The Strategy Unit.

Functional Programming with purrr

Download slides from https://github.com/NHS-R-Community/Webinars/
Join the NHS-R Community on Slack https://tinyurl.com/nhsrslack

Tom Jemmett, Senior Healthcare Analyst

The Strategy Unit, Midlands and Lancashire Commissioning Support Unit

20th May 2020

What is the purrr package for?

> purrr enhances R’s functional programming (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors...

https://purrr.tidyverse.org/

> **Functional programming** (often abbreviated FP) is the process of building software by composing **pure functions**, avoiding shared state, mutable data, and **side-effects**. Functional programming is **declarative** rather than imperative, and application state flows through pure functions. Contrast with object oriented programming, where application state is usually shared and colocated withmethods in objects.
>
> Functional programming is a **programming paradigm**, meaning that it is a way of thinking about software construction based on some fundamental, defining principles (listed above). Other examples of programming paradigms include object oriented programming and procedural programming.
>
> Functional code tends to be more concise, more predictable, and easier to test than imperative or object oriented code — but if you’re unfamiliar with it and the common patterns associated with it, functional code can also seem a lot more dense, and the related literature can be impenetrable to newcomers.

Master the JavaScript Interview: What is Functional Programming?

> Don’t let all the new words scare you away. It’s a lot easier than it sounds.

functions

creating functions

creating functions in R is pretty simple: we assign a function to a variable, just like any other variable assignment:

```r
my_fn ← function(x) {
  2 * x
}
```

a function is made up of arguments (the bit between the parentheses) and the body (the bit between the curly brackets). The last executed expression is "returned" from the function

```r
my_fn(3)
```

```
## [1] 6
```

arguments are like variables that we can specify when we "call" the function.

a function can have 0 arguments, or many arguments

```r
my_fn ← function() {
  3
}
my_fn()
```

```
## [1] 3
```

```r
my_fn ← function(x, y) {
  x + y
}
my_fn(3, 2)
```

```
## [1] 5
```

a function can be "variadic" by specifying a `...` argument, this is somewhat outside of the scope for today, but allows you to create functions with a variable number of arguments. Often in R this is used to provide arguments that are passed to other functions.

```r
function(x, y, ...) {
  other_function(...)
}
```

The body of the function can be split over multiple lines of code. It's common to include flow-control statements within a function, e.g.

- `if` and `else` statements
- `for` and `while` loops
- `stop` statements (used to produce an "error" message)
- `return` statements (immediately exits the function returning a given value)

```r
my_fn ← function(x) {
  if (x > 10) {
    stop ("x is too big!")
  } else if (x > 5) {
    return (x*2)
  }
  y ← x - 1
  for (i in 1:5) {
    y ← y*2
  }
  y
}
```

```r
my_fn(2)
```

```
## [1] 32
```

```r
my_fn(5)
```

```
## [1] 128
```

```r
my_fn(6)
```

```
## [1] 12
```

```r
my_fn(11)
```

```
## Error in my_fn(11): x is too big!
```

function argument/variable scope

the values that are used in the function only exist in the function: we can't use them outside of the function

```
my_fn <- function(x) {
  y <- 2 * x
  c(exists("x"), exists("y"))
}

my_fn(3)
```

```
## [1] TRUE TRUE
```

```
c(exists("x"), exists("y"))
```

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

we can access variables that are defined outside of the function, but we can't modify them[*]

```
v ← 8

my_fn ← function(x) {
  y ← 2 * x
  v ← v - 1
  y + v
}

my_fn(3)
```

```
## [1] 13
```

Now, inside the function call we decreased the value of `v`, but if we look at what value `v` has now we will see it hasn't changed

```
v
```

```
## [1] 8
```

R has created a new version of `v` inside the function and updated that value.

[*] we can modify then using the `←` operator, but this should be avoided

default arguments

you can specify default values for arguments: an argument with a default value does not have to be specified when you call the function

```
my_fn <- function(x, y = 3) {
  x + y
}

my_fn(4)
```

```
## [1] 7
```

In this case we only passed a value for `x` (3), `y` defaulted to 3. But you can provide values, like so

```
my_fn(4, 5)
```

```
## [1] 9
```

argument order

by default, the arguments are evaluated in order (`x`, then `y`). But, you can specify them in any order if you provide the name of the argument:

```
my_fn(y = 5, x = 4)
```

```
## [1] 9
```

```
my_fn(y = 5, 4)
```

```
## [1] 9
```

using functions like variables

because functions are just variables, you are able to pass functions as arguments to other functions

```r
average ← function(values, fn) {
  fn(values)
}

average(c(1, 2, 5, 8), mean)
```

```
## [1] 4
```

```r
average(c(1, 2, 5, 8), median)
```

```
## [1] 3.5
```

note, here the functions are passed without the parentheses: we are passing the function, not the result of evaluating the function you can inspect a function just by typing it's name

```
my_fn
```

```
## function(x, y = 3) {
##   x + y
## }
## <bytecode: 0x0000000008fe4668>
```

you can also inspect functions from other packages, but this may not always be very useful*

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x0000000006c82d60>
## <environment: namespace:base>
```

* functions that look like "UseMethod" use S3 generics, which are outside of the scope of this session. See Object Orientated chapters from Advanced R

functions can be returned from other functions: this pattern is sometimes called a "function factory", as the "outer" function can produce many different functions.

```
dice ← function(sides) {
  function(rolls) {
    sample(1:sides, rolls, TRUE)
  }
}

five_sided ← dice(5)

five_sided(1)
```

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

```
five_sided(10)
```

```
##  [1] 4 4 1 5 1 2 2 1 5 1
```

```
dice(12)(20)
```

```
##  [1]  3  2  7  3  2  8  4  8  4 11  6  2  8  4  4  2 12  2  2  1
```

pure functions

A pure function is a function that given the same inputs always returns the same outputs with no side-effects, that is we aren't changing any state in the program.

an impure function

```
value ← 4
my_fn ← function(x) {
  x + value
}
a ← my_fn(3)
value ← 10
b ← my_fn(3)
c(a, b)
```

```
## [1]  7 13
```

a pure function

```
value ← 4
my_fn ← function(x, y) {
  x + y
}
a ← my_fn(3, value)
value ← 10
b ← my_fn(3, value)
c(a, b)
```

```
## [1]  7 13
```

Pure-functions are much easier to reason with; we can replace a function with the results of the function and our program would still execute in the same way. This makes it much easier to reason with and test your code.

However, it does eliminate a whole class of operations, like reading and writing data from disk!

vectors

There are two main types of vector in R:

Atomic vectors

There are 4 main types of vector's that you will use:

- logical (TRUE/FALSE)
- integer
- double
- character

together, integer and double vectors are known as numerics.

Other types, such as dates and factors are built from these types of vectors, but with attributes and classes to handle the additional logic and data required to implement those types.

An atomic vector is homogenous: they only contain the one type of data.

Lists

A list is heterogenous: it can store different types of data, including other lists.

```
my_list ← list(1,
                c(2, 3),
                "Hello",
                list(4, 5))
str(my_list)
```

```
## List of 4
##  $ : num 1
##  $ : num [1:2] 2 3
##  $ : chr "Hello"
##  $ :List of 2
##   ..$ : num 4
##   ..$ : num 5
```

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

```
## [1] 2 3
```

named vectors

You can access items from a vector by position, for instance:

```
letters[1:3]
```

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

Alternatively, you can create a named vector, and then you can access the items by their name instead.

```
named_vector ← c(a = 1, b = 2, c = 3)
named_vector
```

```
## a b c
## 1 2 3
```

```
named_vector[["a"]]
```

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

You can also add names to a vector after it has been created. There are a couple of ways of achieving this, but the best way is to use the `set_names` function from `purrr`.

```
1:3 %>% set_names(c("a", "b", "c"))
```

```
## a b c
## 1 2 3
```

`set_names` will use a provided vector of values and set these as the names of the items. If you don't provide a vector of names, it will use the values of the vector to create the names instead.

```
letters[1:3] %>% set_names()
```

```
##   a   b   c
## "a" "b" "c"
```

dataframes/tibbles

A dataframe is just a named list, but with a specific constraint: each item in the list is a vector, and every vector in the list has the same length.

That is, items in the list are the columns in the dataframe, and the rows are the nth items from each vector.

The names of the items in the list are the column names.

```
class(iris)
```

```
## [1] "data.frame"
```

```
typeof(iris)
```

```
## [1] "list"
```

```
length(iris)
```

```
## [1] 5
```

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3
##  $ Species     : Factor w/ 3 levels "setosa","ver
```

iteration

Imagine we have a series of numbers:

```
my_values ← c(1,4,5,3,2)
my_values
```

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

and a function:

```
my_fn ← function(x) {
  3 * x + 2
}
```

If we wanted to get the value of `my_fn(x)` for each value from `my_values`, we could simply run:

```
my_fn(my_values)
```

```
## [1]  5 14 17 11  8
```

This is because R is a "vectorised" language. However, not all functions work like this, so sometimes you have to fall back to writing a loop.

for loops

First we have to set up a "results" vector to store the values in:

```
results ← numeric(length(my_values))
results
```

```
## [1] 0 0 0 0 0
```

Then we need to set up a loop to iterate over the values:

```
for (i in 1:length(my_values)) {
  results[[i]] ← my_fn(my_values[[i]])
}
results
```

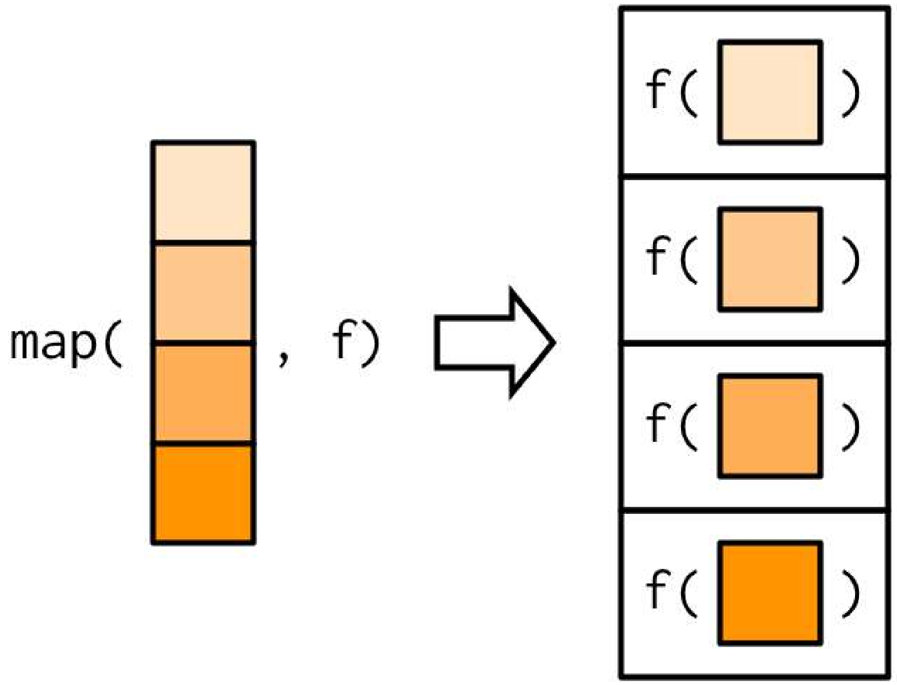
```
## [1]  5 14 17 11  8
```

Writing out a loop like this though is problematic.

- We can forget to initialise our results vector...
- ...or we could initialise it incorrectly (too big or too small)
- We could set up the loop incorrectly (not iterate over all the items from `my_values`, or go past the end of it)
- We could mess up the indexing into results and my_values, e.g. we could always use the first value from `my_values` if we write `my_fn(my_values[[1]])`

map

the function `map` can replace for loops: it takes as arguments the input that you wish to iterate over, and a function that you want to evaluate each item from the input against.



(image from Advanced-R, CC-BY-NC-SA 4.0)

```
results ← map(my_values, my_fn)
results
```

```
## [[1]]
## [1] 5
##
## [[2]]
## [1] 14
##
## [[3]]
## [1] 17
##
## [[4]]
## [1] 11
##
## [[5]]
## [1] 8
```

This gives us the correct results, but `map` always returns a `list`. Lists are much harder to work with, for example, we can't write something like this now:

```
mean(results)
```

```
## Warning in mean.default(results): argument is not numeric or logical: returning
## NA
```

```
## [1] NA
```

But `purrr` provides variants of the `map` function if your function returns a single value. There are 4 main variants that are for when you return an atomic vector:

- `map_lgl` for logicals
- `map_int` for integers
- `map_dbl` for doubles
- `map_chr` for characters

In the previous example, our function returns a double, so we can use `map_dbl`

```
map_dbl(my_values, my_fn)
```

```
## [1]  5 14 17 11  8
```

the function argument

if the function you are using is only ever going to be used in that map expression, you can create an "anonymous" function:

```r
map_dbl(my_values, function(x) {
  3 * x + 2
})
```

```
## [1]  5 14 17 11  8
```

purrr also provides a "formula" syntax which is much quicker to type out

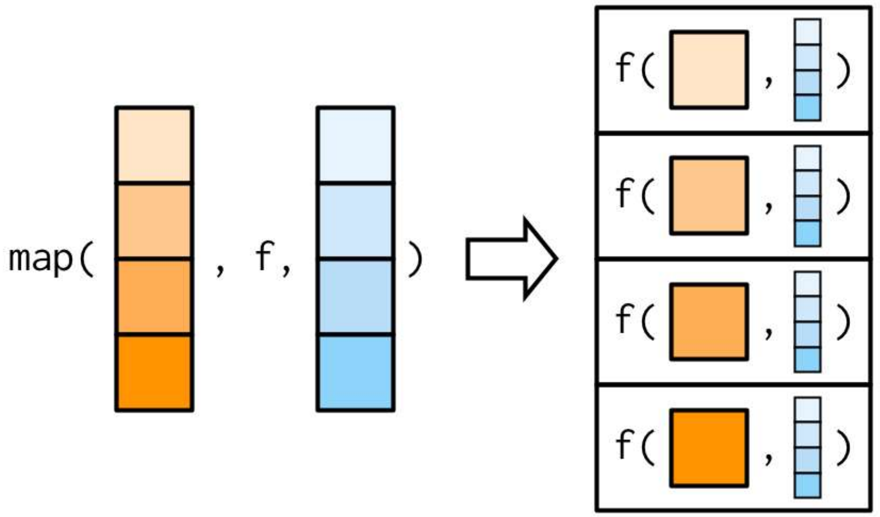
```r
map_dbl(my_values, ~ 3 * .x + 2)
```

```
## [1]  5 14 17 11  8
```

but which is the best to use?

- formula syntax is great if your function is a simple, one-liner
- anonymous syntax works well if your function is a little more complex
- if you have a very complex function that runs over multiple lines, or the function is used by more than just one map statement, then create a function in the environment

if your function has more arguments that you need to set, then these can be passed in via the "..." argument of `map`



(image from Advanced-R, CC-BY-NC-SA 4.0)

```
my_fn <- function(x, y) {
  3 * x + y
}

map_dbl(my_values, my_fn, 2)
```

```
## [1]  5 14 17 11  8
```

it's good practice to use the name of the argument though

```
map_dbl(my_values, my_fn, y = 2)
```

```
## [1]  5 14 17 11  8
```

Using map with dataframes

if we use `map` with a dataframe as the input, it will run the function against each column, using all of the values in the columns.

For example, if you have a dataframe with 5 columns and 150 rows, running `map` will run the function 5 times (not 150 times, or 5 * 150 times).

remember: a dataframe can be thought of as a list

this can be useful to do things like:

look at the classes of each column in your dataframe

```
head(iris, 3)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
```

```
map_chr(iris, class)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##    "numeric"    "numeric"    "numeric"    "numeric"     "factor"
```

compare that to

```
class(iris)
```

```
## [1] "data.frame"
```

calculate some summary for each column

e.g. number of unique values in each column

```
head(iris, 3)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
```

```
iris %>%
  map(unique) %>%
  map_dbl(length)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##           35           23           43           22            3
```

We can also see here that we can achieve more complex results by chaining map statements together

Note that there are many, possibly better, ways to achieve results in R.

With the previous example, we could just stick to dplyr and write

```
iris %>% summarise_all(n_distinct)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           35          23           43          22       3
```

map is a very generalised way to solve a problem though; it can help solve problems that may have clever solutions, but solutions that require you to know other tricks and/or functions.

Using map within dataframes

You can use `map` within a mutate statement to create a new column of values. This can be useful if you want to create a column of values from another column, but the function is not vectorised: e.g. you cannot just run a statement like `mutate(new = fn(col))`

```r
tibble(files = dir(pattern = "\\.png$")) %>%
  mutate(filesize = map_dbl(files, file.size))
```

```
## # A tibble: 2 x 2
##   files                filesize
##   <chr>                   <dbl>
## 1 map-arg-recycle.png     25319
## 2 map.png                 20279
```

A more practical example of using map

now, in the examples above we didn't need to resort to using `map` as the functions were vectorised. But not all functions are, for example, what if we want to read in a folder full of csvs?

```
files ← dir("ae_attendances/",
            "^\\d{4}-\\d{2}-\\d{2}\\.csv$",
            full.names = TRUE)
head(files, 10)
```

```
##  [1] "ae_attendances/2016-04-01.csv" "ae_attendances/2016-05-01.csv"
##  [3] "ae_attendances/2016-06-01.csv" "ae_attendances/2016-07-01.csv"
##  [5] "ae_attendances/2016-08-01.csv" "ae_attendances/2016-09-01.csv"
##  [7] "ae_attendances/2016-10-01.csv" "ae_attendances/2016-11-01.csv"
##  [9] "ae_attendances/2016-12-01.csv" "ae_attendances/2017-01-01.csv"
```

if we want to read in the files, what happens when we call `read_csv` with this vector of file names?

```
read_csv(files)
```

```
## # A tibble: 35 x 1
##    `ae_attendances/2016-04-01.csv`
##    <chr>
##  1 ae_attendances/2016-05-01.csv
##  2 ae_attendances/2016-06-01.csv
##  3 ae_attendances/2016-07-01.csv
##  4 ae_attendances/2016-08-01.csv
##  5 ae_attendances/2016-09-01.csv
##  6 ae_attendances/2016-10-01.csv
##  7 ae_attendances/2016-11-01.csv
##  8 ae_attendances/2016-12-01.csv
##  9 ae_attendances/2017-01-01.csv
## 10 ae_attendances/2017-02-01.csv
## # ... with 25 more rows
```

we could use map to iterate over the files and call the `read_csv` function.

it's good practice with `read_csv` to specify the column types, so I'm adding this in: you will have to specify this yourself based on your file types

```
map(files, read_csv, col_types = "ccddd")
```

```
## [[1]]
## # A tibble: 344 x 5
##    org_code type  attendances breaches admissions
##    <chr>    <chr>       <dbl>    <dbl>      <dbl>
##  1 RF4      1           18788     4082       4074
##  2 RF4      2             561        5          0
##  3 RF4      other        2685       17          0
##  4 R1H      1           27396     5099       6424
##  5 R1H      2             700        5          0
##  6 R1H      other       10317      143          0
##  7 AD913    other        3836        1          0
##  8 RYX      other       17369        0          0
##  9 RQM      1           15154     1199       3415
## 10 RQM      other        6936       76          0
## # ... with 334 more rows
##
## [[2]]
## # A tibble: 345 x 5
##    org_code type  attendances breaches admissions
##    <chr>    <chr>       <dbl>    <dbl>      <dbl>
##  1 RF4      1           20827     4492       4306
##  2 RF4      2             563        0          0
```

the output of this isn't the best though: `map` returns a list of dataframes, in this case because all of the files are the same structure we might want to load this into a single dataframe.

the `map_dfr` function returns a dataframe by binding rows (e.g. SQL UNION ALL).

```
map_dfr(files, read_csv, col_types = "ccddd")
```

```
## # A tibble: 12,765 x 5
##    org_code type  attendances breaches admissions
##    <chr>    <chr>       <dbl>    <dbl>      <dbl>
##  1 RF4      1           18788     4082       4074
##  2 RF4      2             561        5          0
##  3 RF4      other        2685       17          0
##  4 R1H      1           27396     5099       6424
##  5 R1H      2             700        5          0
##  6 R1H      other       10317      143          0
##  7 AD913    other        3836        1          0
##  8 RYX      other       17369        0          0
##  9 RQM      1           15154     1199       3415
## 10 RQM      other        6936       76          0
## # ... with 12,755 more rows
```

unfortunately, our csv's don't contain the date that the file relates to! if we look again at the filenames, the date is in the filename

```
head(files, 2)
```

```
## [1] "ae_attendances/2016-04-01.csv" "ae_attendances/2016-05-01.csv"
```

`map_dfr` has a neat trick up it's sleeves though if you are using a named vector: the `.id` argument takes the name of each item in the vector and adds it to a column with the name that you give it. So first, we need a named vector:

```
files %>%
  set_names() %>%
  head(3)
```

```
##   ae_attendances/2016-04-01.csv   ae_attendances/2016-05-01.csv 
## "ae_attendances/2016-04-01.csv" "ae_attendances/2016-05-01.csv" 
##   ae_attendances/2016-06-01.csv 
## "ae_attendances/2016-06-01.csv"
```

```
files %>%
  set_names() %>%
  map_dfr(read_csv, col_types = "ccddd", .id = "filename") %>%
  # here I'm using a regular expression to find a string that looks like a date
  # in y-m-d format, which is followed by .csv, and the .csv is the end of the
  # string
  mutate(period = str_extract(filename,
                              r"(\d{4}-\d{2}-\d{2}(?=\.csv$))") %>%
           lubridate::ymd())
```

```
## # A tibble: 12,765 x 7
##    filename             org_code type  attendances breaches admissions period
##    <chr>                <chr>    <chr>       <dbl>    <dbl>      <dbl> <date>
##  1 ae_attendances/201~ RF4      1           18788     4082       4074 2016-04-01
##  2 ae_attendances/201~ RF4      2             561        5          0 2016-04-01
##  3 ae_attendances/201~ RF4      other        2685       17          0 2016-04-01
##  4 ae_attendances/201~ R1H      1           27396     5099       6424 2016-04-01
##  5 ae_attendances/201~ R1H      2             700        5          0 2016-04-01
##  6 ae_attendances/201~ R1H      other       10317      143          0 2016-04-01
##  7 ae_attendances/201~ AD913    other        3836        1          0 2016-04-01
##  8 ae_attendances/201~ RYX      other       17369        0          0 2016-04-01
##  9 ae_attendances/201~ RQM      1           15154     1199       3415 2016-04-01
## 10 ae_attendances/201~ RQM      other        6936       76          0 2016-04-01
## # ... with 12,755 more rows
```

for an explanation of what the regular expression is doing above, see here

more variants on map

`map` works over a single vector of values, but what if you have a binary function (a function that takes two arguments)?

`map2` takes two vectors as argument, `.x` and `.y`, and a binary function `.f`.

`map2` also has the same variants as `map`, for example `map2_dbl`:

```
map2_dbl(1:3, 4:6, ~ .x * .y)
```

```
## [1]  4 10 18
```

`pmap` is a generalisation of map that works over as many vectors as you have.

You have to put all of your vectors into a list, and then have a function that accepts as many arguments as you have vectors. The vectors are passed into the function in the order you specify them...

```
list(1:3, 4:6, 7:9) %>%
  pmap_dbl(function(x, y, z) x * y + z)
```

```
## [1] 11 18 27
```

... unless you use a named vector, then purrr will match the list items to the functions arguments

```
list(a = 1:3, b = 4:6, c = 7:9) %>%
  pmap_dbl(function(c, b, a) a * b + c)
```

```
## [1] 11 18 27
```

walk

`walk` is like `map`, except it is designed for functions that you want to run solely for the side-effect.

`walk` doesn't have equivalents to `map_*`, but there is `walk2` and `pwalk`.

Instead of returning the results of the initial function, `walk` returns the input vector, e.g. while

```
map(.x, .f) => .f(.x)
```

with walk we have

```
walk(.x, .f) => .x
```

`walk` is useful to use with operations like saving plots to disk.

demo

In base R:

the `apply`/`lapply` functions from base R does the same thing as `map`. There are then `sapply`/`vapply` which are most similar to the `map_*` functions, but they are not as easy (in my opinion) to work with.

The `map` functions have a far more consistent api to work with, and the documentation is much more approachable and easier to understand.

`map` summary:

- map is like a for loop where you apply a function to every item in the input
- the first argument of map is the input (you want to map over)
- the second argument is the function (you want to apply)
- the ... argument's are passed to the function (for every input)
- map returns a list, map_* returns a vector of that type
- map_dfr returns a dataframe binding by rows (map_dfc binds by columns)
- the function argument can either be a defined function (from a package, or one you have created), an anonymous function, or a formula
- map2 is for when you have 2 vectors to iterate over (as a pair)
- pmap is for when you have multiple vectors to iterate over (all the vectors are contained in a list)
- walk/walk2/pwalk is for when you want to execute a function just for the side effects

Thanks!

Please submit feedback for this session at www.menti.com (code: 32 55 85)!

Download the slides at https://github.com/NHS-R-Community/Webinars/

And join the NHS-R Community on Slack! https://tinyurl.com/nhsrslack

Suggested further reading

R resources

The book R for Data Science covers all of the material covered today

- Vectors chapter in R4DS
- Functions chapter in R4DS
- Iteration chapter in R4DS
- Functional Programming chapters in Advanced R

More general resources

These resources aren't about R, but can help you to learn the concepts of functional programming.

- Master the JavaScript Interview: What is Functional Programming?
- Learn you a Haskell for Great Good!
- Category Theory for Programmers