Functional programming & purrr

Lecture 08

Dr. Colin Rundel

Functional Programming

Functions as objects

We have mentioned in passing that in R functions are treated as 1st class objects (like vectors), meaning they can be assigned names, stored in lists, passed as arguments, etc.

```
f = function(x) {
  x*x
}
f(2)
```

```
[1] 4
```

```
g = f
g(2)
```

```
[1] 4
```

```
l = list(f = f, g = g)
l$f(3)
```

```
[1] 9
```

```
l[[2]](4)
```

```
[1] 16
```

```
l[1](3)
```

```
Error: attempt to apply non-
function
```

Functions as arguments

We can pass in functions as arguments to other functions,

```
do_calc = function(v, func) {
  func(v)
}
```

```
do_calc(1:3, sum)
```

```
[1] 6
```

```
do_calc(1:3, mean)
```

```
[1] 2
```

```
do_calc(1:3, sd)
```

```
[1] 1
```

Anonymous functions

These are short functions that are created without ever assigning a name,

```
1 function(x) {x+1}
```

```
function (x)
{
    x + 1
}
```

```
1 (function(y) {y-1})(10)
```

```
[1] 9
```

this can be particularly helpful for implementing certain types of tasks,

```
1 integrate(function(x) x, 0, 1)
```

```
0.5 with absolute error < 5.6e-15
```

```
1 integrate(function(x) x^2-2*x+1, 0, 1)
```

```
0.3333333 with absolute error < 3.7e-15
```

Base R - anonymous function shorthand

Along with the base pipe (`|>`), R v4.1.0 introduced a shortcut for anonymous functions using `\()`,

```
1 (\(x) {1+x})(1:5)
```

```
[1] 2 3 4 5 6
```

```
1 (function(x) {1+x})(1:5)
```

```
[1] 2 3 4 5 6
```

```
1 (\(x) x^2)(10)
```

```
[1] 100
```

```
1 (function(x) x^2)(10)
```

```
[1] 100
```

```
1 integrate(\(x) sin(x)^2, 0, 1)
```

```
0.2726756 with absolute error < 3e-15
```

```
1 integrate(function(x) sin(x)^2, 0, 1)
```

```
0.2726756 with absolute error < 3e-15
```

We can use this with the base pipe to avoid using `_`,

```
1 data.frame(x = runif(10), y = runif(10)) |>
2   (\(d) lm(y~x, data = d))()
```

```
Call:
lm(formula = y ~ x, data = d)

Coefficients:
(Intercept)            x
     0.3927       0.1989
```

apply (base R)

Apply functions

The apply functions are a collection of tools for functional programming in base R, they are variations of the `map` function found in many other languages and apply a function over the elements of an input (vector).

```
??base::apply

## Help files with alias or concept or title matching ‘apply
## matching:
##
## base::apply                  Apply Functions Over Array Margins
## base::.subset                Internal Objects in Package 'base
## base::by                     Apply a Function to a Data Frame S
## base::eapply                 Apply a Function Over Values in an
## base::lapply                 Apply a Function over a List or Ve
## base::mapply                 Apply a Function to Multiple List
## base::rapply                 Recursively Apply a Function to a
## base::tapply                 Apply a Function Over a Ragged Arr
```

lapply

> Usage: `lapply(X, FUN, ...)`
>
> `lapply` returns a list of the same length as `X`, each element of which is the result of applying `FUN` to the corresponding element of `X`.

```
1 lapply(1:8, sqrt) |>
2   str()
```

```
List of 8
 $ : num 1
 $ : num 1.41
 $ : num 1.73
 $ : num 2
 $ : num 2.24
 $ : num 2.45
 $ : num 2.65
 $ : num 2.83
```

```
1 lapply(1:8, function(x) (x+1)^2) |>
2   str()
```

```
List of 8
 $ : num 4
 $ : num 9
 $ : num 16
 $ : num 25
 $ : num 36
 $ : num 49
 $ : num 64
 $ : num 81
```

Argument matching

```
1 lapply(
2   1:8, function(x, pow) x^pow, pow=3
3 ) |>
4   str()
```

```
List of 8
 $ : num 1
 $ : num 8
 $ : num 27
 $ : num 64
 $ : num 125
 $ : num 216
 $ : num 343
 $ : num 512
```

```
1 lapply(
2   1:8, function(x, pow) x^pow, x=2
3 ) |>
4   str()
```

```
List of 8
 $ : num 2
 $ : num 4
 $ : num 8
 $ : num 16
 $ : num 32
 $ : num 64
 $ : num 128
 $ : num 256
```

sapply

> Usage: `sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)`
>
> `sapply` is a *user-friendly* version and wrapper of `lapply`, it is a *simplifying* version of lapply. Whenever possible it will return a vector, matrix, or an array.

```
1 sapply(1:8, sqrt)
```

```
[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751
[8] 2.828427
```

```
1 sapply(1:8, function(x) (x+1)^2)
```

```
[1]  4  9 16 25 36 49 64 81
```

```
1 sapply(1:8, function(x) c(x, x^2, x^3))
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    2    3    4    5    6    7    8
[2,]    1    4    9   16   25   36   49   64
[3,]    1    8   27   64  125  216  343  512
```

Length mismatch?

```r
sapply(1:6, seq) |> str()
```

```
List of 6
 $ : int 1
 $ : int [1:2] 1 2
 $ : int [1:3] 1 2 3
 $ : int [1:4] 1 2 3 4
 $ : int [1:5] 1 2 3 4 5
 $ : int [1:6] 1 2 3 4 5 6
```

```r
lapply(1:6, seq) |> str()
```

```
List of 6
 $ : int 1
 $ : int [1:2] 1 2
 $ : int [1:3] 1 2 3
 $ : int [1:4] 1 2 3 4
 $ : int [1:5] 1 2 3 4 5
 $ : int [1:6] 1 2 3 4 5 6
```

Type mismatch?

```r
l = list(a = 1:3, b = 4:6, c = 7:9, d = list(10, 11, "A"))
```

```r
sapply(l, function(x) x[1]) |> str()
```

```
List of 4
 $ a: int 1
 $ b: int 4
 $ c: int 7
 $ d: num 10
```

```r
sapply(l, function(x) x[[1]]) |> str()
```

```
 Named num [1:4] 1 4 7 10
 - attr(*, "names")= chr [1:4] "a" "b" "c" "d"
```

```r
sapply(l, function(x) x[[3]]) |> str()
```

```
 Named chr [1:4] "3" "6" "9" "A"
 - attr(*, "names")= chr [1:4] "a" "b" "c" "d"
```

*apply and data frames

We can use these functions with data frames, the key is to remember that a data frame is just a fancy list.

```
df = data.frame(
  a = 1:6,
  b = letters[1:6],
  c = c(TRUE,FALSE)
)
```

```
lapply(df, class) |> str()
```

```
List of 3
 $ a: chr "integer"
 $ b: chr "character"
 $ c: chr "logical"
```

```
sapply(df, class)
```

```
        a           b           c 
"integer" "character"   "logical" 
```

A more useful example

Some sources of data (e.g. some US government agencies) will encode missing values with `-999`, if want to replace these with `NA`s lapply is not a bad choice.

```
d = tibble::tribble(
  ~patient_id, ~age,  ~bp,  ~o2,
             1,   32,  110,   97,
             2,   27,  100,   95,
             3,   56,  125, -999,
             4,   19, -999, -999,
             5,   65, -999,   99
)
```

```
fix_missing = function(x) {
  x[x == -999] = NA
  x
}
lapply(d, fix_missing) |> str()
```

```
List of 4
 $ patient_id: num [1:5] 1 2 3 4 5
 $ age       : num [1:5] 32 27 56 19 65
 $ bp        : num [1:5] 110 100 125 NA NA
 $ o2        : num [1:5] 97 95 NA NA 99
```

```
lapply(d, fix_missing) |>
  as_tibble()
```

```
# A tibble: 5 × 4
  patient_id   age    bp    o2
       <dbl> <dbl> <dbl> <dbl>
1          1    32   110    97
2          2    27   100    95
3          3    56   125    NA
4          4    19    NA    NA
5          5    65    NA    99
```

dplyr alternative

dplyr is also a viable option here using the `across()` helper,

```
d |>
  mutate(
    across(
      bp:o2,
      fix_missing
    )
  )
```

```
# A tibble: 5 × 4
  patient_id   age    bp    o2
       <dbl> <dbl> <dbl> <dbl>
1          1    32   110    97
2          2    27   100    95
3          3    56   125    NA
4          4    19    NA    NA
5          5    65    NA    99
```

```
d |>
  mutate(
    across(
      where(is.numeric),
      fix_missing
    )
  )
```

```
# A tibble: 5 × 4
  patient_id   age    bp    o2
       <dbl> <dbl> <dbl> <dbl>
1          1    32   110    97
2          2    27   100    95
3          3    56   125    NA
4          4    19    NA    NA
5          5    65    NA    99
```

other less common apply functions

- `apply()` - applies a function over the rows or columns of a matrix or array (data frames also work but are bad idea)
- `vapply()` - is similar to `sapply`, but has a enforced return type and size
- `mapply()` - like `sapply` but will iterate over multiple vectors at the same time.
- `rapply()` - a recursive version of `lapply`, behavior depends largely on the `how` argument
- `eapply()` - apply a function over an environment.


purrr
www.rstudio.com

Map functions

Basic functions for looping over objects and returning a value (of a specific type) - type consistent replacements for `lapply`/`sapply`/`vapply`.

- `map()` - returns a list, equivalent to `lapply()`
- `map_lgl()` - returns a logical vector.
- `map_int()` - returns a integer vector.
- `map_dbl()` - returns a double vector.
- `map_chr()` - returns a character vector.
- `walk()` - returns nothing, used for side effects

Type Consistency

R is a weakly / dynamically typed language which means there is no syntactic way to define a function which enforces argument or return types. This flexibility can be useful at times, but often it makes it hard to reason about your code and requires more verbose code to handle edge cases.

```
1 x = list(rnorm(1e3), rnorm(1e3), rnorm(1e3))
```

```
1 map_dbl(x, mean)
```

```
[1] 0.02267445 0.01264834 0.01897747
```

```
1 map_int(x, mean)
```

```
Error in `map_int()`:
ℹ In index: 1.
Caused by error:
! Can't coerce from a number to an integer.
```

```
1 map_chr(x, mean)
```

```
Warning: Automatic coercion from double to
character was deprecated in purrr
1.0.0.
ℹ Please use an explicit call to
`as.character()` within `map_chr()`
  instead.
[1] "0.022674" "0.012648" "0.018977"
```

```
1 map(x, mean) |> str()
```

```
List of 3
 $ : num 0.0227
 $ : num 0.0126
 $ : num 0.019
```

```
1 lapply(x, mean) |> str()
```

```
List of 3
 $ : num 0.0227
 $ : num 0.0126
 $ : num 0.019
```

Working with Data Frames

purrr offers the functions `map_dfr` and `map_dfc` (which were superseded as of v1.0.0) - these allow for the construction of a data frame by row or by column respectively.

```
d = tibble::tribble(
  ~patient_id, ~age,  ~bp,  ~o2,
             1,   32,  110,   97,
             2,   27,  100,   95,
             3,   56,  125, -999,
             4,   19, -999, -999,
             5,   65, -999,   99
)
```

```
fix_missing = function(x) {
  x[x == -999] = NA
  x
}
```

```
purrr::map_dfc(d, fix_missing)
```

```
# A tibble: 5 × 4
  patient_id   age    bp    o2
       <dbl> <dbl> <dbl> <dbl>
1          1    32   110    97
2          2    27   100    95
3          3    56   125    NA
4          4    19    NA    NA
5          5    65    NA    99
```

```
purrr::map(d, fix_missing) |>
  bind_cols()
```

```
# A tibble: 5 × 4
  patient_id   age    bp    o2
       <dbl> <dbl> <dbl> <dbl>
1          1    32   110    97
2          2    27   100    95
3          3    56   125    NA
4          4    19    NA    NA
5          5    65    NA    99
```

Building by row

```
1 map(sw_people, function(x) x[1:5]) |> bind_rows()
```

```
# A tibble: 87 × 5
   name               height mass  hair_color    skin_color
   <chr>              <chr>  <chr> <chr>         <chr>
 1 Luke Skywalker     172    77    blond         fair
 2 C-3PO              167    75    n/a           gold
 3 R2-D2              96     32    n/a           white, blue
 4 Darth Vader        202    136   none          white
 5 Leia Organa        150    49    brown         light
 6 Owen Lars          178    120   brown, grey   light
 7 Beru Whitesun lars 165    75    brown         light
 8 R5-D4              97     32    n/a           white, red
 9 Biggs Darklighter  183    84    black         light
10 Obi-Wan Kenobi     182    77    auburn, white fair
# ℹ 77 more rows
```

```
1 map(sw_people, function(x) x) |> bind_rows()
```

```
Error in `vctrs::data_frame()`:
! Can't recycle `name` (size 5) to match `vehicles` (size 2).
```

purrr style anonymous functions

purrr lets us write anonymous functions using one sided formulas where the argument is given by `.` or `.x` for `map` and related functions.

```
map_dbl(1:5, function(x) x/(x+1))
```

```
[1] 0.5000000 0.6666667 0.7500000 0.8000000 0.8333333
```

```
map_dbl(1:5, ~ ./(.+1))
```

```
[1] 0.5000000 0.6666667 0.7500000 0.8000000 0.8333333
```

```
map_dbl(1:5, ~ .x/(.x+1))
```

```
[1] 0.5000000 0.6666667 0.7500000 0.8000000 0.8333333
```

Generally, the latter option is preferred to avoid confusion with magrittr.

Multiargument anonymous functions

Functions with the `map2` prefix work the same as the `map` prefixed functions but they iterate over two objects instead of one. Arguments for an anonymous function are given by `.x` and `.y` (or `..1` and `..2`) respectively.

```
map2_dbl(1:5, 1:5, function(x,y) x / (y+1))
```

```
[1] 0.5000000 0.6666667 0.7500000 0.8000000 0.8333333
```

```
map2_dbl(1:5, 1:5, ~ .x/(.y+1))
```

```
[1] 0.5000000 0.6666667 0.7500000 0.8000000 0.8333333
```

```
map2_dbl(1:5, 1:5, ~ ..1/(..2+1))
```

```
[1] 0.5000000 0.6666667 0.7500000 0.8000000 0.8333333
```

```
map2_chr(LETTERS[1:5], letters[1:5], paste0)
```

```
[1] "Aa" "Bb" "Cc" "Dd" "Ee"
```

Indexed maps

purrr also has a collection of `imap` prefixed functions which are short hand for mapping over an object and the indexes of that object (i.e. `seq_along(obj)`).

```
iwalk(
  letters[1:5],
  ~cat("index: ", .y, ", value: ", .x, "\n", sep="")
)
```

```
index: 1, value: a
index: 2, value: b
index: 3, value: c
index: 4, value: d
index: 5, value: e
```

Lookups

Very often we want to extract only certain values by name or position from a list, `purrr` provides a shorthand for this operation - instead of a function you can provide either a character or numeric vector, those values will be used to sequentially subset the elements being iterated.

```
1 purrr::map_chr(sw_people, "name") |> head()
```

```
[1] "Luke Skywalker" "C-3PO"          "R2-D2"
[4] "Darth Vader"    "Leia Organa"    "Owen Lars"
```

```
1 purrr::map_chr(sw_people, 1) |> head()
```

```
[1] "Luke Skywalker" "C-3PO"          "R2-D2"
[4] "Darth Vader"    "Leia Organa"    "Owen Lars"
```

```
1 purrr::map_chr(sw_people, list("films", 1)) |> head(n=10)
```

```
 [1] "http://swapi.co/api/films/6/" "http://swapi.co/api/films/5/"
 [3] "http://swapi.co/api/films/5/" "http://swapi.co/api/films/6/"
 [5] "http://swapi.co/api/films/6/" "http://swapi.co/api/films/5/"
 [7] "http://swapi.co/api/films/5/" "http://swapi.co/api/films/1/"
 [9] "http://swapi.co/api/films/1/" "http://swapi.co/api/films/5/"
```

Length coercion?

```
purrr::map_chr(sw_people, list("starships", 1))
```

```
Error in `purrr::map_chr()`:
ℹ In index: 2.
Caused by error:
! Result must be length 1, not 0.
```

```
sw_people[[2]]$name
```

```
[1] "C-3PO"
```

```
sw_people[[2]]$starships
```

```
NULL
```

```
purrr::map_chr(sw_people, list("starships", 1), .default = NA) |> head()
```

```
[1] "http://swapi.co/api/starships/12/"
[2] NA
[3] NA
[4] "http://swapi.co/api/starships/13/"
[5] NA
[6] NA
```

```
purrr::map(sw_people, list("starships", 1)) |> head() |> str()
```

```
List of 6
 $ : chr "http://swapi.co/api/starships/12/"
 $ : NULL
 $ : NULL
 $ : chr "http://swapi.co/api/starships/13/"
```

list columns

```
(chars = tibble(
  name = purrr::map_chr(
    sw_people, "name"
  ),
  starships = purrr::map(
    sw_people, "starships"
  )
))
```

```
# A tibble: 87 × 2
   name               starships
   <chr>              <list>
 1 Luke Skywalker     <chr [2]>
 2 C-3PO              <NULL>
 3 R2-D2              <NULL>
 4 Darth Vader        <chr [1]>
 5 Leia Organa        <NULL>
 6 Owen Lars          <NULL>
 7 Beru Whitesun lars <NULL>
 8 R5-D4              <NULL>
 9 Biggs Darklighter  <chr [1]>
10 Obi-Wan Kenobi     <chr [5]>
# ℹ 77 more rows
```

```
chars |>
  mutate(
    n_starships = map_int(
      starships, length
    )
  )
```

```
# A tibble: 87 × 3
   name               starships n_starships
   <chr>              <list>          <int>
 1 Luke Skywalker     <chr [2]>           2
 2 C-3PO              <NULL>              0
 3 R2-D2              <NULL>              0
 4 Darth Vader        <chr [1]>           1
 5 Leia Organa        <NULL>              0
 6 Owen Lars          <NULL>              0
 7 Beru Whitesun lars <NULL>              0
 8 R5-D4              <NULL>              0
 9 Biggs Darklighter  <chr [1]>           1
10 Obi-Wan Kenobi     <chr [5]>           5
# ℹ 77 more rows
```

Example

List columns and approximating pi

Complex heirarchical data

Often we may encounter complex data structures where our goal is not to rectangle every value (which may not even be possible) but rather to rectangle a small subset of the data.

```
1 str(repurrrsive::discog, max.level = 3)
```

```
List of 155
 $ :List of 5
  ..$ instance_id      : int 354823933
  ..$ date_added       : chr "2019-02-16T17:48:59-08:00"
  ..$ basic_information:List of 11
  .. ..$ labels      :List of 1
  .. ..$ year        : int 2015
  .. ..$ master_url  : NULL
  .. ..$ artists     :List of 1
  .. ..$ id          : int 7496378
  .. ..$ thumb       : chr "https://img.discogs.com/vEVegHrMNTsP6xG_K6OuFXz4h_U=/fit-
in/150x150/filters:strip_icc():format(jpeg):mode_rgb()"| __truncated__
  .. ..$ title       : chr "Demo"
  .. ..$ formats     :List of 1
  .. ..$ cover_image : chr "https://img.discogs.com/EmbMh7vsElksjRgoXLFSuY1sjRQ=/fit-
in/500x499/filters:strip_icc():format(jpeg):mode_rgb()"| __truncated__
  .. ..$ resource_url: chr "https://api.discogs.com/releases/7496378"
       $ master_id   : int 0
```

Partial vs complete rectangling

In the case of `discog` we may want to rectangle the `id`, `year`, `title`, `artist`, and `label` fields but leave the rest of the data as is.

In cases like this using tidyr's `unnest_wider()` and `unnest_long()` is not ideal as they will attempt to rectangle the entire data set when we only want a subset. There is no need to do the expensive work of unnesting columns we will never use.

purrr's `map_*()` functions and tidyr's `hoist()` function are useful for targeting specific columns to rectangle.

purrr

```
tibble(disc = repurrrsive::discog) |>
  mutate(
    id     = purrr::map_int(disc, "id"),
    year   = purrr::map_int(disc, c("basic_information", "year")),
    title  = purrr::map_chr(disc, c("basic_information", "title")),
    artist = purrr::map_chr(disc, list("basic_information", "artists", 1, "name")),
    label  = purrr::map_chr(disc, list("basic_information", "labels", 1, "name"))
  )
```

```
# A tibble: 155 × 6
   disc                    id  year title                    artist label
   <list>               <int> <int> <chr>                    <chr>  <chr>
 1 <named list [5]>   7496378  2015 Demo                     Mollot Tobi…
 2 <named list [5]>   4490852  2013 Observant Com El Mon …   Una B… La V…
 3 <named list [5]>   9827276  2017 I                        S.H.I… La V…
 4 <named list [5]>   9769203  2017 Oído Absoluto            Rata … La V…
 5 <named list [5]>   7237138  2015 A Cat's Cause, No Dog…   Ivy (… Kato…
 6 <named list [5]>  13117042  2019 Tashme                   Tashme High…
 7 <named list [5]>   7113575  2014 Demo                     Desgr… Mind…
 8 <named list [5]>  10540713  2015 Let The Miracles Begin   Phant… Not …
 9 <named list [5]>  11260950  2017 Sub Space                Sub S… Not …
10 <named list [5]>  11726853  2017 Demo                     Small… Pres…
# ℹ 145 more rows
```

hoist()

```
tibble(disc = repurrrsive::discog) %>%
  hoist(
    disc,
    id = "id",
    year = c("basic_information", "year"),
    title = c("basic_information", "title"),
    artist = list("basic_information", "artists", 1, "name"),
    label = list("basic_information", "labels", 1, "name")
  )
```

```
# A tibble: 155 × 6
         id  year title                          artist label disc
      <int> <int> <chr>                          <chr>  <chr> <list>
 1  7496378  2015 Demo                           Mollot Tobi… <named list>
 2  4490852  2013 Observant Com El Mon Es D…     Una B… La V… <named list>
 3  9827276  2017 I                              S.H.I… La V… <named list>
 4  9769203  2017 Oído Absoluto                  Rata … La V… <named list>
 5  7237138  2015 A Cat's Cause, No Dogs Pr…     Ivy (… Kato… <named list>
 6 13117042  2019 Tashme                         Tashme High… <named list>
 7  7113575  2014 Demo                           Desgr… Mind… <named list>
 8 10540713  2015 Let The Miracles Begin         Phant… Not … <named list>
 9 11260950  2017 Sub Space                      Sub S… Not … <named list>
10 11726853  2017 Demo                           Small… Pres… <named list>
# ℹ 145 more rows
```