Introduction to Data Science

Session 3: Functions, iteration, and debugging

Simon Munzert
Hertie School | GRAD-C11/E1339

Table of contents

Functions

Tidy programming basics

"Tidy programming" is not a strictly defined practice in the tidyverse. However, there are some common programming strategies that help you keep your code and workflow tidy. These include:

- Pipes (you already learned how to use them ✅)
- User-generated functions
- Functional programming with `purrr`

The latter two are extremely helpful - in particular when you are confronted with iterative tasks.

We will now learn the basics of creating your own functions and functional programming with R. There is much more to learn about these topics, so we will revisit them as the course progresses.

Functional programming

R is a functional programming (FP) language. As Hadley Wickham puts it in Advanced R:

> This means that it provides many tools for the creation and manipulation of functions. In particular, R has what's known as first-class functions. You can do anything with functions that you can do with vectors: you can assign them to variables, store them in lists, pass them as arguments to other functions, create them inside functions, and even return them as the result of a function.

R encourages you to use and build your own functions to solve problems. Often, this implies decomposing a large problem into small pieces, and solving each of them with independent functions.

There is much more to learn about functions and functional programming. Useful resources include:

- The chapter on functions in R for Data Science.
- The section on functional programming in Advanced R.
- The R packages book. In a way, bundling functions in a package is sometimes the next logical step.

Creating functions

Why creating functions?

That's a legit question. There are 21,000+ **packages** on CRAN (and many, many more on GitHub and other repositories) containing zillions of functions. Why should you create yet another one?

- Every data science project is unique. There are problems only you have to solve.
- For problems that are repetitive, you'll quickly look for options to automate the task.
- Functions are a great way to automate.

Examples where creating functions makes sense

1. You want to scrape thousands of websites. This implies multiple steps, from downloading to parsing and cleaning. All these steps can be achieved with existing functions, but the fine-tuning is specific to the set of websites. You build one (or a set of) scraping functions that take the websites as input and return a cleaned data frame ready to be analyzed.

2. You want to estimate not one but multiple models on your dataset. The models vary both in terms of data input and specification. Again, based on existing modeling functions you tailor your own, allowing you to run all these models automatically and to parse the results into one clean data frame.

Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func ← function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

[1] Yes, a function to create functions. 🤯

Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func ← function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

- We write functions to apply them later. So, we have to give them a name. Here, we name it "`my_func`".
- Also, our function (almost) always needs input, plus we want to specify how exactly the function should behave. We can use arguments for this, which are specified as arguments of the `function()` function.

[1] Yes, a function to create functions. 🤯

Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func ← function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

- Next, we specify anything we want the function to do.
- This comes in between curly brackets, `{ ... }`.
- Importantly, we can recycle arguments by calling them by their name.

[1] Yes, a function to create functions. 🤯

Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func ← function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

- Finally, we specify what the function should return.
- This could be a list, data.frame, vector, sentence - or anything else really.
- Note that R automatically returns the final object that is written (not: assigned!) in your function by default. Still, my recommendation is that you get into the habit of assigning the return object(s) explicitly with `return()`.

[1] Yes, a function to create functions. 🤯

Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func ← function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

- Oh, and don't forget to close the curly brackets...

[1] Yes, a function to create functions. 🤯

Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func ← function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

Let's try it out with a simple example function - one that converts temperatures from Fahrenheit to Celsius:[2]

```
R> fahrenheit_to_celsius ← function(temp_F) {
+   temp_C ← (temp_F - 32) * (5/9)
+   return(temp_C)
+ }
```

[1] Yes, a function to create functions. 🤯

[2] Courtesy of Software Carpentry.

Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func ← function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

Let's try it out with a simple example function - one that converts temperatures from Fahrenheit to Celsius:[2]

```
R> fahrenheit_to_celsius ← function(temp_F) {
+   temp_C ← (temp_F - 32) * (5/9)
+   return(temp_C)
+ }
```

- Our function has an intuitive name.
- Also, it takes just one thing as input, which we call `temp_F`.

[1] Yes, a function to create functions. 🤯

[2] Courtesy of Software Carpentry.

Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func ← function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

Let's try it out with a simple example function - one that converts temperatures from Fahrenheit to Celsius:[2]

```
R> fahrenheit_to_celsius ← function(temp_F) {
+   temp_C ← (temp_F - 32) * (5/9)
+   return(temp_C)
+ }
```

- We now take up the argument `temp_F`, do something with it, and store the output in a new object, `temp_C`.
- Importantly, that object only lives within the function. When the function is run, we cannot access it from the environment.

[1] Yes, a function to create functions. 🤯

[2] Courtesy of Software Carpentry.

Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func ← function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

Let's try it out with a simple example function - one that converts temperatures from Fahrenheit to Celsius:[2]

```
R> fahrenheit_to_celsius ← function(temp_F) {
+   temp_C ← (temp_F - 32) * (5/9)
+   return(temp_C)
+ }
```

- Finally, the output is returned.

[1] Yes, a function to create functions. 🤯

[2] Courtesy of Software Carpentry.

Basic syntax

Writing your own function in R is easy with the `function()` function[1]. The basic syntax is as follows:

```
R> my_func ← function(ARGUMENTS) {
+     OPERATIONS
+     return(VALUE)
+   }
```

Let's try it out with a simple example function - one that converts temperatures from Fahrenheit to Celsius:

```
R> fahrenheit_to_celsius ← function(temp_F) {
+   temp_C ← (temp_F - 32) * (5/9)
+   return(temp_C)
+ }
```

Now, let's try out the function:

```
R> fahrenheit_to_celsius(451)
```

```
  [1] 232.7778
```

Pretty hot, isn't it?

[1] Yes, a function to create functions. 🤯

Functions: default argument values, if(), else()

Let's make the function a bit more complex, but also more fun.

```
R> temp_convert ←
+   function(temp, from = "f") {
+   if (!(from %in% c("f", "c"))){
+     stop("No valid input
+           temperature specified.")
+   }
+   if (from == "f") {
+     out ← (temp - 32) * (5/9)
+   } else {
+     out ← temp * (9/5) + 32
+   }
+   if((from == "c" & temp > 30) |
+      (from == "f" & out > 30)) {
+     message("That's damn hot!")
+   }else{
+     message("That's not so hot.")
+   }
+   return(out) # return temperature
+ }
```

Functions: default argument values, if(), else()

Let's make the function a bit more complex, but also more fun.

- By giving `from` a default value (`"f"`), we ensure that the function returns valid output when only the key input, `temp`, is provided.

```
R> temp_convert ←
+   function(temp, from = "f") {
+   if (!(from %in% c("f", "c"))){
+     stop("No valid input
+           temperature specified.")
+   }
+   if (from == "f") {
+     out ← (temp - 32) * (5/9)
+   } else {
+     out ← temp * (9/5) + 32
+   }
+   if((from == "c" & temp > 30) |
+      (from == "f" & out > 30)) {
+     message("That's damn hot!")
+   }else{
+     message("That's not so hot.")
+   }
+   return(out) # return temperature
+ }
```

Functions: default argument values, if(), else()

Let's make the function a bit more complex, but also more fun.

- By giving `from` a default value (`"f"`), we ensure that the function returns valid output when only the key input, `temp`, is provided.
- `if() { ... }` allows us to make conditional statements. Here, we test for the validity of the input for argument `from`.

```
R> temp_convert ←
+   function(temp, from = "f") {
+   if (!(from %in% c("f", "c"))){
+     stop("No valid input
+           temperature specified.")
+   }
+   if (from == "f") {
+     out ← (temp - 32) * (5/9)
+   } else {
+     out ← temp * (9/5) + 32
+   }
+   if((from == "c" & temp > 30) |
+      (from == "f" & out > 30)) {
+     message("That's damn hot!")
+   }else{
+     message("That's not so hot.")
+   }
+   return(out) # return temperature
+ }
```

Functions: default argument values, if(), else()

Let's make the function a bit more complex, but also more fun.

- By giving `from` a default value (`"f"`), we ensure that the function returns valid output when only the key input, `temp`, is provided.
- `if() { ... }` allows us to make conditional statements. Here, we test for the validity of the input for argument `from`.
- If the condition is not met, the function breaks and prints a message.

```
R> temp_convert ←
+   function(temp, from = "f") {
+   if (!(from %in% c("f", "c"))){
+     stop("No valid input
+           temperature specified.")
+   }
+   if (from == "f") {
+     out ← (temp - 32) * (5/9)
+   } else {
+     out ← temp * (9/5) + 32
+   }
+   if((from == "c" & temp > 30) |
+      (from == "f" & out > 30)) {
+     message("That's damn hot!")
+   }else{
+     message("That's not so hot.")
+   }
+   return(out) # return temperature
+ }
```

Functions: default argument values, if(), else()

Let's make the function a bit more complex, but also more fun.

- By giving `from` a default value (`"f"`), we ensure that the function returns valid output when only the key input, `temp`, is provided.
- `if() { ... }` allows us to make conditional statements. Here, we test for the validity of the input for argument `from`.
- If the condition is not met, the function breaks and prints a message.
- With `else()`, we specify what to do if the `if()` condition is not met.

```
R> temp_convert ←
+   function(temp, from = "f") {
+   if (!(from %in% c("f", "c"))){
+     stop("No valid input
+           temperature specified.")
+   }
+   if (from == "f") {
+     out ← (temp - 32) * (5/9)
+   } else {
+     out ← temp * (9/5) + 32
+   }
+   if((from == "c" & temp > 30) |
+      (from == "f" & out > 30)) {
+     message("That's damn hot!")
+   }else{
+     message("That's not so hot.")
+   }
+   return(out) # return temperature
+ }
```

Functions: default argument values, if(), else()

Let's make the function a bit more complex, but also more fun.

- By giving `from` a default value (`"f"`), we ensure that the function returns valid output when only the key input, `temp`, is provided.
- `if() { ... }` allows us to make conditional statements. Here, we test for the validity of the input for argument `from`.
- If the condition is not met, the function breaks and prints a message.
- We `else()` we specify what to do if the `if()` condition is not met.
- Make R more talkative with `message()`. Future-You will like it!

```
R> temp_convert ←
+   function(temp, from = "f") {
+   if (!(from %in% c("f", "c"))){
+     stop("No valid input
+           temperature specified.")
+   }
+   if (from == "f") {
+     out ← (temp - 32) * (5/9)
+   } else {
+     out ← temp * (9/5) + 32
+   }
+   if((from == "c" & temp > 30) |
+      (from == "f" & out > 30)) {
+     message("That's damn hot!")
+   }else{
+     message("That's not so hot.")
+   }
+   return(out) # return temperature
+ }
```

Anonymous functions

In R, functions are objects in their own right. They aren't automatically bound to a name. If you choose not to give the function a name, you get an **anonymous function**. You use an anonymous function when it's not worth the effort to give it a name.

Examples:

```
R> map(char_vec, function(x) paste(x, collapse = "|"))
R> integrate(function(x) sin(x) ^ 2, 0, pi)
```

Anonymous functions

In R, functions are objects in their own right. They aren't automatically bound to a name. If you choose not to give the function a name, you get an **anonymous function**. You use an anonymous function when it's not worth the effort to give it a name.

As of `R 4.1.0`, there's a new shorthand syntax for anonymous functions: `\(x)`.

Example:

```
R> (function (x) {paste(x, 'is awesome!')})('Data science') # old syntax
```

```
  [1] "Data science is awesome!"
```

```
R> (\(x) {paste(x, 'is awesome!')})('Data science') # new syntax
```

```
  [1] "Data science is awesome!"
```

Anonymous functions

In R, functions are objects in their own right. They aren't automatically bound to a name. If you choose not to give the function a name, you get an **anonymous function**. You use an anonymous function when it's not worth the effort to give it a name.

As of `R 4.1.0`, there's a new shorthand syntax for anonymous functions: `\(x)`. This plays along nicely with the (native) pipe when we want to pass content to the RHS but not to the first argument.

Anonymous functions

In R, functions are objects in their own right. They aren't automatically bound to a name. If you choose not to give the function a name, you get an **anonymous function**. You use an anonymous function when it's not worth the effort to give it a name.

As of `R 4.1.0`, there's a new shorthand syntax for anonymous functions: `\(x)`. This plays along nicely with the (native) pipe when we want to pass content to the RHS but not to the first argument.

Example:

```
R> mtcars |> subset(cyl == 4) |> (\(x) lm(mpg ~ disp, data = x))()
```

`...` (Dot-dot-dot)

Functions can have a special argument `...` (pronounced *dot-dot-dot*). In other programming languages, this type of argument is often called varargs (short for variable arguments), or ellipsis. With it, a function can take any number of additional arguments. That is potentially very powerful!

A common application is to use `...` to pass those additional arguments on to another function.

Toy example:

```
R> my_list_generator ← function(y, z) {
+   list(y = y, z = z)
+ }
R>
R> my_list_generator_2 ← function(x, ... ) {
+   my_list_generator( ... )
+ }
R>
R> str(my_list_generator_2(x = 1, y = 2, z = 3))
```

```
  List of 2
   $ y: num 2
   $ z: num 3
```

Real-life example:

```
R> map(.x, .f, ... )
R> map(mtcars, mean, na.rm = TRUE)
```

Arguments:

- `.x`: A list or atomic vector
- `.f`: A function
- `...`: Additional arguments passed on to the mapped function.

Writing functions with ChatGPT

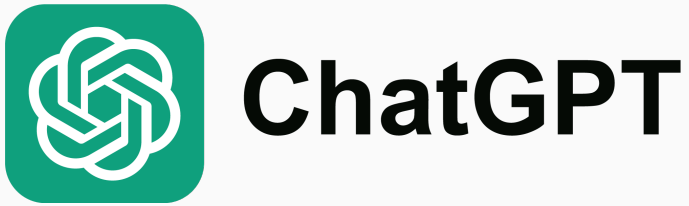
Not every function you plan to write is unique, nor is every problem you want to solve functionally.

ChatGPT, GitHub Copilot and other AI-based coding tools can help you a lot in finding functional solutions you can describe but not verbalize (yet).

I encourage you to use AI for this purpose, but be aware of the necessity to (a) debug and (b) assign credit where due.

Let's try it out with one of the following prompts:

- *Write an R function that capitalizes the first letter of each word in a character vector.*
- *Write an R function that allows me to play one round of black jack.*



Iteration

Iteration

The ubiquity of iteration

- Often we have to run the same task over and over again, with minor variations. Examples:
  - Standardize values of a variable
  - Recode all numeric variables in a dataset
  - Running multiple models with varying covariate sets
- A benefit of scripting languages in data (as opposed to point-and-click solutions) is that we can easily automate the process of iteration

Ways to iterate

- A simple approach is to copy-and-paste code with minor modifications (→ "duplicate code", → "copy-and-paste programming"). This is lazy, error-prone, not very efficient, and violates the "Don't repeat yourself" (DRY) principle.
- In R, vectorization, that is applying a function to every element of a vector at once, already does a good share of iteration for us.
- `for()` loops are intuitive and straightforward to build, but sometimes not very efficient.
- Finally, we learned about functions. Now, we learn how to unleash their power by applying them to anything we interact with in R at scale.

A simple example

Task

Say we want to double each element in a numeric vector, `x = c(1, 2, 3, 4, 5)`. Here are some different approaches to achieve this:

1. Manually (sometimes: copying and pasting code)

```
R> x ← c(1, 2, 3, 4, 5)
R> x_doubled ← c(2, 4, 6, 8, 10)
```

2. Vectorization

```
R> x ← c(1, 2, 3, 4, 5)
R> x_doubled ← x * 2
R> x_doubled
```

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

3. `for()` loop

```
R> x ← c(1, 2, 3, 4, 5)
R> x_doubled ← numeric(length(x))
R> for (i in seq_along(x)) {
+   x_doubled[i] ← x[i] * 2
+ }
R> x_doubled
```

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

4. Using `purrr`

```
R> library(purrr)
R> x ← c(1, 2, 3, 4, 5)
R> x_doubled ← map_dbl(x, ~ .x * 2)
R> x_doubled
```

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

Iteration with purrr

The tidyverse way to iterate

- For *real* functional programming in base R, we can use the `*apply()` family of functions (`lapply()`, `sapply()`, etc.). See here for an excellent summary.
- In the tidyverse, this functionality comes with the `purrr` package.
- At its core is the `map*()` family of functions.

How `purrr` works

- The idea is always to **apply** a function to **x**, where x can be a list, vector, data.frame, or something more complex.
- The output is then returned as output of a pre-defined type (e.g., a list).

Iteration with purrr: map()

The `map*()` functions all follow a similar syntax:

$$map(.x, .f, ...)$$

We use it to apply a function `.f` to each piece in `.x`. Additional arguments to `.f` can be passed on in `...`.

For instance, if we want to identify the object class of every column of a data.frame, we can write:

```
R> map(starwars, class)
```

```
$name
[1] "character"

$height
[1] "integer"

$mass
[1] "numeric"

$hair_color
[1] "character"

$skin_color
[1] "character"
```

Iteration with purrr: map() *cont.*

By default, `map()` returns a list. But we can also use other `map*()` functions to give us an atomic vector of an indicated type (e.g., `map_int()` to return an integer vector, or `map_vec()` to return a vector that is the simplest common type).

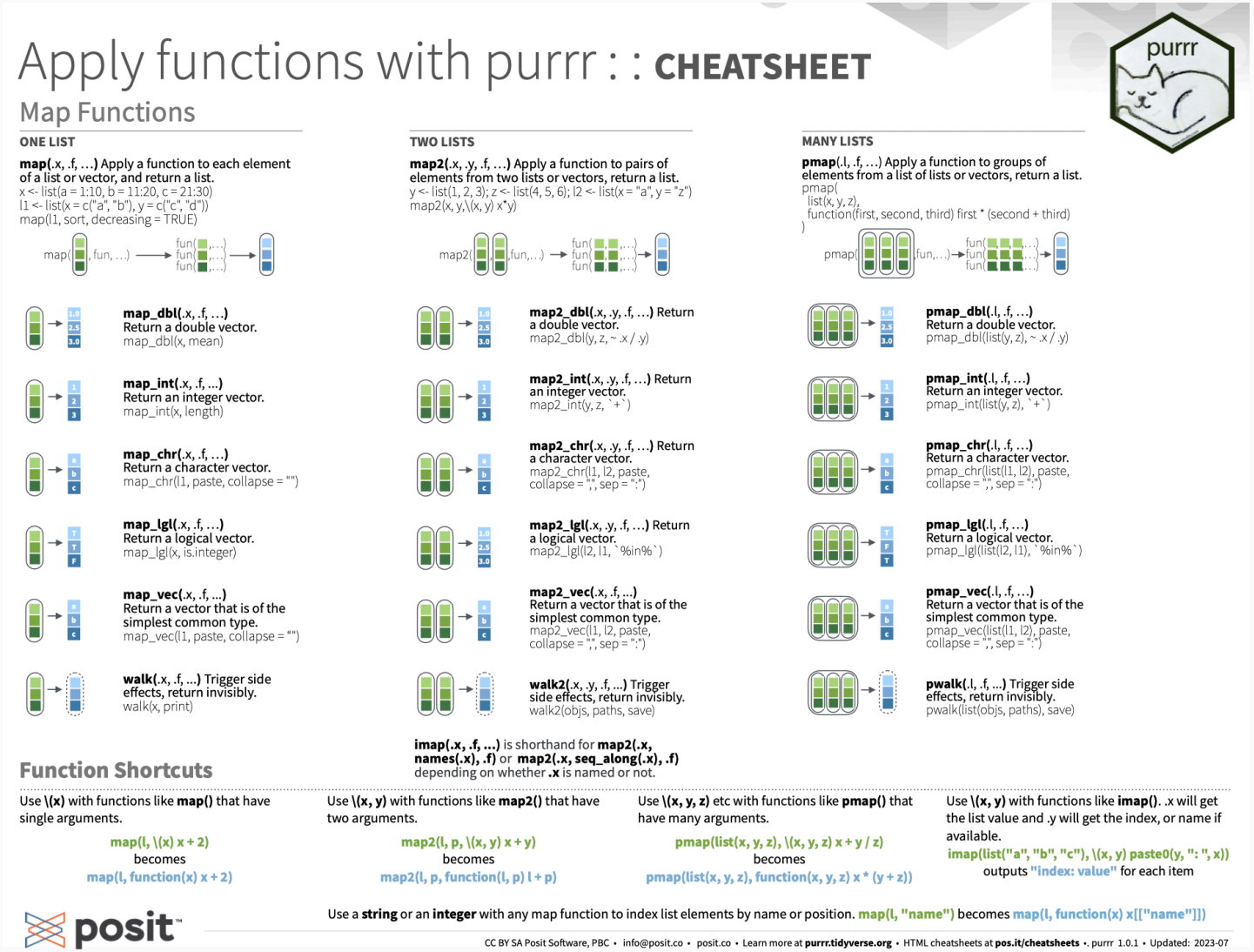
Going back to the previous example, we can also use `map_chr()`, which returns a character vector:

```
R> map_chr(starwars, class)
```

```
       name      height        mass  hair_color  skin_color   eye_color
"character"   "integer"   "numeric" "character" "character" "character"
 birth_year         sex      gender   homeworld     species       films
  "numeric" "character" "character" "character" "character"      "list"
   vehicles   starships
     "list"      "list"
```

The `purrr` function set is quite comprehensive. Be sure to check out the cheat sheet and the tutorials. You'll survive without `purrr` but you probably don't want to live without it. Together with `dplyr` it's easily the most powerful package for data wrangling in the tidyverse. If you master it, it will save you a lot of time and headaches.

Iteration with purrr: map() *cont.*



Apply functions with purrr : : CHEATSHEET

Map Functions

ONE LIST

map(.x, .f, …**)** Apply a function to each element of a list or vector, and return a list.

```
x <- list(a = 1:10, b = 11:20, c = 21:30)
l1 <- list(x = c("a", "b"), y = c("c", "d"))
map(l1, sort, decreasing = TRUE)
```

map(, fun, …) → fun(,…) fun(,…) fun(,…) →

map_dbl(.x, .f, …**)**
Return a double vector.
map_dbl(x, mean)

map_int(.x, .f, ...**)**
Return an integer vector.
map_int(x, length)

map_chr(.x, .f, …**)**
Return a character vector.
map_chr(l1, paste, collapse = "")

map_lgl(.x, .f, …**)**
Return a logical vector.
map_lgl(x, is.integer)

map_vec(.x, .f, ...**)**
Return a vector that is of the simplest common type.
map_vec(l1, paste, collapse = "")

walk(.x, .f, ...**)** Trigger side effects, return invisibly.
walk(x, print)

TWO LISTS

map2(.x, .y, .f, …**)** Apply a function to pairs of elements from two lists or vectors, return a list.

```
y <- list(1, 2, 3); z <- list(4, 5, 6); l2 <- list(x = "a", y = "z")
map2(x, y,\(x, y) x*y)
```

map2(, , fun,…) → fun(,,…) fun(,,…) fun(,,…) →

map2_dbl(.x, .y, .f, …**)** Return a double vector.
map2_dbl(y, z, ~ .x / .y)

map2_int(.x, .y, .f, …**)** Return an integer vector.
map2_int(y, z, `+`)

map2_chr(.x, .y, .f, …**)** Return a character vector.
map2_chr(l1, l2, paste, collapse = ",", sep = ":")

map2_lgl(.x, .y, .f, …**)** Return a logical vector.
map2_lgl(l2, l1, `%in%`)

map2_vec(.x, .y, .f, ...**)**
Return a vector that is of the simplest common type.
map2_vec(l1, l2, paste, collapse = ",", sep = ":")

walk2(.x, .y, .f, ...**)** Trigger side effects, return invisibly.
walk2(objs, paths, save)

imap(.x, .f, ...) is shorthand for **map2(.x, names(.x), .f)** or **map2(.x, seq_along(.x), .f)** depending on whether **.x** is named or not.

MANY LISTS

pmap(.l, .f, …**)** Apply a function to groups of elements from a list of lists or vectors, return a list.

```
pmap(
  list(x, y, z),
  function(first, second, third) first * (second + third)
)
```

pmap(, , fun,…) → fun(,,…) fun(,,…) fun(,,…) →

pmap_dbl(.l, .f, …**)**
Return a double vector.
pmap_dbl(list(y, z), ~ .x / .y)

pmap_int(.l, .f, …**)**
Return an integer vector.
pmap_int(list(y, z), `+`)

pmap_chr(.l, .f, …**)**
Return a character vector.
pmap_chr(list(l1, l2), paste, collapse = ",", sep = ":")

pmap_lgl(.l, .f, …**)**
Return a logical vector.
pmap_lgl(list(l2, l1), `%in%`)

pmap_vec(.l, .f, …**)**
Return a vector that is of the simplest common type.
pmap_vec(list(l1, l2), paste, collapse = ",", sep = ":")

pwalk(.l, .f, ...**)** Trigger side effects, return invisibly.
pwalk(list(objs, paths), save)

Function Shortcuts

Use **\(x)** with functions like **map()** that have single arguments.

map(l, \(x) x + 2)
becomes
map(l, function(x) x + 2)

Use **\(x, y)** with functions like **map2()** that have two arguments.

map2(l, p, \(x, y) x + y)
becomes
map2(l, p, function(l, p) l + p)

Use **\(x, y, z)** etc with functions like **pmap()** that have many arguments.

pmap(list(x, y, z), \(x, y, z) x + y / z)
becomes
pmap(list(x, y, z), function(x, y, z) x * (y + z))

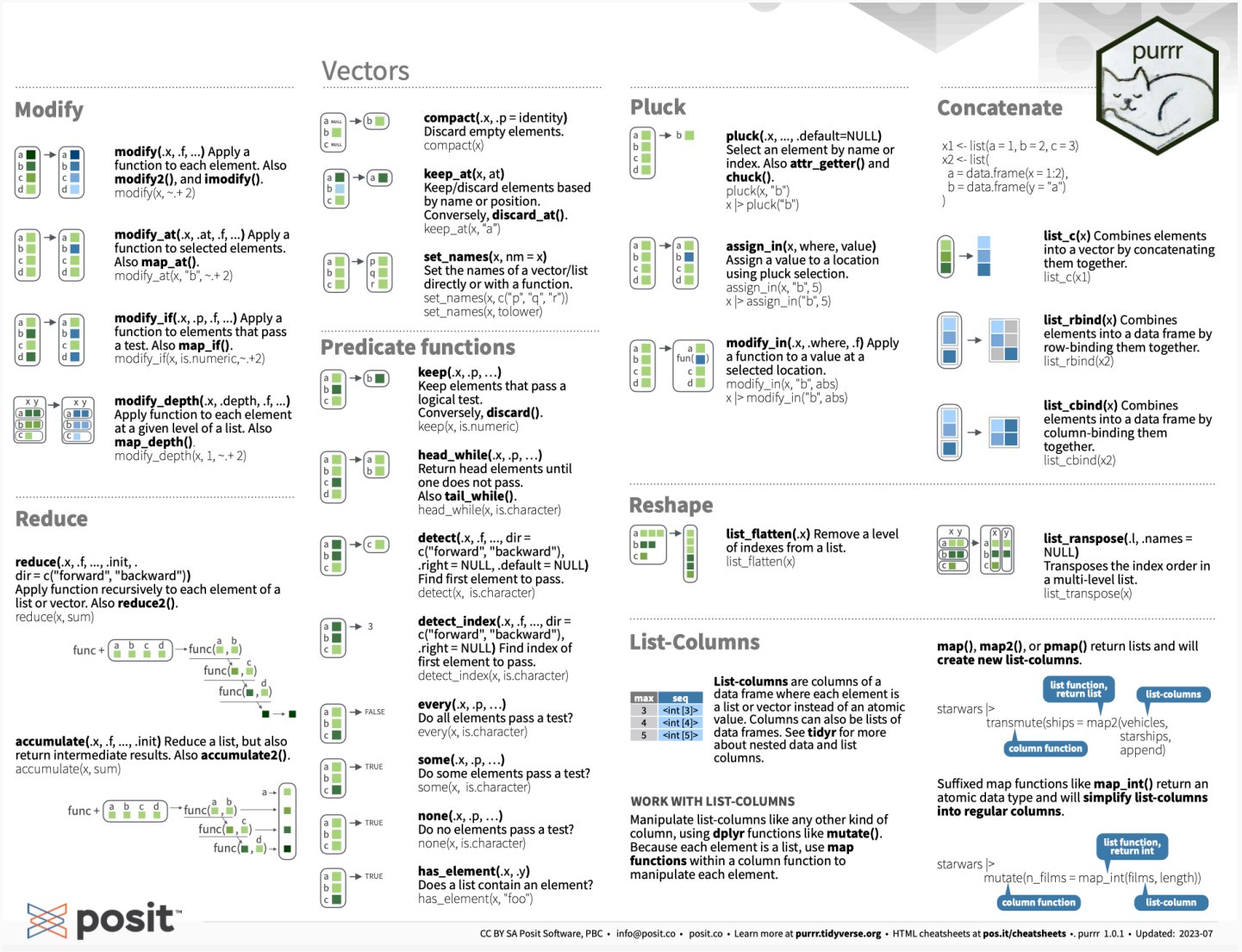
Use **\(x, y)** with functions like **imap()**. .x will get the list value and .y will get the index, or name if available.

imap(list("a", "b", "c"), \(x, y) paste0(y, ": ", x))
outputs **"index: value"** for each item

posit™

Use a **string** or an **integer** with any map function to index list elements by name or position. **map(l, "name")** becomes **map(l, function(x) x[["name"]])**

Iteration with purrr: map() *cont.*



Another example

Task

Let's say we want to calculate the mean and standard deviation of height and mass for each species in the `starwars` dataset.

```
R> # Load the starwars dataset
R> data(starwars)
R>
R> # Custom function for calculations
R> calc_stats ← function(df) {
+   df %>%
+     summarise(
+       height_mean = mean(height, na.rm = TRUE),
+       height_sd = sd(height, na.rm = TRUE),
+       mass_mean = mean(mass, na.rm = TRUE),
+       mass_sd = sd(mass, na.rm = TRUE)
+     )
+ }
```

```
R> # Group by species and apply the custom function
R> species_stats ← starwars %>%
+   group_by(species) %>%
+   nest() # Nesting the data
R> species_stats
```

```
  # A tibble: 38 × 2
  # Groups:   species [38]
     species        data
     <chr>          <list>
   1 Human          <tibble [35 × 13]>
   2 Droid          <tibble [6 × 13]>
   3 Wookiee        <tibble [2 × 13]>
   4 Rodian         <tibble [1 × 13]>
   5 Hutt           <tibble [1 × 13]>
   6 <NA>           <tibble [4 × 13]>
   7 Yoda's species <tibble [1 × 13]>
   8 Trandoshan     <tibble [1 × 13]>
   9 Mon Calamari   <tibble [1 × 13]>
  10 Ewok           <tibble [1 × 13]>
  # ℹ 28 more rows
```

Another example

Task

Let's say we want to calculate the mean and standard deviation of height and mass for each species in the `starwars` dataset.

```
R> # Load the starwars dataset
R> data(starwars)
R>
R> # Custom function for calculations
R> calc_stats ← function(df) {
+   df %>%
+     summarise(
+       height_mean = mean(height, na.rm = TRUE),
+       height_sd = sd(height, na.rm = TRUE),
+       mass_mean = mean(mass, na.rm = TRUE),
+       mass_sd = sd(mass, na.rm = TRUE)
+     )
+ }
```

```
R> # Group by species and apply the custom function
R> species_stats ← starwars %>%
+   group_by(species) %>%
+   nest() %>% # Nesting the data
+   # purrr magic
+   mutate(stats = map(data, calc_stats))
R> species_stats
```

```
  # A tibble: 38 × 3
  # Groups:   species [38]
     species        data               stats
     <chr>          <list>             <list>
   1 Human          <tibble [35 × 13]> <tibble [1 × 4]>
   2 Droid          <tibble [6 × 13]>  <tibble [1 × 4]>
   3 Wookiee        <tibble [2 × 13]>  <tibble [1 × 4]>
   4 Rodian         <tibble [1 × 13]>  <tibble [1 × 4]>
   5 Hutt           <tibble [1 × 13]>  <tibble [1 × 4]>
   6 <NA>           <tibble [4 × 13]>  <tibble [1 × 4]>
   7 Yoda's species <tibble [1 × 13]>  <tibble [1 × 4]>
   8 Trandoshan     <tibble [1 × 13]>  <tibble [1 × 4]>
   9 Mon Calamari   <tibble [1 × 13]>  <tibble [1 × 4]>
  10 Ewok           <tibble [1 × 13]>  <tibble [1 × 4]>
```

Another example

Task

Let's say we want to calculate the mean and standard deviation of height and mass for each species in the `starwars` dataset.

```
R> # Load the starwars dataset
R> data(starwars)
R>
R> # Custom function for calculations
R> calc_stats ← function(df) {
+   df %>%
+     summarise(
+       height_mean = mean(height, na.rm = TRUE),
+       height_sd = sd(height, na.rm = TRUE),
+       mass_mean = mean(mass, na.rm = TRUE),
+       mass_sd = sd(mass, na.rm = TRUE)
+     )
+ }
```

```
R> # Group by species and apply the custom function
R> species_stats ← starwars %>%
+   group_by(species) %>%
+   nest() %>% # Nesting the data
+   # purrr magic
+   mutate(stats = map(data, calc_stats)) %>%
+   select(species, stats) %>% # Select columns
+   unnest(stats) # Unnest the data
R> species_stats
```

```
  # A tibble: 38 × 5
  # Groups:   species [38]
     species        height_mean height_sd mass_mean mass_
     <chr>                <dbl>     <dbl>     <dbl>   <dl
   1 Human                178       12.0       81.3    1
   2 Droid                131.      49.1       69.8    5
   3 Wookiee              231        4.24     124      1
   4 Rodian               173       NA         74      NA
   5 Hutt                 175       NA       1358      NA
   6 <NA>                 175       12.4       81      3
   7 Yoda's species        66       NA         17      NA
   8 Trandoshan           190       NA        113      NA
```

Strategies for debugging

What's debugging?

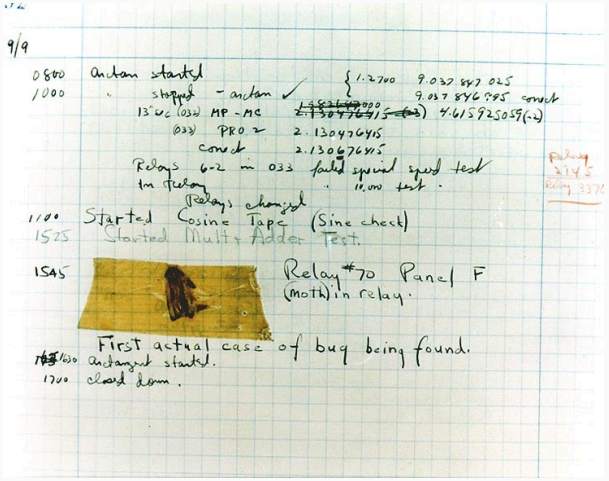
Straight from the Wikipedia

"Debugging is the process of finding and resolving bugs (defects or problems that prevent correct operation) within computer programs, software, or systems."

A famous (yet not the first) bug:

The term "bug" was used in an account by computer pioneer Grace Hopper (see on the right). While she was working on a Mark II computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system. This bug was carefully removed and taped to the log book (see on the right).

Above: Grace Hopper, *Below:* The bug



Why debugging matters

The Wikipedia list of software bugs with significant consequences is growing and you don't want to be on it.

NASA software engineers are famous for producing bug-free code. This was learned the hard and costly way though. Some highlights from space:

- 1962: A booster went off course during launch, resulting in the destruction of NASA Mariner 1 . This was the result of the failure of a transcriber to notice an overbar in a handwritten specification for the guidance program, resulting in an incorrect formula the FORTRAN code.
- 1999: NASA's Mars Climate Orbiter was destroyed, due to software on the ground generating commands based on parameters in pound-force (lbf) rather than newtons (N)
- 2004: NASA's Spirit rover became unresponsive on January 21, 2004, a few weeks after landing on Mars. Engineers found that too many files had accumulated in the rover's flash memory (the problem could be fixed though by deleting unnecessary files, and the Rover lived happily ever after. Until it froze to death in 2011).

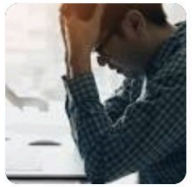
Why debugging matters (cont.)

ZDNet

Microsoft Excel blunder: Developers blamed for loss of thousands of COVID-19 test results

The error has hampered the UK's contact-tracing program at a time when the country is undergoing a second wave of coronavirus infections. By using the XLS ...

1 month ago



ABC News

Excel glitch leads to nearly 16,000 confirmed coronavirus cases going unreported in United Kingdom

For the test and trace program to work well, contacts should be notified as soon as possible. Health Secretary Matt Hancock told MPs that the problem related to ...

1 month ago

Metro

How were 16,000 Test and Trace coronavirus cases lost on Excel?

The cases were lost due to a technical error on a Microsoft Excel spreadsheet. ... Trace 'immediately' after the issue was resolved and thanked contact tracers for ...

1 month ago

Does Contact Tracing Work? Quasi-Experimental Evidence from an Excel Error in England*

Thiemo Fetzer† Thomas Graeber‡

November 24, 2020

Abstract

Contact tracing has been a central pillar of the public health response to the COVID-19 pandemic. Yet, contact tracing measures face substantive challenges in practice and well-identified evidence about their effectiveness remains scarce. This paper exploits quasi-random variation in COVID-19 contact tracing. Between September 25 and October 2, 2020, a total of 15,841 COVID-19 cases in England (around 15 to 20% of all cases) were not immediately referred to the contact tracing system due to a data processing error. Case information was truncated from an Excel spreadsheet after the row limit had been reached, which was discovered on October 3. There is substantial variation in the degree to which different parts of England areas were exposed – by chance – to delayed referrals of COVID-19 cases to to the contact tracing system. We show that more affected areas subsequently experienced a drastic rise in new COVID-19 infections and deaths alongside an increase in the positivity rate and the number of test performed, as well as a decline in the performance of the contact tracing system. Conservative estimates suggest that the failure of timely contact tracing due to the data glitch is associated with more than 125,000 additional infections and over 1,500 additional COVID-19-related deaths. Our findings provide strong quasi-experimental evidence for the effectiveness of contact tracing.

Keywords: Health, Coronavirus
JEL Classification: I31, Z18

Why debugging matters (cont.)

Technology

Facebook made big mistake in data it provided to researchers, undermining academic work

Company accidentally left out half of all of its U.S. users in providing data to a research consortium

The error resulted from Facebook accidentally excluding data from U.S. users who had no detectable political leanings — a group that amounted to roughly half of all of Facebook's users in the United States. Data from users in other countries was not affected.

"It's data. Of course, there are errors," said Gary King, a Harvard professor who co-chairs Social Science One. "This, of course, was a big error."

King, director of the university's Institute for Quantitative Social Science, said dozens of papers from researchers affiliated with Social Science One had relied on the data since Facebook shared the flawed set in February 2020, but he said the impact could be determined only after Facebook provided corrected data that could be reanalyzed. He said some of the errors may cause little or no problems, but others could be serious.

Social Science One shared the flawed data with at least 110 researchers, King said.

An Italian researcher, Fabio Giglietto, discovered data anomalies last month and brought them to Facebook's attention. The company contacted researchers in recent days with news that they had failed to include roughly half of its U.S. users — a group that likely is less politically polarized than Facebook's overall user base. The New York Times first reported Facebook's error.

`Source` Washington Post

`Source` Solomon Messing / Twitter

A general strategy for debugging

1. Google
2. Reset
3. Debug
4. Deter

Google

According to this analysis, the most common error types in R are:[1]

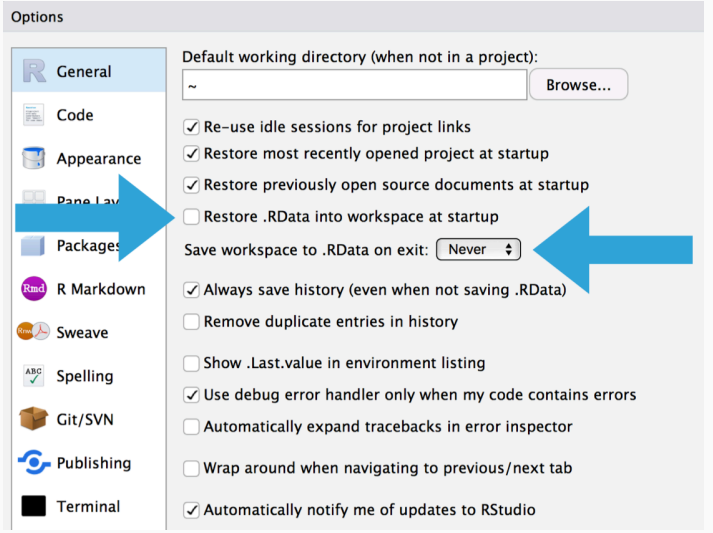
1. `Could not find function` errors, usually caused by typos or not loading a required package.
2. `Error in if` errors, caused by non-logical data or missing values passed to R's `if` conditional statement.
3. `Error in eval` errors, caused by references to objects that don't exist.
4. `Cannot open` errors, caused by attempts to read a file that doesn't exist or can't be accessed.
5. `no applicable method` errors, caused by using an object-oriented function on a data type it doesn't support.
6. `subscript out of bounds` errors, caused by trying to access an element or dimension that doesn't exist
7. Package errors caused by being unable to install, compile or load a package.

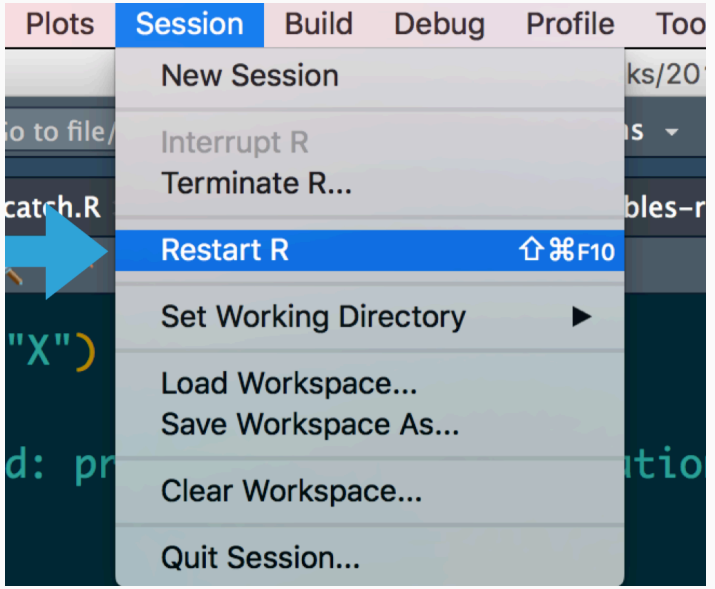
Whenever you see an error message, start by googling it. Improve your chances of a good match by removing any variable names or values that are specific to your problem. Also, look for Stack Overflow posts and list of answers.

[1] Do you get an error message you don't understand? That's good news actually, because the really nasty bugs come without errors.

Reset

- If at first you don't succeed, try exactly the same thing again.
- Have you tried turning it off and on again?
- Do you use `rm(list = ls())`? Don't. Packages remain loaded, options and environment variables set, ... all possible sources of error!
- A fresh start clears the workspace, resets options, environment variables, and the path.
- While we're at it, check out James Wade's advice "How I set up RStudio for Efficient Coding" (YouTube).





Debug

Make the error repeatable.

- Execute the code many times as you consider and reject hypotheses. To make that iteration as quick possible, it's worth some upfront investment to make the problem both easy and fast to reproduce.
- Work with reproducible and minimal examples by removing innocuous code and simplifying data.
- Consider automated testing. Add some nearby tests to ensure that existing good behaviour is preserved.

Track the error down.

- Execute code step by step and inspect intermediate outputs.
- Adopt the scientific method: Generate hypotheses, design experiments to test them, and record your results.

Once found, fix the error and test it.

- Ensure you haven't introduced any new bugs in the process.
- Make sure to carefully record the correct output, and check against the inputs that previously failed.
- Reset and run again to make sure everything still works.

Deter

Defensive programming

- **Pay attention.** Do results make sense? Do they look different from previous results? Why?
- **Know what you're doing**, and what you're expecting.
  - Avoid functions that return different types of output depending on their input, e.g., `[]` and `sapply()`.
  - Be strict about what you accept (e.g., only scalars).
  - Avoid functions that use non-standard evaluation (e.g., `with()`)
- **Fail fast**.
  - As soon as something wrong is discovered, signal an error.
  - Add tests (e.g., with the `testthat` package).
  - Practice good condition/exception handling, e.g., with `try()` and `tryCatch()`.
  - Write error messages for humans.

Transparency

- Collaborate! Pair programming is an established software development technique that increases code robustness. It also works from remote.
- Be transparent! Let others access your code and comment on it.

Debugging R

What you get

```
Error : .onLoad failed in loadNamespace() for 'rJava', details:
call: dyn.load(file, DLLpath = DLLpath, ...)
error: unable to load shared object '/Users/janedoe/Library/R/3.6/library/rJava/libs/rJava.so':
libjvm.so: cannot open shared object file: No such file or directory
Error: loading failed
Execution halted
ERROR: loading failed
* removing '/Users/janedoe/Library/R/3.6/library/rJava/'
Warning in install.packages :
installation of package 'rJava' had non-zero exit status
```

`Credit` Jenny Bryan

What you see

```
Error : blah failed blah blah() blah 'blah', blah:
call: blah.blah(blah, blah = blah, ...)
error: unable to blah blah blah '/blah/blah/blah/blah/blah/blah/blah/blah/blah.so':
blah.so: cannot open blah blah blah: No blah blah blah blah
Error: blah failed
blah blah
ERROR: blah failed
* removing '/blah/blah/blah/blah/blah/blah/blah/'
Warning in blah.blah :
blah of blah 'blah' blah blah-blah blah blah
```

`Credit` Jenny Bryan

Strategies to debug your R code

Sometimes the mistake in your code is hard to diagnose, and googling doesn't help. Here are a couple of strategies to debug your code:

- Use `traceback()` to determine where a given error is occurring.
- Output diagnostic information in code with `print()`, `cat()` or `message()` statements.
- Use `browser()` to open an interactive debugger before the error
- Use `debug()` to automatically open a debugger at the start of a function call.
- Use `trace()` to make temporary code modifications inside a function that you don't have easy access to.

Locating errors with traceback()

Motivation and usage

- When an error occurs with an unidentifiable error message or an error message that you are in principle familiar with but cannot locate its sources, the `traceback()` function comes in handy.
- The `traceback()` function prints the sequence of calls that led to an uncaught error.
- The `traceback()` output reads from bottom to top.
- Note that errors caught via `try()` or `tryCatch()` do not generate a traceback!
- If you're calling code that you `source()`d into R, the traceback will also display the location of the function, in the form `filename.r#linenumber`.

Example

In the call sequence below, the execution of `g()` triggers an error:

```
R> f ← function(x) x + 1
R> g ← function(x) f(x)
R> g("a")
```

```
#> Error in x + 1 : non-numeric argument to binary o
```

Doing the traceback reveals that the function call f(x) is what lead to the error:

```
R> traceback()
```

```
#> 2: f(x) at #1
#> 1: g("a")
```

Interactive debugging with browser()

Motivation and usage

- Sometimes, you need more information than the precise location of an error in a function to fix it.
- The interactive debugger lets you pause the run of a function and interactively explore its state.
- Two options to enter the interactive debugger:
  1. Through RStudio's "Rerun with Debug" tool, shown to the right of an error message.
  2. You can insert a call to `browser()` into the function at the stage where you want to pause, and re-run the function.
- In either case, you'll end up in an interactive environment inside the function where you can run arbitrary R code to explore the current state. You'll know when you're in the interactive debugger because you get a special prompt, `Browse[1]>`.

Example

```
R> h ← function(x) x + 3
R> g ← function(b) {
+   browser()
+   h(b)
+ }
R> g(10)
```

Some useful things to do are:

1. Use `ls()` to determine what objects are available in the current environment.
2. Use `str()`, `print()` etc. to examine the objects.
3. Use `n` to evaluate the next statement.
4. Use `s`: like `n` but also step into function calls.
5. Use `where` to print a stack trace (→ traceback).
6. Use `c` to exit debugger and continue execution.
7. Use `Q` to exit debugger and return to the R prompt.

Debugging other peoples' code

Motivation

- Sometimes the error is outside your code in a package you're using, you might still want to be able to debug.
- Two options:
  1. Get a local version of the package code and debug as if it were your own.
  2. Use functions which which allow you to start a browser in existing functions, including `recover()` and `debug()`.

Debugging other peoples' code (cont.)

Motivation

- `recover()` serves as an alternative error handler which you activate by calling `options(error = recover)`.
- You can then select from a list of current calls to browse.
- `options(error = NULL)` turns off this debugging mode again.
- A simpler alternative is `options(error = browser)`, but this only allows you to browse the call where the error occurred.

Example

- Activate debugging mode; then execute (flawed) function:

```
R> options(error = recover)
R> lm(mpg ~ wt, data = "mtcars")
```

```
Error in model.frame.default(formula = mpg ~ wt, data = "mtcars", drop
 'data' must be a data.frame, environment, or list

Enter a frame number, or 0 to exit

1: lm(mpg ~ wt, data = "mtcars")
2: eval(mf, parent.frame())
3: eval(mf, parent.frame())

Selection:
```

- Deactivate debugging mode:

```
R> options(error = NULL)
```

Debugging other peoples' code (cont.)

Motivation

- `debug()` activates the debugger on any function, including those in packages (see on the right). `undebug()` deactivates the debugger again.
- Some functions in another package are easier to find than others. There are
  - *exported* functions which are available outside of a package and
  - *internal* functions which are only available within a package.
- To find (and debug) exported functions, use the `::` syntax, as in `ggplot2::ggplot`.
- To find un-exported functions, use the `:::` syntax, as in `ggplot2:::check_required_aesthetics`.

Example

- Activate debugging mode for `lm()` function; then execute function:

```
R> debug(stats::lm)
R> lm(mpg ~ weight, data = "mtcars")
```

- Interactive debugging mode for `lm()` is entered; use the common `browser()` functionality to navigate:

```
debugging in: lm(mpg ~ weight, data = mtcars)
debug: {
    ret.x ← x
    ...
Browse[2]>
```

- Deactivate debugging mode:

```
R> undebug(stats::lm)
```

Debug Mode

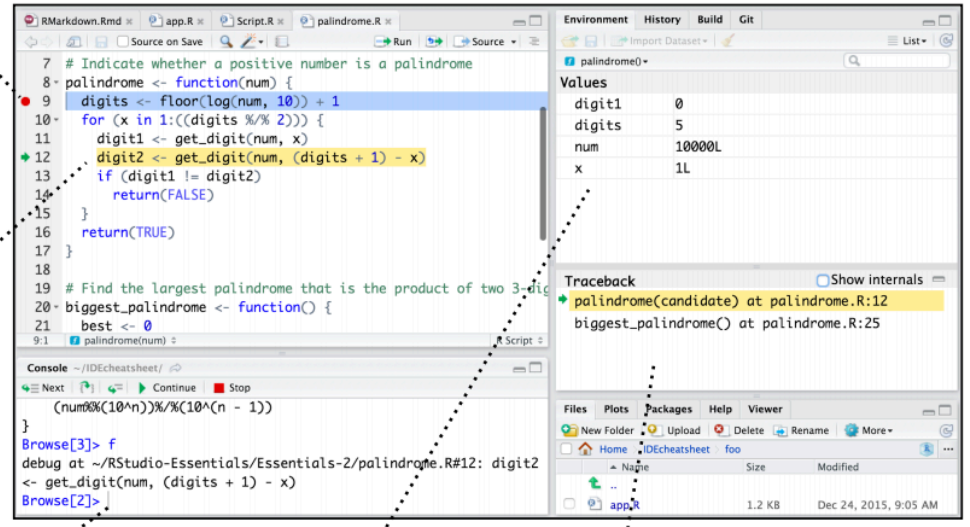
Open with **debug(), browser(),** or a breakpoint. RStudio will open the debugger mode when it encounters a breakpoint while executing code.

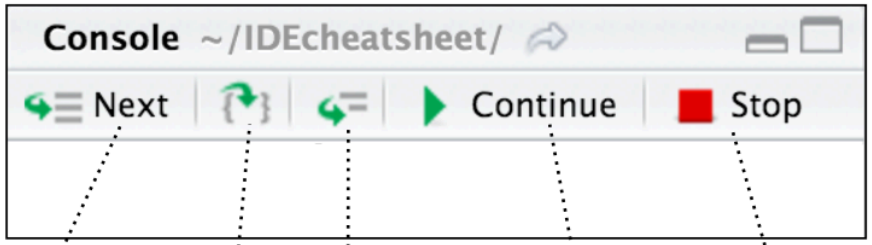
Launch debugger mode from origin of error

Open traceback to examine the functions that R called before the error occurred

Click next to line number to add/remove a breakpoint.

Highlighted line shows where execution has paused





Run commands in environment where execution has paused

Examine variables in executing environment

Select function in traceback to debug

Step through code one line at a time

Step into and out of functions to run

Resume execution

Quit debug mode

More on debugging R

Further reading

- 12-minute video on debugging in R
- Jenny Bryan's talk on debugging at rstudio::conf 2020
- Jenny Bryan and Jim Hester's "What They Forgot to Teach You About R", Chapter 11: Debugging R code
- Jonathan McPherson's Debugging with RStudio

Next steps

Assignment

Assignment 2 is online! You have a bit more than a week to work on it - final upload deadline is Sep 24, 9:30am CET.

Next lecture

Relational databases and SQL. Buckle up and bring coffee, because it'll get both exciting and tedious at the same time.