  p.value    df
#>   <fct>           <dbl>         <dbl> <dbl>     <dbl>    <dbl> <dbl>
#> 1 Afghanistan     0.948         0.942 1.22      181.  9.84e- 8     1
#> 2 Albania         0.911         0.902 1.98      102.  1.46e- 6     1
#> 3 Algeria         0.985         0.984 1.32      662.  1.81e-10     1
#> 4 Angola          0.888         0.877 1.41       79.1 4.59e- 6     1
#> 5 Argentina       0.996         0.995 0.292    2246.  4.22e-13     1
#> 6 Australia       0.980         0.978 0.621     481.  8.67e-10     1
#> # … with 6 more variables: logLik <dbl>, AIC <dbl>, BIC <dbl>,
#> #   deviance <dbl>, df.residual <int>, nobs <int>
```

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.

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

```
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)
```

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 × 2
#> # Groups:   Species [3]
#>   Species    data
#>   <fct>      <list>
#> 1 setosa     <tibble [50 × 4]>
#> 2 versicolor <tibble [50 × 4]>
#> 3 virginica  <tibble [50 × 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

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. https://r4ds.had.co.nz/iteration.html

For more technical details on functional programming you may be interested in the Functional Programming section of Advanced R Programming. https://adv-r.hadley.nz/fp.html

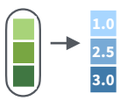
For a great overview of the core functions in the purrr package we would recommend downloading the cheat sheet from the RStudio website. https://github.com/rstudio/cheatsheets/blob/main/purrr.pdf

Apply functions with purrr : : **CHEAT SHEET**

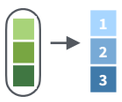
Map Functions

ONE LIST

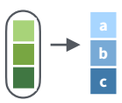
map(.x, .f, …**)** Apply a function to each element of a list or vector, return a list.
x <- list(1:10, 11:20, 21:30)
l1 <- list(x = c("a", "b"), y = c("c", "d"))
map(l1, sort, decreasing = TRUE)



map_dbl(.x, .f, …**)**
Return a double vector.
map_dbl(x, mean)

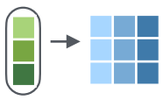


map_int(.x, .f, ...**)**
Return an integer vector.
map_int(x, length)

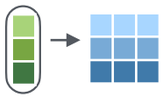


map_chr(.x, .f, …**)**
Return a character vector.
map_chr(l1, paste, collapse = "")

map_lgl(.x, .f, …**)**
Return a logical vector.
map_lgl(x, is.integer)



map_dfc(.x, .f, ...**)**
Return a data frame created by column-binding.
map_dfc(l1, rep, 3)



map_dfr(.x, .f, ..., .id = NULL**)**
Return a data frame created by row-binding.
map_dfr(x, summary)

walk(.x, .f, ...**)** Trigger side effects, return invisibly.
walk(x, print)

TWO LISTS

map2(.x, .y, .f, …**)** Apply a function to pairs of elements from two lists or vectors, return a list.
y <- list(1, 2, 3); z <- list(4, 5, 6); l2 <- list(x = "a", y = "z")
map2(x, y, ~ .x * .y)

map2_dbl(.x, .y, .f, …**)**
Return a double vector.
map2_dbl(y, z, ~ .x / .y)

map2_int(.x, .y, .f, …**)**
Return an integer vector.
map2_int(y, z, `+`)

map2_chr(.x, .y, .f, …**)**
Return a character vector.
map2_chr(l1, l2, paste, collapse = ",", sep = ":")

map2_lgl(.x, .y, .f, …**)**
Return a logical vector.
map2_lgl(l2, l1, `%in%`)

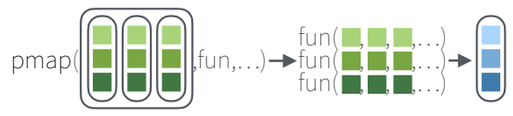
map2_dfc(.x, .y, .f, ...**)**
Return a data frame created by column-binding.
map2_dfc(l1, l2,
~ as.data.frame(c(.x, .y)))

map2_dfr(.x, .y, .f, ..., .id = NULL**)** Return a data frame created by row-binding.
map2_dfr(l1, l2,
~ as.data.frame(c(.x, .y)))

walk2(.x, .y, .f, ...**)** Trigger side effects, return invisibly.
walk2(objs, paths, save)

MANY LISTS

pmap(.l, .f, …**)** Apply a function to groups of elements from a list of lists or vectors, return a list.
pmap(list(x, y, z), ~ ..1 * (..2 + ..3))



pmap_dbl(.l, .f, …**)**
Return a double vector.
pmap_dbl(list(y, z), ~ .x / .y)

pmap_int(.l, .f, …**)**
Return an integer vector.
pmap_int(list(y, z), `+`)

pmap_chr(.l, .f, …**)**
Return a character vector.
pmap_chr(list(l1, l2), paste, collapse = ",", sep = ":")

pmap_lgl(.l, .f, …**)**
Return a logical vector.
pmap_lgl(list(l2, l1), `%in%`)

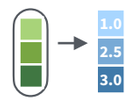
pmap_dfc(.l, .f, ...**)** Return a data frame created by column-binding.
pmap_dfc(list(l1, l2),
~ as.data.frame(c(.x, .y)))

pmap_dfr(.l, .f, ..., .id = NULL**)** Return a data frame created by row-binding.
pmap_dfr(list(l1, l2),
~ as.data.frame(c(.x, .y)))

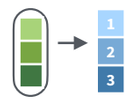
pwalk(.l, .f, ...**)** Trigger side effects, return invisibly.
pwalk(list(objs, paths), save)

LISTS AND INDEXES

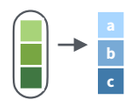
imap(.x, .f, ...**)** Apply .f to each element and its index, return a list.
imap(y, ~ paste0(.y, ": ", .x))



imap_dbl(.x, .f, …**)**
Return a double vector.
imap_dbl(y, ~ .y)

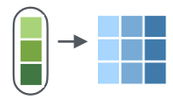


imap_int(.x, .f, ...**)**
Return an integer vector.
imap_int(y, ~ .y)



imap_chr(.x, .f, …**)**
Return a character vector.
imap_chr(y, ~ paste0(.y, ": ", .x))

imap_lgl(.x, .f, …**)**
Return a logical vector.
imap_lgl(l1, ~ is.character(.y))



imap_dfc(.x, .f, ...**)**
Return a data frame created by column-binding.
imap_dfc(l2,
~ as.data.frame(c(.x, .y)))

imap_dfr(.x, .f, ..., .id = NULL**)**
Return a data frame created by row-binding.
imap_dfr(l2,
~ as.data.frame(c(.x, .y)))

iwalk(.x, .f, ...**)** Trigger side effects, return invisibly.
iwalk(z, ~ print(paste0(.y, ": ", .x)))

Function Shortcuts

Use **~ .** with functions like **map()** that have single arguments.

map(l, ~ . + 2)
becomes
map(l, function(x) x + 2))

Use **~ .x .y** with functions like **map2()** that have two arguments.

map2(l, p, ~ .x +.y)
becomes
map2(l, p, function(l, p) l + p)

Use **~ ..1 ..2 ..3** etc with functions like **pmap()** that have many arguments.

pmap(list(a, b, c), ~ ..3 + ..1 - ..2)
becomes
pmap(list(a, b, c), function(a, b, c) c + a -b)

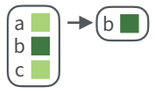
Use **~ .x .y** with functions like **imap()**. .x will get the list value and .y will get the index, or name if available.

imap(list(a, b, c), ~ paste0(.y, ": ", .x)
outputs **"index: value"** for each item

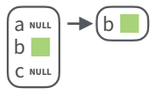
Use a **string** or an **integer** with any map function to index list elements by name or position. **map(l, "name")** becomes **map(l, function(x) x[["name"]])**

Work with Lists

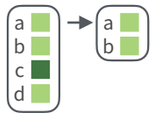
Filter



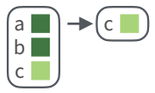
keep(.x, .p, …**)**
Select elements that pass a logical test.
Conversely, **discard()**.
keep(x, is.na)



compact(.x, .p = identity**)**
Drop empty elements.
compact(x)



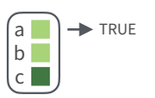
head_while(.x, .p, …**)**
Return head elements until one does not pass.
Also **tail_while()**.
head_while(x, is.character)



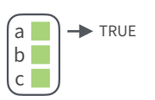
detect(.x, .f, ..., dir = c("forward", "backward"), .right = NULL, .default = NULL**)**
Find first element to pass.
detect(x, is.character)

detect_index(.x, .f, ..., dir = c("forward", "backward"), .right = NULL**)** Find index of first element to pass.
detect_index(x, is.character)

every(.x, .p, …**)**
Do all elements pass a test?
every(x, is.character)

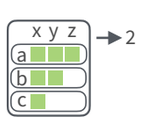


some(.x, .p, …**)**
Do some elements pass a test?
some(x, is.character)



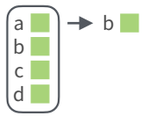
none(.x, .p, …**)**
Do no elements pass a test?
none(x, is.character)

has_element(.x, .y**)**
Does a list contain an element?
has_element(x, "foo")

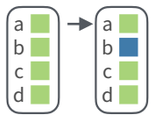


vec_depth(x**)**
Return depth (number of levels of indexes).
vec_depth(x)

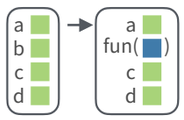
Index



pluck(.x, ..., .default=NULL**)**
Select an element by name or index. Also **attr_getter()** and **chuck()**.
pluck(x, "b")
x %>% pluck("b")

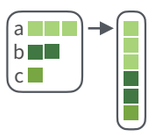


assign_in(x, where, value**)**
Assign a value to a location using pluck selection.
assign_in(x, "b", 5)
x %>% assign_in("b", 5)



modify_in(.x, .where, .f**)**
Apply a function to a value at a selected location.
modify_in(x, "b", abs)
x %>% modify_in("b", abs)

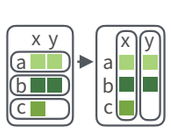
Reshape



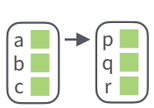
flatten(.x**)** Remove a level of indexes from a list.
Also **flatten_chr()** etc.
flatten(x)

array_tree(array, margin = NULL**)** Turn array into list.
Also **array_branch()**.
array_tree(x, margin = 3)

cross2(.x, .y, .filter = NULL**)**
All combinations of .x and .y.
Also **cross()**, **cross3()**, and **cross_df()**.
cross2(1:3, 4:6)



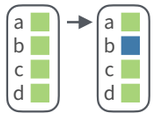
transpose(.l, .names = NULL**)**
Transposes the index order in a multi-level list.
transpose(x)



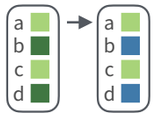
set_names(x, nm = x**)**
Set the names of a vector/list directly or with a function.
set_names(x, c("p", "q", "r"))
set_names(x, tolower)

Modify

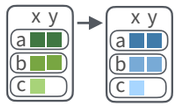
modify(.x, .f, ...**)** Apply a function to each element. Also **modify2()**, and **imodify()**.
modify(x, ~.+ 2)



modify_at(.x, .at, .f, ...**)** Apply a function to selected elements. Also **map_at()**.
modify_at(x, "b", ~.+ 2)

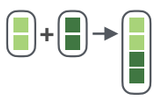


modify_if(.x, .p, .f, ...**)** Apply a function to elements that pass a test. Also **map_if()**.
modify_if(x, is.numeric,~.+2)



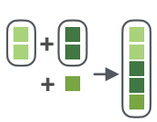
modify_depth(.x, .depth, .f, ...**)** Apply function to each element at a given level of a list. Also **map_depth()**.
modify_depth(x, 2, ~.+ 2)

Combine



append(x, values, after = length(x)**)** Add values to end of list.
append(x, list(d = 1))

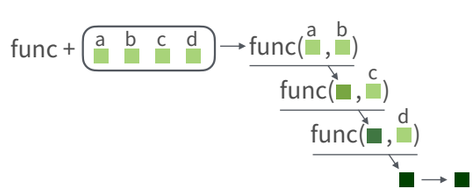
prepend(x, values, before = 1**)**
Add values to start of list.
prepend(x, list(d = 1))



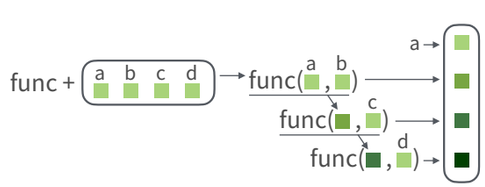
splice(…**)** Combine objects into a list, storing S3 objects as sub-lists.
splice(x, y, "foo")

Reduce

reduce(.x, .f, ..., .init, .dir = c("forward", "backward")**)** Apply function recursively to each element of a list or vector. Also **reduce2()**.
reduce(x, sum)



accumulate(.x, .f, ..., .init**)** Reduce a list, but also return intermediate results. Also **accumulate2()**.
accumulate(x, sum)



List-Columns

max	seq
3	<int [3]>
4	<int [4]>
5	<int [5]>

List-columns are columns of a data frame where each element is a list or vector instead of an atomic value. Columns can also be lists of data frames. See **tidyr** for more about nested data and list columns.

WORK WITH LIST-COLUMNS

Manipulate list-columns like any other kind of column, using **dplyr** functions like **mutate()** and **transmute()**. Because each element is a list, use **map functions** within a column function to manipulate each element.

map(), **map2()**, or **pmap()** return lists and will **create new list-columns**.

```
starwars %>%
  transmute(ships = map2(vehicles,
                         starships,
                         append)
```

(list function, return list; list-columns; column function)

Suffixed map functions like **map_int()** return an atomic data type and will **simplify list-columns into regular columns**.

```
starwars %>%
  mutate(n_films = map_int(films, length))
```

(list function, return int; column function; list-column)

```
library(dplyr)
library(repurrrsive)
library(purrr)
library(tidyr)
library(ggplot2)
library(modelr)
library(broom)

# 1.3 ----------------------------------------------------------------------
# 1. Using the gap_split data in the repurrrsive package
# a. How many elements are in the list?
gap_split |> length()

# b. Do the elements have names?
(names(gap_split) |> length()) > 0

# c. Extract the data from the United Kingdom. What type of data is it?
gap_split[['United Kingdom']] |> class()

# 2. Write a function that, when given the data and a country name will calculate the
mean life expectancy for that country.
mean_life_expectancy <- function(input, country_name) {
  input[[country_name]] |>
    pull('lifeExp') |>
    mean()
}
mean_life_expectancy(gap_split, 'United Kingdom')



# 2.1 ----------------------------------------------------------------------
# 1. Using the split gapminder data:
# a. Find the minimum value of the population for each country
#option 1
pops <- map(gap_split, "pop")
map(pops, min)

gap_split |>
  map("pop") |>
```

```
  map(min)

# other solutions ----------------------------------------------------
min_pop <- function(data){
  data |>
    filter(pop == min(pop)) |>
    pull("pop")
}

gap_split |> map(min_pop)

#option 3
gap_split |> map(function(x) {
  x[['pop']] |> min()
})

#option 4
gap_split |> map(~ min(.$pop))

# b ------------------------------------------------------------------
#Calculate the variance of the GDP per capita
gap_split |>
  map('gdpPercap') |>
  map(var)

# Extension Questions
# 2. For each country, extract the value of the population in 1952.
pop52 <- function(data) {
  data |>
    filter(year == 1952) |>
    pull('pop')
}
gap_split |> map(pop52)

# 3. Which country had the lowest population in 1952? (hint: take a look at which.min)
gap_split |>
  map(pop52) |>
  which.min()
```

```
# 2.2 ----------------------------------------------------------------------
# 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
get_year_of_low_pop <- function(input) {
  input |>
    filter(pop == min(pop)) |>
    pull("year")
}

# 2. Run this function on the split gapminder data to find the year that each country had its lowest population.
gap_split |> map(get_year_of_low_pop)

# 3. Re-write this code to use the purrr shortcuts
gap_split |> map(function(x) {
  x |>
    filter(pop == min(pop)) |>
    pull("year")
})

gap_split |> map(~get_year_of_low_pop(.))

# Extension
# 4. Which country had its lowest population most recently?
gap_split |>
  map(get_year_of_low_pop) |>
  which.max() |> names()



# 2.3 ----
# 1. Find the average life expectancy for each country, storing the output in a numeric vector
life_exp_vec <- gap_split |> map_dbl(function(x) {
  x[['lifeExp']] |> mean()
```

```
})

# 2. Can you store the output in an integer vector?
life_exp_vec <- gap_split |> map_int(function(x) {
  x[['lifeExp']] |> mean()
})
# No

# 3.1 ----
# 1. Write a function to test if the life expectancy 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.
is_most_recent <- function(input) {
  rec_life_exp <- input |>
    filter(year == max(year, na.rm = TRUE)) |>
    pull('lifeExp')

  max_life_exp <- max(input$lifeExp, na.rm = TRUE)

  rec_life_exp == max_life_exp
}



# 2. Test your function on the data for Botswana and the data for Denmark.
is_most_recent(gap_split[['Botswana']])
is_most_recent(gap_split[['Denmark']])

# 3. Filter the split gapminder data to return only elements where the life expectancy in
2007 is not it's highest life expectancy.
not_most_rec <- gap_split |>
  discard(is_most_recent)

not_most_rec |>
  names()

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

```
pop_comparison <- function(input) {
  life_exp_max <- max(
    input[['lifeExp']],
    na.rm = TRUE)

  life_exp_2007 <- input |>
    filter(year == 2007) |>
    pluck('lifeExp')

  data.frame(
    le_max = life_exp_max,
    le_2007 = life_exp_2007
  )
}

not_most_rec |>
  map_dfr(
    pop_comparison,
    .id = 'country') |>
  mutate(
    delta = le_2007 /  le_max - 1
    ) |>
  View()



# 4.1 ----
# 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 …")
country_max_life_exp <- function(life_exp, country_name) {
  paste0(
    'The maximum for ',
    country_name, ' is ',
    max(life_exp, na.rm = TRUE)
  )
}
```

```
# 2. Create:
# a. a list containing only life expectancy values for all countries in the gapminder data
ls_life_exp <- gap_split |> map('lifeExp')

# b. a list of the country names in the gapminder data
ls_country_names <- gap_split |> names() |> as.list()

# 3. Apply your function over the list of life expectancies and countries
map2(ls_life_exp, ls_country_names, country_max_life_exp)

list(
  life_exp = ls_life_exp,
  country_names = ls_country_names
) |>
  pmap(function(life_exp, country_names) {
    country_max_life_exp(life_exp, country_names)
  })

# 4.2 ----
# 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 …")
country_max_life_exp_v2 <- function(life_exp, country_name) {
  output_string <- paste0(
    'The maximum for ',
    country_name, ' is ',
    max(life_exp, na.rm = TRUE)
  )

  print(output_string)
}

# 2. Apply your function over the list of life expectancies using the iwalk function.
gap_split |>
  map('lifeExp') |>
  iwalk(country_max_life_exp_v2)
```

```
# 5.1 ----
# 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
gap_nested |>
  mutate(
    min_pop = map_dbl(data, ~ min(.$pop)),
    var_gdp = map_dbl(data, ~ var(.$gdpPercap))
  ) |>
View()

gap_nested |>
  mutate(
    min_pop = map_dbl(data, ~ min(.[['pop']])),
    var_gdp = map_dbl(data, ~ var(.[['gdpPercap']]))
  ) |>
  View()

# Extension Questions
# 2. For each country, extract the value of the population in 1952.
get_pop_year <- function(input, target_year) {
  input |>
    filter(year == target_year) |>
    pluck('pop')
}
gap_nested |>
  mutate(
    pop_1952 = map_dbl(
      data, ~ get_pop_year(., 1952))
  )

# 3. Which country had the lowest population in 1952?
gap_nested_pop_year <- gap_nested |>
  mutate(
    pop_1952 = map_dbl(
      data, ~ get_pop_year(., 1952))
  )
```

```
gap_nested_pop_year |>
  select(country, pop_1952) |>
  arrange(pop_1952)
# alternatively ...
gap_nested_pop_year[['country']][which.min(gap_nested_pop_year[['pop_1952']])]

idx <- gap_nested_pop_year[['pop_1952']] |>
  which.min()

gap_nested_pop_year[['country']][[idx]]

# 5.2 ----
# 1. Starting with the gap_simple data, convert to a nested data frame, grouping by
continent.
gap_simple |>
  group_by(continent) |>
  nest()

# 2. For each continent, fit a single linear model for life expectancy
gap_simple |>
  group_by(continent) |>
  nest() |>
  mutate(
    model = map(data, ~ lm(
      data = ., formula = lifeExp ~ year + pop + gdpPercap
    ))
  )

# 3. Add a column containing the model metrics to the nested data
out <- gap_simple |>
  group_by(continent) |>
  nest() |>
  mutate(
    model = map(data, ~ lm(
      data = ., formula = lifeExp ~ year + pop + gdpPercap
    )),
    model_metrics = map(model, glance)
```

```
  )

# additionally ...
out |>
  mutate(adj_rsqr = map_dbl(model_metrics, 'adj.r.squared'))

out |>
  unnest(cols = c(model_metrics)) |>
  View()
```

EARL conference

Thank you!