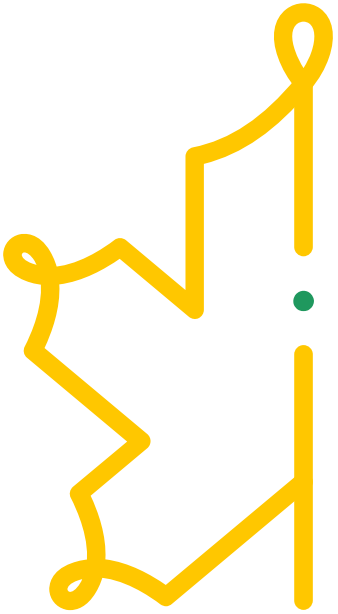
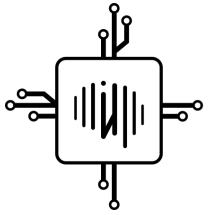


Razвед анализ данных

Функции

Функциональное программирование



Фонд интеллект

Анна Валяева

Лекция 9 - 2022

Функции

- Если вы заметили, что несколько раз используете один и тот же код, то запишите его в функцию.

```
df <- tibble(
  a = 1:10,
  b = rnorm(10),
  c = runif(10))

# возведем все столбцы в куб
df$a <- df$a ** 3
df$b <- df$b * 3
df$c <- df$c **3
```

Функции

- Если вы заметили, что несколько раз используете один и тот же код, то запишите его в функцию.

```
cube <- function(x) {
  x ** 3
}

cube(x = 2)
```

```
[1] 8
```

```
df$a <- cube(df$a)
df$b <- cube(df$b)
df$c <- cube(df$c)
```

Функции

Нужно придумать:

- **имя** функции. Оно не должно совпадать с именами функций из базового R или пакетов, которые вы используете. В идеале оно отражает смысл вашей функции.

```
# не делайте так!
mean <- function(x) { sum(x) }
```

- список **параметров**, которые функция принимает на вход. Например, `function(x, y, z)`.
- сам код, выполняющий работу, который вы записываете в **тело** функции внутри `{...}`.

```
smart_name <- function(input1, input2, param3) {
  ...
  body
  ...
}
```

Функции

- У функции может быть 0 или несколько параметров.
- Функция может возвращать максимум 1 объект.

```r
# ничего не требует
say_hello <- function() {
  print("hello!")
}


# ничего не возвращает
save_res <- function(df) {
  df = df[df$pval < 0.05, c(1,3:5)]
  write.csv(df, "path-to-file.csv")
}
```

Взглянуть на код функции

И на свою функцию посмотреть:

```
cube
```

```
function(x) {
  x ** 3
}
<bytecode: 0x000000001856ff10>
```

И на чужую:

```
xor
```

```
function (x, y)
{
    (x | y) & !(x & y)
}
<bytecode: 0x000000001946a320>
<environment: namespace:base>
```

Return

- Функция возвращает результат последнего выражения либо то, что указано как `return(...)`.

```
cube_or_not <- function(x) {
  x ** 3
}

cube_or_not(2)
```

```
cube_or_not <- function(x) {
  return(x * 3)
  x ** 3
}

cube_or_not(2)
```

Implicit return

```
check_sign_i <- function(x) {
  # check if x is positive
  if (x > 0) {
    "positive"
  }
  # check if x is negative
  else if (x < 0) {
    "negative"
  }
  # check if x is not positive nor negative
  else {
    "zero"
  }
}

check_sign_i(10)
```

```
[1] "positive"
```

Explicit return

```
check_sign_e <- function(x) {
  # check if x is positive
  if (x > 0) {
    return("positive")
  }
  # check if x is negative
  else if (x < 0) {
    return("negative")
  }
  # check if x is not positive nor negative
  else {
    return("zero")
  }
}

check_sign_e(10)
```

```
[1] "positive"
```

Выполнение кода по условию

```
if (condition) {
  # что делать, когда condition = TRUE
} else {
  # что делать, когда condition = FALSE
}
```

Логическое выражение `condition` должно возвращать либо TRUE, либо FALSE.

Выполнение кода по условию

Логическое выражение `condition` должно возвращать либо TRUE, либо FALSE.

- Если `condition` - это вектор, то будет предупреждение:

```
if (c(TRUE, FALSE)) {}
```

```
Warning in if (c(TRUE, FALSE)) {: the condition has length > 1 and only the
first element will be used
```

```
NULL
```

- Если `condition` - это пропущенное значение, то будет ошибка:

```
if (NA) {}
```

```
Error in if (NA) {: missing value where TRUE/FALSE needed
```

Несколько условий

```
if (this) {
  # делай это
} else if (that) {
  # делай что-то другое
} else {
  # делай что-то третье
}
```

Не путайте `else if () {...}` с `ifelse()`.

Несколько условий

Если условий слишком много, то в них можно запутаться. Тогда лучше использовать другие подходы. Например, использовать `switch()`.

```
centre <- function(x, type) {
  switch(type,
         mean = mean(x),
         median = median(x),
         trimmed = mean(x, trim = .1),
         stop("Unknown central tendency!"))
}
```

```
set.seed(123)
x <- rnorm(10)
centre(x, "mean")
```

```
[1] 0.07462564
```

```
centre(x, "median")
```

```
[1] -0.07983455
```

```
centre(x, "trimmed")
```

```
[1] 0.03703159
```

```
centre(x, "mode")
```

```
Error in centre(x, "mode"): Unknown central
   tendency!
```

Параметры

Через параметры на вход функции передаются **данные** или какие-то **детали**. Обычно данные передаются первому параметру. В таком случае эту функцию будет легко использовать с `%>%`.

Для параметров можно задать значение по умолчанию:

```
centre <- function(x, type = "mean") {
  switch(type,
         mean = mean(x),
         median = median(x),
         trimmed = mean(x, trim = .1),
         stop("Unknown central tendency!"))
}

centre(rnorm(10))
```

```
[1] 0.208622
```

Если при вызове функции вы заменяете значение по умолчанию, то указывайте название параметра (не надейтесь только на позицию). Так всем будет понятнее.

Названия параметров

Идеи для названий параметров:

- `x, y, z` - вектора,
- `w` - вектор весов,
- `df` - датафрейм,
- `i, j` - индексы (строки и столбцы),
- `n` - длина или число строк,
- `p` - число столбцов.

Провека формата входных данных

В каком случае нужно остановиться.

if + stop

```
cube <- function(x) {
  if (!is.numeric(x)) {
    stop("`x` must be numeric")
    }
  x ** 3
}

cube("twelve")
```

```
Error in cube("twelve"): `x` must be numeric
```

stopifnot

```
cube <- function(x) {
  stopifnot(is.numeric(x))
  x ** 3
}

cube("twelve")
```

```
Error in cube("twelve"): is.numeric(x) is not TRUE
```

Multiple returns

Чтобы функция возвращала несколько объектов, нужно эти объекты возвращать в виде списка.

```
return_two_and_four <- function(){
  list(2, 4)
}

return_two_and_four()
```

```
[[1]]
[1] 2

[[2]]
[1] 4
```

Локальные переменные

`x` внутри функции (в ее среде) и вне функции (в глобальной среде) существуют независимо.

```
x <- 1000

add_ten <- function(x){
  x + 10
}

add_ten(32)
```

```
[1] 42
```

```
x
```

```
[1] 1000
```

Глобальные переменные

Изнутри функции можно переписать глобальную переменную с помощью оператора `<<-`.

```
x <- 1000

add_ten <- function(x){
  x <<- 32
  x + 10
}

add_ten(32)
```

```
[1] 42
```

```
x
```

```
[1] 32
```

Глобальные переменные

Если R не нашел переменную в среде функции, то он будет искать ее в глобальной среде.

```
y <- 1000

add_ten <- function(){
  y + 10
}

add_ten()
```

```
[1] 1010
```

Векторизация

Многие функции в R приспособлены к работе с векторами и итерации по элементам вектора. Благодаря этому нам не нужно лишний раз писать циклы.

```
1:5 * 2
```

```
[1]  2  4  6  8 10
```

```
1:5 > 3
```

```
[1] FALSE FALSE FALSE  TRUE  TRUE
```

```
vec <- 1:5
out_vec <- c()

for (i in 1:length(vec)) {
  out_vec[i] = vec[i] * 2
}

out_vec
```

```
[1]  2  4  6  8 10
```

Как избегать циклов?

Можно использовать функции из семейства `apply`:

- `apply()`
- `lapply()`
- `sapply()`
- `tapply()`

Как избегать циклов?

Если нужно применить функцию к элементам некоторого списка, то используйте `map` из пакета **{purrr}** (входит в **{tidyverse}**).

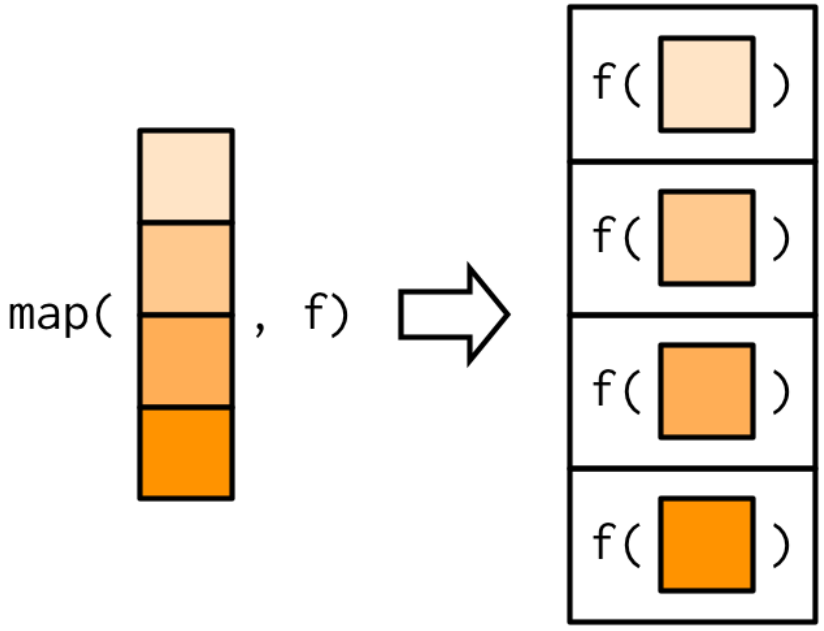
```
# вместо
for (i in 1:3){
  f(i)
}

# или
list(f(1), f(2), f(3))

# нужно всего лишь...
map(1:3, f)
```

```
library(purrr)
# library(tidyverse)
```

Семейство функций `map`

```
cube <- function(x) x ** 3

map(1:3, cube)
```

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

[[2]]
[1] 8

[[3]]
[1] 27
```



Разные `map_`

- Простой `map()` всегда возвращает list.
- Если вы уверены, что ваш результат подходит под определение вектора (данные одного типа), используйте `map_`:
  - `map chr`
  - `map lgl`
  - `map int`
  - `map_dbl`

```
map_dbl(1:3, cube)
```

```
[1]  1  8 27
```

Разные `map_`

```
map_chr(mtcars, typeof)
```

```
     mpg      cyl     disp       hp     drat       wt     qsec       vs
"double" "double" "double" "double" "double" "double" "double" "double"
      am     gear     carb
"double" "double" "double"
```

```
map_lgl(mtcars, is.double)
```

```
 mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

```
map_int(mtcars, n_distinct)
```

```
 mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
  25    3   27   22   22   29   30    2    2    3    6
```

Anonymous functions / lambda functions

```
# Если забыли про n_distinct из dplyr:
map_dbl(mtcars, function(x) length(unique(x)))
```

```
mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
 25    3   27   22   22   29   30    2    2    3    6
```

```
# Лень писать так много. Так проще:
map_dbl(mtcars, ~ length(unique(.x)))
```

```
mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
 25    3   27   22   22   29   30    2    2    3    6
```

map_df

- `map_dfr` - это `map()` + `bind_rows()`
- `map_dfc` - это `map()` + `bind_cols()`

```
input_files <- c("file1.csv", "file2.csv", "file3.csv")
```

```
file1 <- read_csv("file1.csv")
file2 <- read_csv("file2.csv")
file3 <- read_csv("file3.csv")

file <- bind_rows(file1, file2, file3)
```

```
file <- map_dfr(input_files, read_csv)
```

map ❤ СПИСКИ

```
x <- list(
  list(-1, x = 1, y = c(2), z = "a"),
  list(-2, x = 4, y = c(5, 6), z = "b"),
  list(-3, x = 8, y = c(9, 10, 11)))

map_dbl(x, "x") # по имени элемента
```

```
[1] 1 4 8
```

```
map_dbl(x, list("y", 1)) # по позиции
```

```
[1] 2 5 9
```

```
map_chr(x, "z", .default = NA)
```

```
[1] "a" "b" NA
```

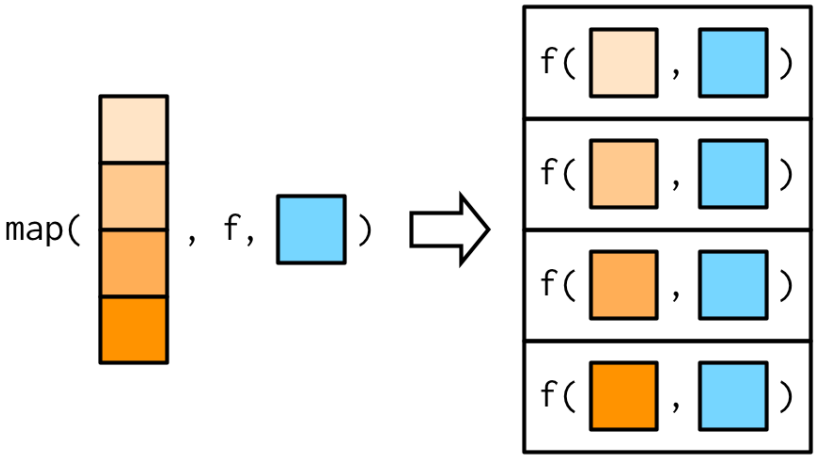
Аргументы функции

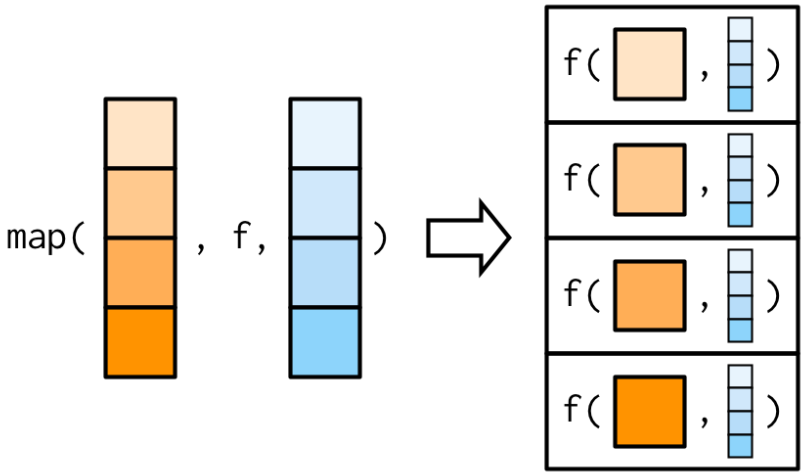
```
x <- list(1:5, c(1:10, NA))

# Не очень
map_dbl(x, ~ mean(.x, na.rm = TRUE))
```

```
[1] 3.0 5.5
```

```
# Получше
map_dbl(x, mean, na.rm = TRUE)
```

```
[1] 3.0 5.5
```



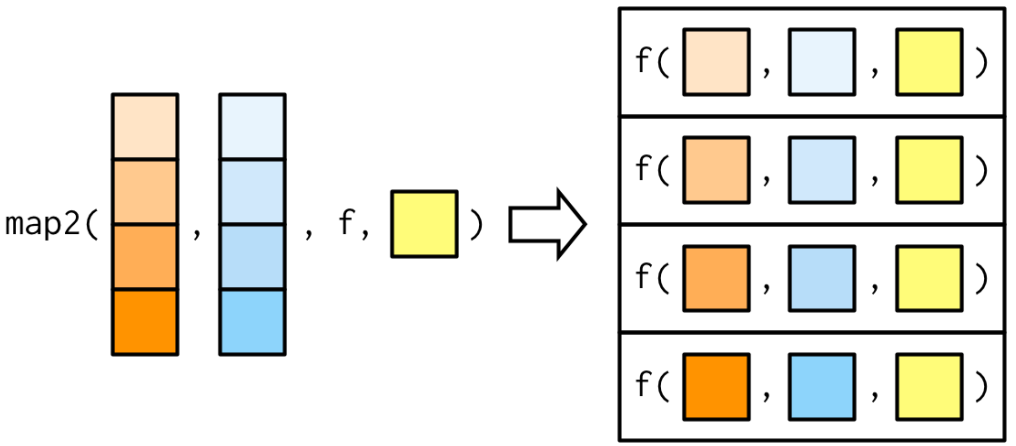


map2

- Если нужно итерировать и по элементам списка, и по вектору аргумента функции.

```
xs <- map(1:8, ~ runif(10))
ws <- map(1:8, ~ rpois(10, 5) + 1)

map2_dbl(xs, ws, weighted.mean, na.rm = TRUE)
```

```
[1] 0.3328024 0.4576310 0.5632621 0.4544391 0.5557217 0.3965493 0.5413291
[8] 0.5414916
```



`pmap`

- Когда не хватает `map2`, а нужен `map3` или даже `map4`...
- Нужно подать список всех аргументов функции.
- `map2(x, y, f)` - то же, что и `pmap(list(x, y), f)`.

```
pmap_dbl(list(xs, ws), weighted.mean)
```

```
[1] 0.3328024 0.4576310 0.5632621 0.4544391 0.5557217 0.3965493 0.5413291
[8] 0.5414916
```

imap

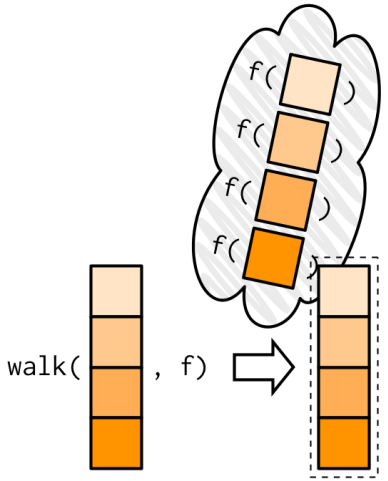
- `imap(x, f)` - то же, что и `map2(x, seq_along(x), f)` или `map2(x, names(x), f)`

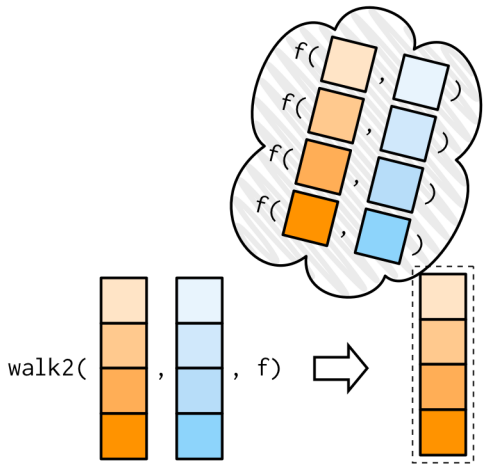
```
# .y - это название элемента списка
# .x - элемент списка
x <- map(1:6, ~ sample(1000, 10))
imap_chr(x, ~ paste0("The highest value of ", .y, " is ", max(.x)))
```

```
[1] "The highest value of 1 is 985" "The highest value of 2 is 876"
[3] "The highest value of 3 is 951" "The highest value of 4 is 804"
[5] "The highest value of 5 is 983" "The highest value of 6 is 978"
```

walk

```
ggplots <- list(gg1, gg2, gg3)
output_files <- c("plot1.png", "plot2.png",
    "plot3.png")

walk2(output_files, ggplots, ggsave)
```



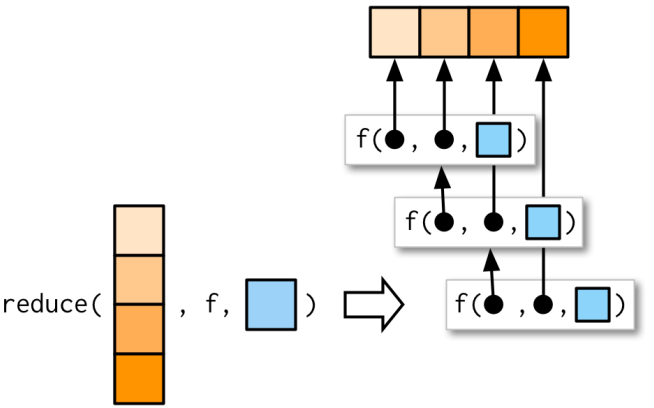


Семейство функций `reduce`

`reduce()` берет на вход вектор длины n и возвращает вектор длины 1, применяя функцию к элементам вектора попарно.

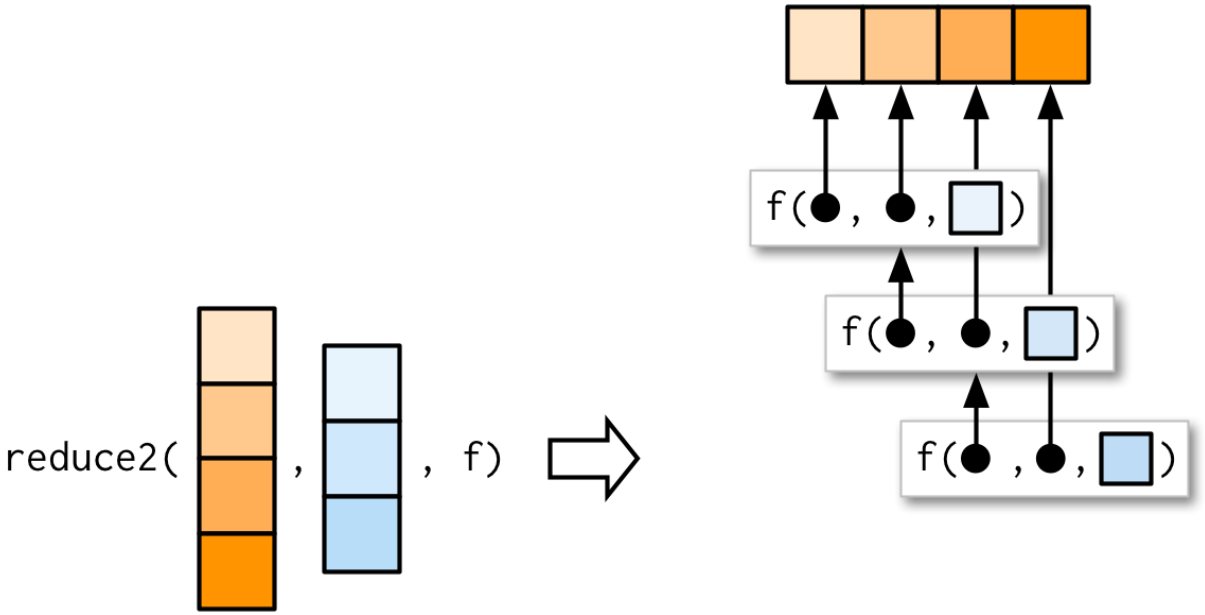
Удобно, если надо объединить несколько датафреймов.

```
# вместо
f(f(f(1, 2), 3), 4)

# нужно всего лишь...
reduce(1:4, f)
```



reduce2

Если нужно, например, объединить несколько датафреймов, но использовать разные переменные, по которым делать объединение.



Отслеживать и ловить ошибки

- `safely()` возвращает список из двух элементов:
  - `result` нужный результат или NULL если была ошибка,
  - `error` error object или NULL, если ошибки не было.
- `possibly()` позволяет использовать default value, если возникает ошибка.
- `quietly()` выдает result, output, messages и warnings.

safely

```
# можно воспринимать safely() как наречие, а функцию log() как глагол
safe_log <- safely(log)
str(safe_log(10))
```

```
List of 2
 $ result: num 2.3
 $ error : NULL
```

```
str(safe_log("a"))
```

```
List of 2
 $ result: NULL
 $ error :List of 2
  ..$ message: chr "non-numeric argument to mathematical function"
  ..$ call   : language .Primitive("log")(x, base)
  ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

`safely`

`safely()` удобно использовать вместе с `map`.

```
x <- list(1, 10, "a")
y <- x %>% map(safely(log))
str(y)
```

```
List of 3
 $ :List of 2
  ..$ result: num 0
  ..$ error : NULL
 $ :List of 2
  ..$ result: num 2.3
  ..$ error : NULL
 $ :List of 2
  ..$ result: NULL
  ..$ error :List of 2
  .. ..$ message: chr "non-numeric argument to mathematical function"
  .. ..$ call   : language .Primitive("log")(x, base)
  .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

`safely`

Можно отделить результаты от ошибок.

```
y <- transpose(y)
str(y)
```

```
List of 2
 $ result:List of 3
  ..$ : num 0
  ..$ : num 2.3
  ..$ : NULL
 $ error :List of 3
  ..$ : NULL
  ..$ : NULL
  ..$ :List of 2
  .. ..$ message: chr "non-numeric argument to mathematical function"
  .. ..$ call   : language .Primitive("log")(x, base)
  .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

Затем определить элементы, которые вызвали ошибку.

```
is_ok <- y$error %>% map_lgl(is_null)
x[!is_ok]
```

```
[[1]]
[1] "a"
```

`possibly`

Позволяет использовать default value, если возникает ошибка.

```
x <- list(1, 10, "a")
x %>% map_dbl(possibly(log, otherwise = NA_real_))
```

```
[1] 0.000000 2.302585       NA
```

`quietly`

`quietly` собирает не ошибки, а предупреждения и сообщения.

```
x <- list(1, -1)
x %>% map(quietly(log)) %>% str()
```

```
List of 2
 $ :List of 4
  ..$ result  : num 0
  ..$ output  : chr ""
  ..$ warnings: chr(0)
  ..$ messages: chr(0)
 $ :List of 4
  ..$ result  : num NaN
  ..$ output  : chr ""
  ..$ warnings: chr "NaNs produced"
  ..$ messages: chr(0)
```

Что почитать про функции и purrr

- Functions Chapter in R4DS
- Functions Chapter in Advanced R
- purrr cheatsheet
- Functional Chapter in Advanced R
- purrr tutorial by Jenny Bryan
- Картинки, иллюстрирующие принцип работы функций из **purrr**, взяты из 'Advanced R' by Hadley Wickham.