Funciones R

How to Write a Function

Calling functions

One way to make your code more readable is to be careful about the order you pass arguments when you call functions, and whether you pass the arguments by position or by name.

gold_medals, a numeric vector of the number of gold medals won by each country in the 2016 Summer Olympics, is provided.

For convenience, the arguments of median() and rank() are displayed using args(). Setting rank()'s na.last argument to "keep" means "keep the rank of NA values as NA".

Best practice for calling functions is to include them in the order shown by args(), and to only name rare arguments.

Instructions `100 XP`

- The final line calculates the median number of gold medals each country won.
- Rewrite the call to median(), following best practices.

E1.R

```
# Look at the gold medals data
gold_medals

# Note the arguments to median()
args(median)

# Rewrite this function call, following best practices
median(gold_medals, na.rm = TRUE)
```

The benefits of writing functions

There are lots of great reasons that you should write your own functions.

Which of these is not one of them?

Answer the question `50XP`

Possible Answers

- You can type less code, saving effort and making your analyses more readable. Respuesta
- You make less "copy and paste"-related errors.
- You can reuse your code from project to project.
- You can make your code harder to read, potentially improving your job security because only you can maintain it.

Your first function: tossing a coin

Time to write your first function! It's a really good idea when writing functions to start simple. You can always make a function more complicated later if it's really necessary, so let's not worry about arguments for now.

Instructions `100 XP`

- Simulate a single coin toss by using sample() to sample from coin_sides once.
- Write a template for your function, naming it toss_coin. The function should take no arguments. Don't include the body of the function yet.
- Copy your script, and paste it into the function body.
- Call your function.

E2.R

```
coin_sides <- c("head", "tail")

# Sample from coin_sides once
sample(coin_sides,1)

# Write a template for your function, toss_coin()
toss_coin <- function() {
```

```
  # (Leave the contents of the body for later)
# Add punctuation to finish the body
}

# Your script, from a previous step
coin_sides <- c("head", "tail")

# Paste your script into the function body
toss_coin <- function() {
  sample(coin_sides, 1)

}

# Your functions, from previous steps
toss_coin <- function() {
  coin_sides <- c("head", "tail")
  sample(coin_sides, 1)
}

# Call your function
toss_coin()
```

nputs to functions

Most functions require some sort of input to determine what to compute. The inputs to functions are called arguments. You specify them inside the parentheses after the word "function."

As mentioned in the video, the following exercises assume that you are using sample() to do random sampling.

Instructions `100 XP`

- Sample from coin_sides n_flips times with replacement.
- Update the definition of toss_coin() to accept a single argument, n_flips. The function should sample coin_sides n_flips times with replacement. Remember to change the signature and the body.
- Generate 10 coin flips.

E3.R

```
coin_sides <- c("head", "tail")
n_flips <- 10

# Sample from coin_sides n_flips times with replacement
sample(coin_sides,n_flips,replace = TRUE)

# Update the function to return n coin tosses
toss_coin <- function(n_flips) {
  coin_sides <- c("head", "tail")
  sample(coin_sides, n_flips,replace = TRUE)
}

# Generate 10 coin tosses
toss_coin(10)
```

Multiple inputs to functions

If a function should have more than one argument, list them in the function signature, separated by commas.

To solve this exercise, you need to know how to specify sampling weights to sample(). Set the prob argument to a numeric vector with the same length as x. Each value of prob is the probability of sampling the corresponding element of x, so their values add up to one. In the following example, each sample has a 20% chance of "bat", a 30% chance of "cat" and a 50% chance of "rat".

> sample(c("bat", "cat", "rat"), 10, replace = TRUE, prob = c(0.2, 0.3, 0.5))

##Instructions `100 XP`

- Bias the coin by weighting the sampling. Specify the prob argument so that heads are sampled with probability p_head (and tails are sampled with probability 1 - p_head).
- Update the definition of toss_coin() so it accepts an argument, p_head, and weights the samples using the code you wrote in the previous step.
- Generate 10 coin tosses with an 80% chance of each head.

E4.R

```
coin_sides <- c("head", "tail")
n_flips <- 10
```

```r
p_head <- 0.8

# Define a vector of weights
weights <- c(p_head, 1 - p_head)

# Update so that heads are sampled with prob p_head
sample(coin_sides, n_flips, replace = TRUE, prob = weights)

# Update the function so heads have probability p_head
toss_coin <- function(n_flips,p_head) {
  coin_sides <- c("head", "tail")
  # Define a vector of weights
  weights <- c(p_head,1-p_head)
  # Modify the sampling to be weighted
  sample(coin_sides, n_flips, replace = TRUE,prob=weights)
}

# Generate 10 coin tosses
toss_coin(10,0.8)
```

Renaming GLM

R's generalized linear regression function, glm(), suffers the same usability problems as lm(): its name is an acronym, and its formula and data arguments are in the wrong order.

To solve this exercise, you need to know two things about generalized linear regression:

glm() formulas are specified like lm() formulas: response is on the left, and explanatory variables are added on the right. To model count data, set glm()'s family argument to poisson, making it a Poisson regression. Here you'll use data on the number of yearly visits to Snake River at Jackson Hole, Wyoming, snake_river_visits.

Instructions `100 XP`

- Run a generalized linear regression by calling glm(). Model n_visits vs. gender, income, and travel on the snake_river_visits dataset, setting the family to poisson.
- Define a function, run_poisson_regression(), to run a Poisson regression. This should take two arguments: data and formula, and call glm(), passing those arguments and setting family to poisson.

- Recreate the Poisson regression model from the first step, this time by calling your run_poisson_regression() function.

E5.R

```
# Run a generalized linear regression
glm(
  # Model no. of visits vs. gender, income, travel
  n_visits ~ gender + income + travel,
  # Use the snake_river_visits dataset
  data = snake_river_visits,
  # Make it a Poisson regression
  family = poisson
)

# Write a function to run a Poisson regression
run_poisson_regression <- function(data, formula) {
    glm(formula, data, family = poisson)
}

# From previous step
run_poisson_regression <- function(data, formula) {
  glm(formula, data, family = poisson)
}

# Re-run the Poisson regression, using your function
model <- snake_river_visits %>%
  run_poisson_regression(n_visits ~ gender + income + travel)

# Run this to see the predictions
snake_river_explanatory %>%
  mutate(predicted_n_visits = predict(model, ., type = "response"))%>%
  arrange(desc(predicted_n_visits))
```

Numeric defaults

cut_by_quantile() converts a numeric vector into a categorical variable where quantiles define the cut points. This is a useful function, but at the moment you have to specify five arguments to make it work. This is too much thinking and typing.

By specifying default arguments, you can make it easier to use. Let's start with n, which specifies how many categories to cut x into.

A numeric vector of the number of visits to Snake River is provided as n_visits.

Instructions 100 XP

Update the definition of cut_by_quantile() so that the n argument defaults to 5. Remove the n argument from the call to cut_by_quantile().

E6.R

```
# Set the default for n to 5
cut_by_quantile <- function(x, n=5, na.rm, labels, 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)
}

# Remove the n argument from the call
cut_by_quantile(
  n_visits,
  na.rm = FALSE,
  labels = c("very low", "low", "medium", "high", "very high"),
  interval_type = "(lo, hi]"
)
formals(cut_by_quantile)
```

Logical defaults

cut_by_quantile() is now slightly easier to use, but you still always have to specify the na.rm argument. This removes missing values—it behaves the same as the na.rm argument to mean() or sd().

Where functions have an argument for removing missing values, the best practice is to not remove them by default (in case you hadn't spotted that you had missing values). That means that the default for na.rm should be FALSE.

Instructions 100 XP

Update the definition of cut_by_quantile() so that the na.rm argument defaults to FALSE. Remove the na.rm argument from the call to cut_by_quantile().

E7.R

```r
# Set the default for na.rm to FALSE
cut_by_quantile <- function(x, n = 5, na.rm = FALSE, labels, 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)
}

# Remove the na.rm argument from the call
cut_by_quantile(
  n_visits,
  labels = c("very low", "low", "medium", "high", "very high"),
  interval_type = "(lo, hi]"
)
```

All About Arguments

NULL defaults

The cut() function used by cut_by_quantile() can automatically provide sensible labels for each category. The code to generate these labels is pretty complicated, so rather than appearing in the function signature directly, its labels argument defaults to NULL, and the calculation details are shown on the ?cut help page.

Instructions `100 XP`

Update the definition of cut_by_quantile() so that the labels argument defaults to NULL. Remove the labels argument from the call to cut_by_quantile().

E8.R

```r
# Set the default for labels to NULL
cut_by_quantile <- function(x, n = 5, na.rm = FALSE, labels=NULL,
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)
}
```

```r
# Remove the labels argument from the call
cut_by_quantile(
  n_visits,
  interval_type = "(lo, hi]"
)
```

Categorical defaults

When cutting up a numeric vector, you need to worry about what happens if a value lands exactly on a boundary. You can either put this value into a category of the lower interval or the higher interval. That is, you can choose your intervals to include values at the top boundary but not the bottom (in mathematical terminology, "open on the left, closed on the right", or (lo, hi]). Or you can choose the opposite ("closed on the left, open on the right", or [lo, hi)). cut_by_quantile() should allow these two choices.

The pattern for categorical defaults is:

function(cat_arg = c("choice1", "choice2")) { cat_arg <- match.arg(cat_arg) }

Free hint: In the console, type head(rank) to see the start of rank()'s definition, and look at the ties.method argument.

Instructions `100 XP`

Update the signature of cut_by_quantile() so that the interval_type argument can be "(lo, hi]" or "[lo, hi)". Note the space after each comma. Update the body of cut_by_quantile() to match the interval_type argument. Remove the interval_type argument from the call to cut_by_quantile().

E9.R

```r
# 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)
}
```

```
# Remove the interval_type argument from the call
cut_by_quantile(n_visits)
```

Harmonic mean

The harmonic mean is the reciprocal of the arithmetic mean of the reciprocal of the data. That is

The harmonic mean is often used to average ratio data. You'll be using it on the price/earnings ratio of stocks in the Standard and Poor's 500 index, provided as std_and_poor500. Price/earnings ratio is a measure of how expensive a stock is.

The dplyr package is loaded.

Instructions `100 XP`

- Look at std_and_poor500 (you'll need this later). Write a function, get_reciprocal, to get the reciprocal of an input x. Its only argument should be x, and it should return one over x.
- Write a function, calc_harmonic_mean(), that calculates the harmonic mean of its only input, x.
- Using std_and_poor500, group by sector, and summarize to calculate the harmonic mean of the price/earning ratios in the pe_ratio column.

E10.R

```
# Look at the Standard and Poor 500 data
glimpse(std_and_poor500)

# Write a function to calculate the reciprocal
get_reciprocal <- function(x){
  1/x
}

# From previous step
get_reciprocal <- function(x) {
  1 / x
}

# Write a function to calculate the harmonic mean
```

```r
calc_harmonic_mean <- function(x) {
  x %>%
    get_reciprocal() %>%
    mean() %>%
    get_reciprocal()
}

# From previous steps
get_reciprocal <- function(x) {
  1 / x
}
calc_harmonic_mean <- function(x) {
  x %>%
    get_reciprocal() %>%
    mean() %>%
    get_reciprocal()
}

std_and_poor500 %>%
  # Group by sector
  group_by(sector) %>%
  # Summarize, calculating harmonic mean of P/E ratio
  summarize(hmean_pe_ratio = calc_harmonic_mean(pe_ratio))
```

Dealing with missing values

In the last exercise, many sectors had an NA value for the harmonic mean. It would be useful for your function to be able to remove missing values before calculating.

Rather than writing your own code for this, you can outsource this functionality to mean().

The dplyr package is loaded.

Instructions `100 XP`

- Modify the signature and body of calc_harmonic_mean() so it has an na.rm argument, defaulting to false, that gets passed to mean().
- Using std_and_poor500, group by sector, and summarize to calculate the harmonic mean of the price/earning ratios in the pe_ratio column, removing missing values.

E11.R

```
# Add an na.rm arg with a default, and pass it to mean()
calc_harmonic_mean <- function(x,na.rm = FALSE) {
  x %>%
    get_reciprocal() %>%
    mean(na.rm =na.rm) %>%
    get_reciprocal()
}

# From previous step
calc_harmonic_mean <- function(x, na.rm = FALSE) {
  x %>%
    get_reciprocal() %>%
    mean(na.rm = na.rm) %>%
    get_reciprocal()
}

std_and_poor500 %>%
  # Group by sector
  group_by(sector) %>%
  # Summarize, calculating harmonic mean of P/E ratio
  summarize(hmean_pe_ratio = calc_harmonic_mean(pe_ratio,
   na.rm = TRUE))
```

Passing arguments with ...

Rather than explicitly giving calc_harmonic_mean() and na.rm argument, you can use … to simply "pass other arguments" to mean().

The dplyr package is loaded.

Instructions 100 XP

- Replace the na.rm argument with … in the signature and body of calc_harmonic_mean().
- Using std_and_poor500, group by sector, and summarize to calculate the harmonic mean of the price/earning ratios in the pe_ratio column, removing missing values.

E12.R

```r
# Swap na.rm arg for ... in signature and body
calc_harmonic_mean <- function(x, ...) {
  x %>%
    get_reciprocal() %>%
    mean(...) %>%
    get_reciprocal()
}
calc_harmonic_mean(x = c(1, NA, 3, NA, 5))

calc_harmonic_mean <- function(x, ...) {
  x %>%
    get_reciprocal() %>%
    mean(...) %>%
    get_reciprocal()
}

std_and_poor500 %>%
  # Group by sector
  group_by(sector) %>%
  # Summarize, calculating harmonic mean of P/E ratio
  summarize(hmean_pe_ratio = calc_harmonic_mean(pe_ratio,
   na.rm = TRUE))
```

Throwing errors with bad arguments

If a user provides a bad input to a function, the best course of action is to throw an error letting them know. The two rules are

Throw the error message as soon as you realize there is a problem (typically at the start of the function). Make the error message easily understandable. You can use the assert_*() functions from assertive to check inputs and throw errors when they fail.

Instructions `100 XP`

- Add a line to the body of calc_harmonic_mean() to assert that x is numeric.
- Look at what happens when you pass a character argument to calc_harmonic_mean().

E13.R

```
alc_harmonic_mean <- function(x, na.rm = FALSE) {
  # Assert that x is numeric
  assert_is_numeric(x)
  x %>%
    get_reciprocal() %>%
    mean(na.rm = na.rm) %>%
    get_reciprocal()
}

# See what happens when you pass it strings
calc_harmonic_mean(std_and_poor500$sector)
```

Custom error logic

Sometimes the `assert_*()` functions in assertive don't give the most informative error message. For example, the assertions that check if a number is in a numeric range will tell the user that a value is out of range, but the won't say why that's a problem. In that case, you can use the is_*() functions in conjunction with messages, warnings, or errors to define custom feedback.

The harmonic mean only makes sense when x has all positive values. (Try calculating the harmonic mean of one and minus one to see why.) Make sure your users know this!

Instructions `100 XP`

- If any values of x are non-positive (ignoring NAs) then throw an error.
- Look at what happens when you pass a character argument to calc_harmonic_mean().

E14.R

```
calc_harmonic_mean <- function(x, na.rm = FALSE) {
  assert_is_numeric(x)
  # Check if any values of x are non-positive
  if(any(is_non_positive(x), na.rm = TRUE)) {
    # Throw an error
    stop("x contains non-positive values, so the harmonic mean makes no sense.")
  }
  x %>%
    get_reciprocal() %>%
    mean(na.rm = na.rm) %>%
    get_reciprocal()
}
```

```
# See what happens when you pass it negative numbers
calc_harmonic_mean(std_and_poor500$pe_ratio - 20)
```

Fixing function arguments

The harmonic mean function is almost complete. However, you still need to provide some checks on the na.rm argument. This time, rather than throwing errors when the input is in an incorrect form, you are going to try to fix it.

na.rm should be a logical vector with one element (that is, TRUE, or FALSE).

The assertive package is loaded for you.

Instructions `100 XP`

- Update calc_harmonic_mean() to fix the na.rm argument by using use_first() to select the * first na.rm element, and coerce_to() to change it to logical.

E15.R

```
# Update the function definition to fix the na.rm argument
calc_harmonic_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 harmonic mean makes no sense.")
  }
  # Use the first value of na.rm, and coerce to logical
  na.rm <- coerce_to(use_first(na.rm), target_class  = "logical")
  x %>%
    get_reciprocal() %>%
    mean(na.rm = na.rm) %>%
    get_reciprocal()
}

# See what happens when you pass it malformed na.rm
calc_harmonic_mean(std_and_poor500$pe_ratio, na.rm = 1:5)
```

Return Values and Scope

Returning early

Sometimes, you don't need to run through the whole body of a function to get the answer. In that case you can return early from that function using return().

To check if x is divisible by n, you can use is_divisible_by(x, n) from assertive.

Alternatively, use the modulo operator, %%. x %% n gives the remainder when dividing x by n, so x %% n == 0 determines whether x is divisible by n. Try 1:10 %% 3 == 0 in the console.

To solve this exercise, you need to know that a leap year is every 400th year (like the year 2000) or every 4th year that isn't a century (like 1904 but not 1900 or 1905).

assertive is loaded.

Instructions `100 XP`

Complete the definition of is_leap_year(), checking for the cases of year being divisible by 400, then 100, then 4, returning early from the function in each case.

E16.R

```
is_leap_year <- function(year) {
  # If year is div. by 400 return TRUE
  if(is_divisible_by(year,400)) {
    return(TRUE)
  }
  # If year is div. by 100 return FALSE
  if(is_divisible_by(year,100)) {
    return(FALSE)
  }
  # If year is div. by 4 return TRUE
  if(is_divisible_by(year,4)) {
    return(TRUE)
  }

  # Otherwise return FALSE
  FALSE
}
```

Returning invisibly

When the main purpose of a function is to generate output, like drawing a plot or printing something in the console, you may not want a return value to be rinted as well. In that case, the value should be invisibly returned.

The base R plot function returns NULL, since its main purpose is to draw a plot. This isn't helpful if you want to use it in piped code: instead it should invisibly return the plot data to be piped on to the next step.

Recall that plot() has a formula interface: instead of giving it vectors for x and y, you can specify a formula describing which columns of a data frame go on the x and y axes, and a data argument for the data frame. Note that just like lm(), the arguments are the wrong way round because the detail argument, formula, comes before the data argument.

plot(y ~ x, data = data)

Instructions 100 XP

- Use the cars dataset and the formula interface to plot(), draw a scatter plot of dist versus speed.
- Give pipeable_plot() data and formula arguments (in that order) and make it draw the plot, then invisibly return data.
- Draw the scatter plot of dist vs. speed again by calling pipeable_plot()

E17.R

```
# Using cars, draw a scatter plot of dist vs. speed
plt_dist_vs_speed <- plot(dist ~ speed, data = cars)

# Oh no! The plot object is NULL
plt_dist_vs_speed

# Define a pipeable plot fn with data and formula args
pipeable_plot <- function(data, formula) {
  # Call plot() with the formula interface
  plot(formula, data)
  # Invisibly return the input dataset
  invisible(data)
}

# Draw the scatter plot of dist vs. speed again
```

```
plt_dist_vs_speed <- cars %>%
  pipeable_plot(dist ~ speed)
```

Returning many things

Functions can only return one value. If you want to return multiple things, then you can store them all in a list.

If users want to have the list items as separate variables, they can assign each list element to its own variable using zeallot's multi-assignment operator, %<-%.

glance(), tidy(), and augment() each take the model object as their only argument.

The Poisson regression model of Snake River visits is available as model. broom and zeallot are loaded.

Instructions `100 XP`

- Examine the structure of model.
- Use broom functions on model to create a list containing the model-, coefficient-, and observation-level parts of model.
- Wrap the code into a function, groom_model(), that accepts model as its only argument.
- Call groom_model() on model, multi-assigning the result to three variables at once: mdl, cff, and obs.

E18.R

```
# Look at the structure of model (it's a mess!)
str(model)

# Use broom tools to get a list of 3 data frames
list(
  # Get model-level values
  model = glance(model),
  # Get coefficient-level values
  coefficients = tidy(model),
  # Get observation-level values
  observations = augment(model)
)
```

```r
# Wrap this code into a function, groom_model
groom_model <- function(model){
  list(
    model = glance(model),
    coefficients = tidy(model),
    observations = augment(model)
  )
}

groom_model(model)

# From previous step
groom_model <- function(model) {
  list(
    model = glance(model),
    coefficients = tidy(model),
    observations = augment(model)
  )
}

# Call groom_model on model, assigning to 3 variables
c(mdl, cff, obs) %<-% groom_model(model)
#c(var1, var2, var3) %<-% fn(args)

# See these individual variables
mdl; cff; obs
```

Returning metadata

Sometimes you want to return multiple things from a function, but you want the result to have a particular class (for example, a data frame or a numeric vector), so returning a list isn't appropriate. This is common when you have a result plus metadata about the result. (Metadata is "data about the data". For example, it could be the file a dataset was loaded from, or the username of the person who created the variable, or the number of iterations for an algorithm to converge.)

In that case, you can store the metadata in attributes. Recall the syntax for assigning attributes is as follows.

attr(object, "attribute_name") <- attribute_value

Instructions 100 XP

- Update pipeable_plot() so the result has an attribute named "formula" with the value of formula.
- plt_dist_vs_speed, that you previously created, is shown. Examine its updated structure.

E19.R

```
pipeable_plot <- function(data, formula) {
  plot(formula, data)
  # Add a "formula" attribute to data
  attr(data, "formula") <- formula
  invisible(data)
}

# From previous exercise
plt_dist_vs_speed <- cars %>%
  pipeable_plot(dist ~ speed)

# Examine the structure of the result
str(plt_dist_vs_speed)
```

Creating and exploring environments

Environments are used to store other variables. Mostly, you can think of them as lists, but there's an important extra property that is relevant to writing functions. Every environment has a parent environment (except the empty environment, at the root of the environment tree). This determines which variables R know about at different places in your code.

Facts about the Republic of South Africa are contained in capitals, national_parks, and population.

Instructions 100 XP

- Create rsa_lst, a named list from capitals, national_parks, and population. Use those values as the names.
- List the structure of each element of rsa_lst using ls.str().
- Convert the list to an environment, rsa_env, using list2env().
- List the structure of each element of rsa_env

- Find the parent environment of rsa_env and print its name.

E20.R

```r
# Add capitals, national_parks, & population to a named list
rsa_lst <- list(
  capitals = capitals,
  national_parks = national_parks,
  population = population
)

# List the structure of each element of rsa_lst
ls.str(rsa_lst)

# From previous step
rsa_lst <- list(
  capitals = capitals,
  national_parks = national_parks,
  population = population
)

# Convert the list to an environment
rsa_env <- list2env(rsa_lst)

# List the structure of each variable
ls.str(rsa_env)

# From previous steps
rsa_lst <- list(
  capitals = capitals,
  national_parks = national_parks,
  population = population
)
rsa_env <- list2env(rsa_lst)

# Find the parent environment of rsa_env
parent <- parent.env(rsa_env)
environmentName(parent)

# Print its name
print(environmentName)
```

Do variables exist?

If R cannot find a variable in the current environment, it will look in the parent environment, then the grandparent environment, and so on until it finds it.

rsa_env has been modified so it includes capitals and national_parks, but not population.

Instructions `100 XP`

- Check if population exists in rsa_env, using default inheritance rules.
- Check if population exists in rsa_env, ignoring inheritance.

E21.R

```
# Compare the contents of the global environment and rsa_env
ls.str(globalenv())
ls.str(rsa_env)

# Does population exist in rsa_env?
exists("population", envir = rsa_env)

# Does population exist in rsa_env, ignoring inheritance?
exists("population", envir = rsa_env,inherits = FALSE)
```

Variable precedence 1

Consider this code, run in a fresh R session.

x_plus_y <- function(x) { y <- 3 x + y } y <- 7

If you now call x_plus_y(5), what is the result?

Answer the question `50XP`

Possible Answers

8. Respuesta
9.
10.

An error is thrown.

Variable precedence 2

Consider this (slightly different) code, run in a fresh R session.

x_plus_y <- function(x) { x <- 6 y <- 3 x + y } y <- 7 If you now call x_plus_y(5), what is the result?

Answer the question 50XP

Possible Answers

8.
9. respuesta
10.
11. An error is thrown.

Case Study on Grain Yields

Converting areas to metric 1

In this chapter, you'll be working with grain yield data from the United States Department of Agriculture, National Agricultural Statistics Service. Unfortunately, they report all areas in acres. So, the first thing you need to do is write some utility functions to convert areas in acres to areas in hectares.

To solve this exercise, you need to know the following:

There are 4840 square yards in an acre. There are 36 inches in a yard and one inch is 0.0254 meters. There are 10000 square meters in a hectare.

Instructions 100 XP

- Write a function, acres_to_sq_yards(), to convert areas in acres to areas in square yards. This should take a single argument, acres.
- Write a function, yards_to_meters(), to convert distances in yards to distances in meters. This should take a single argument, yards.
- Write a function, sq_meters_to_hectares(), to convert areas in square meters to areas in hectares. This should take a single argument, sq_meters.

E22.R

```
# Write a function to convert acres to sq. yards
acres_to_sq_yards <- function(x) {
  x * 4840
}


# Write a function to convert yards to meters
yards_to_meters <- function(x) {
  x * 36*0.0254
}

# Write a function to convert sq. meters to hectares
sq_meters_to_hectares <- function(sq_meters) {
  (sq_meters)/10000
}
```

Converting areas to metric 2

You're almost there with creating a function to convert acres to hectares. You need another utility function to deal with getting from square yards to square meters. Then, you can bring everything together to write the overall acres-to-hectares conversion function. Finally, in the next exercise you'll be calculating area conversions in the denominator of a ratio, so you'll need a harmonic acre-to-hectare conversion function.

Free hints: magrittr's raise_to_power() will be useful here. The last step is similar to Chapter 2's Harmonic Mean.

The three utility functions from the last exercise (acres_to_sq_yards(), yards_to_meters(), and sq_meters_to_hectares()) are available, as is your get_reciprocal() from Chapter 2. magrittr is loaded.

Instructions `100 XP`

- Write a function to convert distance in square yards to square meters. It should take the square root of the input, then convert yards to meters, then square the result.
- Write a function to convert areas in acres to hectares. The function should convert the input from acres to square yards, then to square meters, then to hectares.
- Write a function to harmonically convert areas in acres to hectares. The function should get the reciprocal of the input, then convert from acres to hectares, then get the reciprocal again.

E23.R

```
# Write a function to convert sq. yards to sq. meters
sq_yards_to_sq_meters <- function(sq_yards) {
  sq_yards %>%
    # Take the square root
    sqrt() %>%
    # Convert yards to meters
    yards_to_meters() %>%
    # Square it
    raise_to_power(2)
     }

# Load the function from the previous step
load_step2()

# Write a function to convert acres to hectares
acres_to_hectares <- function(acres) {
  acres %>%
    # Convert acres to sq yards
    acres_to_sq_yards() %>%
    # Convert sq yards to sq meters
    sq_yards_to_sq_meters() %>%
    # Convert sq meters to hectares
    sq_meters_to_hectares()
}

# Load the functions from the previous steps
load_step3()

# Define a harmonic acres to hectares function
harmonic_acres_to_hectares <- function(acres) {
  acres %>%
    # Get the reciprocal
    get_reciprocal() %>%
    # Convert acres to hectares
    acres_to_hectares() %>%
    # Get the reciprocal again
    get_reciprocal()
}
```

Converting yields to metric

The yields in the NASS corn data are also given in US units, namely bushels per acre. You'll need to write some more utility functions to convert this unit to the metric unit of kg per hectare.

Bushels historically meant a volume of 8 gallons, but in the context of grain, they are now defined as masses. This mass differs for each grain! To solve this exercise, you need to know these facts.

One pound (lb) is 0.45359237 kilograms (kg). One bushel is 48 lbs of barley, 56 lbs of corn, or 60 lbs of wheat. magrittr is loaded.

Instructions `100 XP`

- Write a function to convert masses in lb to kg. This should take a single argument, lbs.
- Write a function to convert masses in bushels to lbs. This should take two arguments, bushels and crop. It should define a lookup vector of scale factors for each crop (barley, corn, wheat), extract the scale factor for the crop, then multiply this by the number of bushels.
- Write a function to convert masses in bushels to kgs. This should take two arguments, bushels and crop. It should convert the mass in bushels to lbs then to kgs.
- Write a function to convert yields in bushels/acre to kg/ha. The arguments should be bushels_per_acre and crop. Three choices of crop should be allowed: "barley", "corn", and "wheat". It should match the crop argument, then convert bushels to kgs, then convert harmonic acres to hectares.

E24.R

```
# Write a function to convert lb to kg
lbs_to_kgs <- function(lbs) {lbs * 0.45359237}

# Write a function to convert bushels to lbs
bushels_to_lbs <- function(bushels, crop) {
  # Define a lookup table of scale factors
  c(barley = 48, corn = 56, wheat = 60) %>%
    # Extract the value for the crop
    extract(crop) %>%
    # Multiply by the no. of bushels
    multiply_by(bushels)
}
```

```r
# Load fns defined in previous steps
load_step3()

# Write a function to convert bushels to kg
bushels_to_kgs <- function(bushels, crop) {
  bushels %>%
    # Convert bushels to lbs for this crop
    bushels_to_lbs(crop) %>%
    # Convert lbs to kgs
    lbs_to_kgs()
}

# Load fns defined in previous steps
load_step4()

# Write a function to convert bushels/acre to kg/ha
bushels_per_acre_to_kgs_per_hectare <- function(bushels_per_acre,
crop = c("barley", "corn", "wheat")) {
  # Match the crop argument
  crop <- match.arg(crop)
  bushels_per_acre %>%
    # Convert bushels to kgs for this crop
    bushels_to_kgs(crop) %>%
    # Convert harmonic acres to ha
    harmonic_acres_to_hectares()
}
```

Applying the unit conversion

Now that you've written some functions, it's time to apply them! The NASS corn dataset is available, and you can fortify it (jargon for "adding new columns") with metrics areas and yields.

This fortification process can also be turned into a function, so you'll define a function for this, and test it on the NASS wheat dataset.

Instructions `100 XP`

- Look at the corn dataset. Add two columns: farmed_area_ha should be farmed_area_acres converted to hectares; yield_kg_per_ha should be yield_bushels_per_acre converted to kilograms per hectare.

- Wrap the mutation code into a function called fortify_with_metric_units that takes two arguments, data and crop with no defaults. (In the function body, swap corn for the data argument and pass the function's local crop variable to the crop argument.)
- Use fortify_with_metric_units() on the wheat dataset.

E25.R

```
# View the corn dataset
glimpse(corn)

corn %>%
  # Add some columns
  mutate(
    # Convert farmed area from acres to ha
    farmed_area_ha = acres_to_hectares(farmed_area_acres),
    # Convert yield from bushels/acre to kg/ha
    yield_kg_per_ha = bushels_per_acre_to_kgs_per_hectare(
      yield_bushels_per_acre,
      crop = "corn"
    )
  )

# Wrap this code into a function
fortify_with_metric_units <- function(data, crop) {
  data %>%
    mutate(
      farmed_area_ha = acres_to_hectares(farmed_area_acres),
      yield_kg_per_ha = bushels_per_acre_to_kgs_per_hectare(
        yield_bushels_per_acre,
        crop = crop
      )
    )
}

# Try it on the wheat dataset
fortify_with_metric_units(wheat, crop = "wheat")
```

Plotting yields over time

Now that the units have been dealt with, it's time to explore the datasets. An obvious question to ask about each crop is, "how do the yields change over time in each US state?" Let's draw

a line plot to find out!

ggplot2 is loaded, and corn and wheat datasets are available with metric units.

Instructions `100 XP`

- Using the corn dataset, plot yield_kg_per_ha versus year. Add a line layer grouped by state and a smooth trend layer.
- Turn the plotting code into a function, plot_yield_vs_year(). This should accept a single argument, data.

E26.R

```
# Using corn, plot yield (kg/ha) vs. year
ggplot(corn, aes(year, yield_kg_per_ha)) +
  # Add a line layer, grouped by state
  geom_line(aes(group = state)) +
  # Add a smooth trend layer
  geom_smooth()

# Wrap this plotting code into a function
plot_yield_vs_year <- function(data) {
  ggplot(data, aes(year, yield_kg_per_ha)) +
    geom_line(aes(group = state)) +
    geom_smooth()
}

# Test it on the wheat dataset
plot_yield_vs_year(wheat)
```

A nation divided

The USA has a varied climate, so we might expect yields to differ between states. Rather than trying to reason about 50 states separately, we can use the USA Census Regions to get 9 groups.

The "Corn Belt", where most US corn is grown is in the "West North Central" and "East North Central" regions. The "Wheat Belt" is in the "West South Central" region.

dplyr is loaded, the corn and wheat datasets are available, as is usa_census_regions.

Instructions 100 XP

- Inner join corn to usa_census_regions by "state".
- Turn the code into a function, fortify_with_census_region(). This should accept a single argument, data.

E27.R

```
# Inner join the corn dataset to usa_census_regions by state
corn %>%
  inner_join(usa_census_regions, by = "state")

# Wrap this code into a function
fortify_with_census_region <- function(data){
  data %>%
    inner_join(usa_census_regions, by = "state")
}

# Try it on the wheat dataset
fortify_with_census_region(wheat)
```

Plotting yields over time by region

So far, you have used a function to plot yields over time for each crop, and you've added a census_region column to the crop datasets. Now you are ready to look at how the yields change over time in each region of the USA.

ggplot2 is loaded. corn and wheat have been fortified with census regions. plot_yield_vs_year() is available.

Instructions 100 XP

- Use the function you wrote to plot yield versus year for the corn dataset, then facet the plot, wrapped by census_region.
- Turn the code into a function, plot_yield_vs_year_by_region(), that should take a single argument, data.

E28.R

```
# Plot yield vs. year for the corn dataset
plot_yield_vs_year(corn) +
```

```
  # Facet, wrapped by census region
  facet_wrap(vars(census_region))

# Wrap this code into a function
plot_yield_vs_year_by_region <- function(data) {

  plot_yield_vs_year(data) +
    facet_wrap(vars(census_region))
}

# Try it on the wheat dataset
plot_yield_vs_year_by_region(wheat)
```

Running a model

The smooth trend line you saw in the plots of yield over time use a generalized additive model (GAM) to determine where the line should lie. This sort of model is ideal for fitting nonlinear curves. So we can make predictions about future yields, let's explicitly run the model. The syntax for running this GAM takes the following form.

gam(response ~ s(explanatory_var1) + explanatory_var2, data = dataset)

Here, s() means "make the variable smooth", where smooth very roughly means nonlinear.

mgcv and dplyr are loaded; the corn and wheat datasets are available.

Instructions 100 XP

- Run a GAM of yield_kg_per_ha versus smoothed year and census region, using the corn dataset.
- Wrap the modeling code into a function, run_gam_yield_vs_year_by_region. This should take a single argument, data, with no default.

E29.R

```
# Run a generalized additive model of yield vs. smoothed year and census region

gam(yield_kg_per_ha ~ s(year) + census_region, data = corn)

# Wrap the model code into a function
run_gam_yield_vs_year_by_region <- function(data){
```

```
  gam(yield_kg_per_ha ~ s(year) + census_region, data = corn)
}

# Try it on the wheat dataset
run_gam_yield_vs_year_by_region(wheat)
```

Making yield predictions

The fun part of modeling is using the models to make predictions. You can do this using a call to predict(), in the following form.

predict(model, cases_to_predict, type = "response") mgcv and dplyr are loaded; GAMs of the corn and wheat datasets are available as corn_model and wheat_model. A character vector of census regions is stored as census_regions.

Instructions `100 XP`

- In predict_this, set the prediction year to 2050.
- Predict the yield using corn_model and the cases specified in predict_this.
- Mutate predict_this to add the prediction vector as a new column named pred_yield_kg_per_ha.
- Wrap the script into a function, predict_yields. It should take two arguments, model and year, with no defaults. Remember to update 2050 to the year argument. Try predict_yields() on wheat_model with year set to 2050.

E30.R

```
# Make predictions in 2050
predict_this <- data.frame(
  year = 2050,
  census_region = census_regions
)

# Predict the yield
pred_yield_kg_per_ha <- predict(corn_model, predict_this, type = "response")

predict_this %>%
  # Add the prediction as a column of predict_this
    mutate(pred_yield_kg_per_ha = pred_yield_kg_per_ha)

# Wrap this prediction code into a function
```

```
predict_yields <- function(model, year) {
  predict_this <- data.frame(
    year = 2050,
    census_region = census_regions
  )
  pred_yield_kg_per_ha <- predict(model, predict_this, type = "response")
  predict_this %>%
    mutate(pred_yield_kg_per_ha = pred_yield_kg_per_ha)
}

# Try it on the wheat dataset
predict_yields(wheat_model,2050)
```

Do it all over again

Hopefully, by now, you've realized that the real benefit to writing functions is that you can reuse your code easily. Now you are going to rerun the whole analysis from this chapter on a new crop, barley. Since all the infrastructure is in place, that's less effort than it sounds!

Barley prefers a cooler climate compared to corn and wheat and is commonly grown in the US mountain states of Idaho and Montana.

dplyr and ggplot2, and mgcv are loaded; fortify_with_metric_units(), fortify_with_census_region(), plot_yield_vs_year_by_region(), run_gam_yield_vs_year_by_region(), and predict_yields() are available.

Instructions `100 XP`

- Fortify the barley data with metric units, then with census regions.
- Using the fortified barley data, plot the yield versus year by census region.
- Using the fortified barley data, run a GAM of yield versus year by census region, then predict the yields in 2050.

E31.R

```
fortified_barley <- barley %>%
  # Fortify with metric units
    fortify_with_metric_units() %>%
  # Fortify with census regions
  fortify_with_census_region()
```

```r
# See the result
glimpse(fortified_barley)

# From previous step
fortified_barley <- barley %>%
  fortify_with_metric_units() %>%
  fortify_with_census_region()

# Plot yield vs. year by region
plot_yield_vs_year_by_region(fortified_barley)

# From previous step
fortified_barley <- barley %>%
  fortify_with_metric_units() %>%
  fortify_with_census_region()

fortified_barley %>%
  # Run a GAM of yield vs. year by region
  run_gam_yield_vs_year_by_region()  %>%
  # Make predictions of yields in 2050
  predict_yields(2050)
```