

# Functional Programming

# Functions as objects

We have mentioned in passing that functions in R are treated as 1st class objects (like vectors)

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

```
f(2)
```

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

```
g = f
```

```
g(2)
```

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

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

```
## Error in eval(expr, envir, enclos): attempt to apply non-function
```

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

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

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

```
## [1] 16
```

# 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,

```
function(x) {x+1}  
## function(x) {x+1}  
(function(y) {y-1})(10)  
## [1] 9
```

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

```
integrate(function(x) x, 0, 1)  
## 0.5 with absolute error < 5.6e-15  
integrate(function(x) x^2-2*x+1, 0, 1)  
## 0.3333333 with absolute error < 3.7e-15
```

# 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 the input.

```
??base::apply
---
## Help files with alias or concept or title matching 'apply' using fuzzy
## matching:
## 
## base::apply          Apply Functions Over Array Margins
## base::subset         Internal Objects in Package 'base'
## base::by             Apply a Function to a Data Frame Split by Factors
## base::eapply          Apply a Function Over Values in an Environment
## base::lapply          Apply a Function over a List or Vector
## base::mapply          Apply a Function to Multiple List or Vector Arguments
## base::rapply          Recursively Apply a Function to a List
## base::tapply          Apply a Function Over a Ragged Array
```

# 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`.

```
lapply(1:8, sqrt) %>% 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
```

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

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

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

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

```
lapply(1:8, function(x, pow) x^pow, x=2) %>% 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.

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

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

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

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

```
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
```

## What happens if the returned lengths don't match?

```
sapply(1:6, seq)
```

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

```
lapply(1:6, seq)
```

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

## What happens if the types don't match?

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

```
sapply(l, function(x) x[1])
```

```
## $a  
## [1] 1  
##  
## $b  
## [1] 4  
##  
## $c  
## [1] 7  
##  
## $d  
## [1] 10
```

```
sapply(l, function(x) x[[1]])
```

```
##   a   b   c   d  
##   1   4   7  10
```

```
sapply(l, function(x) x[[3]])
```

```
##   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

Recall Exercise 2 from Lecture 5 where we converted -999 to NAS.

```
d = data.frame(  
  patient_id = c(1, 2, 3, 4, 5),  
  age = c(32, 27, 56, 19, 65),  
  bp = c(110, 100, 125, -999, -999),  
  o2 = c(97, 95, -999, -999, 99)  
)
```

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

```
## $patient_id  
## [1] 1 2 3 4 5  
##  
## $age  
## [1] 32 27 56 19 65  
##  
## $bp  
## [1] 110 100 125 NA NA
```

```
lapply(d, fix_missing) %>%  
  as.data.frame()
```

```
##   patient_id age   bp o2  
## 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 is also a viable option here,

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

```
##   patient_id age  bp o2
## 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)
  )
```

```
##   patient_id age  bp o2
## 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 data frame, matrix or array
- `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.



# Map functions

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

- `map()` - returns a list.
- `map_lgl()` - returns a logical vector.
- `map_int()` - returns a integer vector.
- `map_dbl()` - returns a double vector.
- `map_chr()` - returns a character vector.
- `map_dfr()` - returns a data frame by row binding.
- `map_dfc()` - returns a data frame by column binding.
- `walk()` - returns nothing, function used exclusively for its 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.

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

map_dbl(x, mean)

## [1] 0.01487616 0.04696427 -0.06971873

map_chr(x, mean)

## [1] "0.014876"  "0.046964"  "-0.069719"

map_int(x, mean)

## Error: Can't coerce element 1 from a double to a integer

map(x, mean) %>% str()

lapply(x, mean) %>% str()
```

# Working with Data Frames

`map_dfr` and `map_dfc` are particularly useful when working with and/or creating data frames.

Take for example the Lecture 5 Exercise 2 example from above,

```
d = data.frame(  
  patient_id = c(1, 2, 3, 4, 5),  
  age = c(32, 27, 56, 19, 65),  
  bp = c(110, 100, 125, -999, -999),  
  o2 = c(97, 95, -999, -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
```

```
map_dfr(sw_people, function(x) x[1:5])
```

```
## # 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
## # ... with 77 more rows
```

```
map_dfr(sw_people, function(x) x)
```

```
## Error: Can't recycle `name` (size 5) to match `vehicles` (size 2).
```

# Shortcut - 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 - e.g. map2

Functions with the `map2` prefix work the same as the `map` functions but they iterate over two objects instead of one. Arguments in 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"
```

# Shortcut - Lookups

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

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

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

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

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

```
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: Result 2 must be a single string, not NULL of length 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/" NA
## [3] NA                               "http://swapi.co/api/starships/13/"
## [5] NA
```

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

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

# 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]>  
## # ... with 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  
## # ... with 77 more rows
```

## **Example - list columns and approximating pi**

# Selective rectangling

# Complex hierarchical 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.

```
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_K60uFXz4h_U=/fit-in/150x150/filters:strip"
##     ...$ title          : chr "Demo"
##     ...$ formats        :List of 1
##     ...$ cover_image    : chr "https://img.discogs.com/EmbMh7vsElksjRgoXLFSuY1sjRQ=/fit-in/500x499/filters:strip"
##     ...$ resource_url   : chr "https://api.discogs.com/releases/7496378"
##     ...$ master_id      : int 0
##   ..$ id              : int 7496378
##   ..$ rating          : int 0
```

## **Example - discog + purrr / tidyverse**

# tidyr's hoist

```
tibble(disc = repurrrsive::discog) %>%  
  hoist(  
    disc,  
    info = "basic_information"  
)  
  
## # A tibble: 155 × 2  
##   info             disc  
##   <list>          <list>  
## 1 <named list [11]> <named list [4]>  
## 2 <named list [11]> <named list [4]>  
## 3 <named list [11]> <named list [4]>  
## 4 <named list [11]> <named list [4]>  
## 5 <named list [11]> <named list [4]>  
## 6 <named list [11]> <named list [4]>  
## 7 <named list [11]> <named list [4]>  
## 8 <named list [11]> <named list [4]>  
## 9 <named list [11]> <named list [4]>  
## 10 <named list [11]> <named list [4]>  
## # ... with 145 more rows
```

```
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 Recor... <name...
## 2 4490852 2013 Observant Com El Mon Es Destruix Una Bèst... La Vida Es... <name...
## 3 9827276 2017 I             S.H.I.T.... La Vida Es... <name...
## 4 9769203 2017 Oido Absoluto Rata Neg... La Vida Es... <name...
## 5 7237138 2015 A Cat's Cause, No Dogs Problem Ivy (18) Katorga Wo... <name...
## 6 13117042 2019 Tashme       Tashme     High Fashi... <name...
## 7 7113575 2014 Demo         Desgraci... Mind Contr... <name...
## 8 10540713 2015 Let The Miracles Begin Phantom ... Not On Lab... <name...
## 9 11260950 2017 Sub Space    Sub Spac... Not On Lab... <name...
## 10 11726853 2017 Demo        Small Ma... Prescience... <name...
## # ... with 145 more rows
```