Benefits of writing functions

Functions eliminate repetition from your code, which

- can reduce your workload, and
- help avoid errors.

Functions also allow code reuse and sharing.

```
library(readr)
test_scores_geography_raw <- read_csv("test_scores_geography.csv")

library(dplyr)
library(lubridate)
test_scores_geography_clean <- test_scores_geography_raw %>%
  select(person_id, first_name, last_name, test_date, score) %>%
  mutate(test_date = mdy(test_date)) %>%
  filter(!is.na(score))
```

```
library(readr)
test_scores_english_raw <- read_csv("test_scores_english.csv")

library(dplyr)
library(lubridate)
test_scores_english_clean <- test_scores_english_raw %>%
  select(person_id, first_name, last_name, test_date, score) %>%
  mutate(test_date = mdy(test_date)) %>%
  filter(!is.na(score))
```

```
library(readr)
test_scores_art_raw <- read_csv("test_scores_art.csv")

library(dplyr)
library(lubridate)
test_scores_art_clean <- test_scores_art_raw %>%
  select(person_id, first_name, last_name, test_date, score) %>%
  mutate(test_date = mdy(test_date)) %>%
  filter(is.na(score))
```

```
library(readr)
test_scores_spanish_raw <- read_csv("test_scores_spanish.csv")

library(dplyr)
library(lubridate)
test_scores_spanish_clean <- test_scores_spanish_raw %>%
  select(person_id, first_name, last_name, test_date, score) %>%
  mutate(test_date = mdy(test_date)) %>%
  filter(!is.na(score))
```

Function names should contain a verb

- get
- calculate (or maybe just calc)
- run
- process
- import
- clean
- tidy
- draw

lm() is badly named

- Acronyms aren't self-explanatory
- It doesn't contain a verb
- There are lots of di erent linear models

A be er name would be `run_linear_regression()`

Readability vs. typeability

- Understanding code >> typing code
- Code editors have autocomplete
- You can alias common functions

```
h <- head
```

```
data(cats, package = "MASS")
h(cats)
```

```
  Sex Bwt Hwt
1   F 2.0 7.0
2   F 2.0 7.4
3   F 2.0 9.5
4   F 2.1 7.2
5   F 2.1 7.3
6   F 2.1 7.6
```

Types of argument

- Data arguments: what you compute on
- Detail arguments: how you perform the computation

```
args(cor)
```

```
function (x, y = NULL, use = "everything",
  method = c("pearson", "kendall", "spearman"))
```

Set the default in the signature

```
toss_coin <- function(n_flips, p_head = 0.5) {
  coin_sides <- c("head", "tail")
  weights <- c(p_head, 1 - p_head)
  sample(coin_sides, n_flips, replace = TRUE, prob = weights)
}
```

We might default the answer of other argument as simplifyVector vectors does

```
library(jsonlite)
args(fromJSON)
```

```
function (txt, simplifyVector = TRUE, simplifyDataFrame = simplifyVector,
    simplifyMatrix = simplifyVector, flatten = FALSE, ...)
```

Categorical defaults

1. Pass a character vector in the signature.
2. Call `match.arg()` in the body.

```
args(prop.test)
```

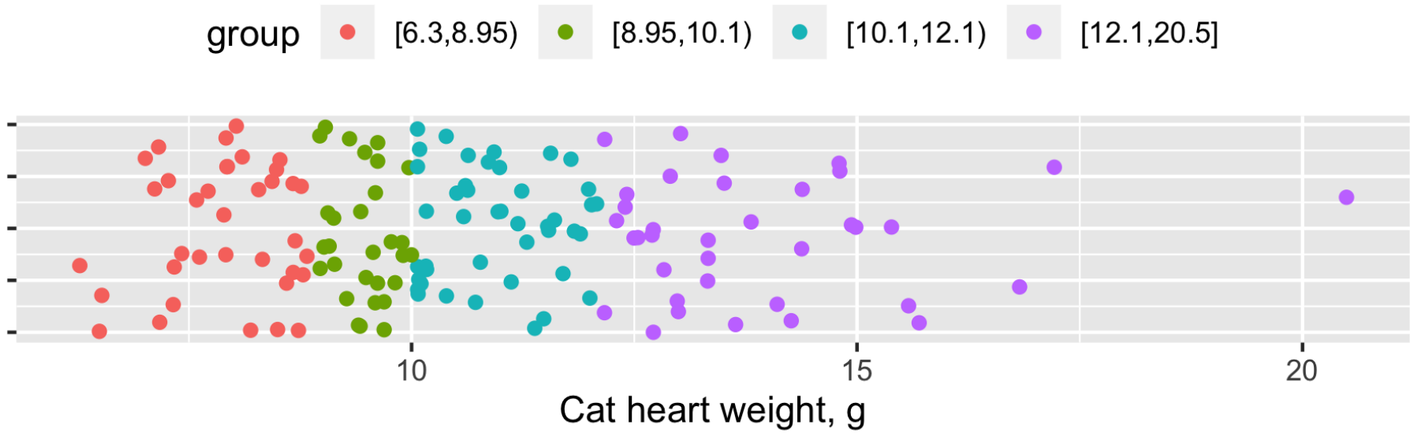
```
function (x, n, p = NULL, alternative = c("two.sided", "less", "greater"),
  conf.level = 0.95, correct = TRUE)
```

Inside the body

```
alternative <- match.arg(alternative)
```

Cutting a vector by quantile

```
# Set the categories for interval_type to "(lo, hi]" and "[lo, hi)"
cut_by_quantile <- function(x, n = 5, na.rm = FALSE, labels = NULL,
                            interval_type = c("(lo, hi]","[lo, hi)")) {
  # Match the interval_type argument
  interval_type <- match.arg(interval_type)
  probs <- seq(0, 1, length.out = n + 1)
  qtiles <- quantile(x, probs, na.rm = na.rm, names = FALSE)
  right <- switch(interval_type, "(lo, hi]" = TRUE, "[lo, hi)" = FALSE)
  cut(x, qtiles, labels = labels, right = right, include.lowest = TRUE)
}
```



Using ...

```
calc_geometric_mean <- function(x, ...) {
  x %>%
    log() %>%
    mean(...) %>%
    exp()
}
```

The tradeoff

Benefits

- Less typing for you
- No need to match signatures

Drawbacks

- You need to trust the inner function
- The interface is not as obvious to users

Checking types of inputs

- `assert_is_numeric()`
- `assert_is_character()`
- `is_data.frame()`
- ...
- `is_two_sided_formula()`
- `is_tskernel()`

Packages to check user inputs

https://github.com/cran/assertive

https://github.com/hadley/assertthat

Using assertive to check x

```
calc_geometric_mean <- function(x, na.rm = FALSE) {
  assert_is_numeric(x)
  x %>%
    log() %>%
    mean(na.rm = na.rm) %>%
    exp()
}
```

```
Error in calc_geometric_mean(letters) :
 is_numeric : x is not of class 'numeric'; it has class 'character'.
```

Checking x is positive

```
calc_geometric_mean <- function(x, na.rm = FALSE) {
  assert_is_numeric(x)
  assert_all_are_positive(x)
  x %>%
    log() %>%
    mean(na.rm = na.rm) %>%
    exp()
}
```

```
calc_geometric_mean(c(1, -1))
```

```
Error in calc_geometric_mean(c(1, -1)) :
  is_positive : x contains non-positive values.
There was 1 failure:
  Position Value   Cause
1        2    -1 too low
```

is_* functions

- `assert_is_numeric()`
- `assert_all_are_positive()`

- `is_numeric()` (returns logical value)
- `is_positive()` (returns logical vector)
- `is_non_positive()`

Custom checks

```
calc_geometric_mean <- function(x, na.rm = FALSE) {
  assert_is_numeric(x)
  if(any(is_non_positive(x), na.rm = TRUE)) {
    stop("x contains non-positive values, so the geometric mean makes no sense.")
  }
  x %>%
    log() %>%
    mean(na.rm = na.rm) %>%
    exp()
}
```

```
calc_geometric_mean(c(1, -1))
```

```
Error in calc_geometric_mean(c(1, -1)) :
  x contains non-positive values, so the geometric mean makes no sense.
```

Fixing input

```
use_first(c(1, 4, 9, 16))
```

```
[1] 1
Warning message:
Only the first value of c(1, 4, 9, 16) (= 1) will be used.
```

```
coerce_to(c(1, 4, 9, 16), "character")
```

```
[1] "1"  "4"  "9"  "16"
Warning message:
Coercing c(1, 4, 9, 16) to class ‘character’.
```

Fixing na.rm

```
calc_geometric_mean <- function(x, na.rm = FALSE) {
  assert_is_numeric(x)
  if(any(is_non_positive(x), na.rm = TRUE)) {
    stop("x contains non-positive values, so the geometric mean makes no sense.")
  }
  na.rm <- coerce_to(use_first(na.rm), target_class = "logical")
  x %>%
    log() %>%
    mean(na.rm = na.rm) %>%
    exp()
}
```

```
calc_geometric_mean(1:5, na.rm = 1:5)
```

```
[1] 2.605171
Warning messages:
1: Only the first value of na.rm (= 1) will be used.
2: Coercing use_first(na.rm) to class ‘logical’.
```

Reasons for returning early

1. You already know the answer.
2. The input is an edge case.

A simple sum function

```
simple_sum <- function(x) {
  if(anyNA(x)) {
    return(NA)
  }
  total <- 0
  for(value in x) {
    total <- total + value
  }
  total
}
```

```
simple_sum(c(0, 1, 3, 6, NA, 7))
```

```
NA
```

Returning a value with a warning

```
calc_geometric_mean <- function(x, na.rm = FALSE) {
  assert_is_numeric(x)
  if(any(is_non_positive(x), na.rm = TRUE)) {
    warning("x contains non-positive values, so the geometric mean makes no sense.")
    return(NaN)
  }
  na.rm <- coerce_to(use_first(na.rm), "logical")
  x %>%
    log() %>%
    mean(na.rm = na.rm) %>%
    exp()
}
```

Hiding of console the return value

```
simple_sum <- function(x) {
  if(anyNA(x)) {
    return(NA)
  }
  total <- 0
  for(value in x) {
    total <- total + value
  }
  invisible(total)
}
```

```
simple_sum(c(0, 1, 3, 6, 2, 7))
```

Using list to get many results

```
session <- function() {
  list(
    r_version = R.version.string,
    operating_system = Sys.info()[c("sysname", "release")],
    loaded_pkgs = loadedNamespaces()
  )
}
```

Multi-assignment from a list

```
library(zeallot)
c(vrsn, os, pkgs) %<-% session()
```

```
vrsn
```

```
"R version 3.5.3 (2019-03-11)"
```

```
os
```

```
sysname                       release
"Linux" "4.14.106-79.86.amzn1.x86_64"
```

Using attributes to save extra info

```
month_no <- setNames(1:12, month.abb)
month_no
```

```
Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
  1   2   3   4   5   6   7   8   9  10  11  12
```

```
attributes(month_no)
```

```
$names
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul"
 [8] "Aug" "Sep" "Oct" "Nov" "Dec"
```

```
attr(month_no, "names")
```

```
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul"
 [8] "Aug" "Sep" "Oct" "Nov" "Dec"
```

```
attr(month_no, "names") <- month.name
month_no
```

```
  January  February     March     April       May
        1         2         3         4         5
     June      July    August September   October
        6         7         8         9        10
 November  December
       11        12
```

Attributes of a data frame

```
orange_trees
```

```
# A tibble: 35 x 3
   Tree    age circumference
   <ord> <dbl>         <dbl>
 1 1       118            30
 2 1       484            58
 3 1       664            87
 4 1      1004           115
 5 1      1231           120
 6 1      1372           142
 7 1      1582           145
 8 2       118            33
 9 2       484            69
10 2       664           111
# … with 25 more rows
```

```
attributes(orange_trees)
```

```
$names
[1] "Tree"          "age"
[3] "circumference"

$row.names
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
[16] 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
[31] 31 32 33 34 35

$class
[1] "tbl_df"     "tbl"        "data.frame"
```

[1] data(Orange, package = "datasets")

Attributes added by group_by()

```
library(dplyr)
orange_trees %>%
  group_by(Tree) %>%
  attributes()
```

```
$names
[1] "Tree"          "age"           "circumference"

$row.names
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
[19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35

$class
[1] "grouped_df" "tbl_df"     "tbl"        "data.frame"

$groups
# A tibble: 5 x 2
  Tree  .rows
  <ord> <list>
1 3     <int [7]>
2 1     <int [7]>
3 5     <int [7]>
4 2     <int [7]>
5 4     <int [7]>
```

Environments are like lists

```
datacamp_lst <- list(
  name = "DataCamp",
  founding_year = 2013,
  website = "https://www.datacamp.com"
)
```

```
ls.str(datacamp_lst)
```

```
founding_year :  num 2013
name :  chr "DataCamp"
website :  chr "https://www.datacamp.com"
```

Transforming a list to an env

```
datacamp_env <- list2env(datacamp_lst)
```

```
ls.str(datacamp_env)
```

```
founding_year :  num 2013
name :  chr "DataCamp"
website :  chr "https://www.datacamp.com"
```

Environments have parents

Getting the parent environment

```
parent <- parent.env(datacamp_env)
environmentName(parent)
```

```
"R_GlobalEnv"
```

```
grandparent <- parent.env(parent)
environmentName(grandparent)
```

```
"package:stats"
```

```
search()
```

```
[1] ".GlobalEnv"        "package:stats"
[3] "package:graphics"  "package:grDevices"
[5] "package:utils"     "package:datasets"
[7] "package:methods"   "Autoloads"
[9] "package:base"
```

Does a variable exist?

```
datacamp_lst <- list(
  name = "DataCamp",
  website = "https://www.datacamp.com"
)
datacamp_env <- list2env(datacamp_lst)
founding_year <- 2013
```

```
exists("founding_year", envir = datacamp_env)
```

```
TRUE
```

```
exists("founding_year", envir = datacamp_env, inherits = FALSE)
```

```
FALSE
```

Accessing variables outside functions

```
x_times_y <- function(x) {
  x * y
}
```

```
x_times_y(10)
```

```
Error in x_times_y(10) :
  object 'y' not found
```

```
x_times_y <- function(x) {
  x * y
}
y <- 4
```

```
x_times_y(10)
```

```
40
```