

Functions

Writing your own functions

So far we've seen many functions, like `c()`, `class()`, `filter()`, `dim()` ...

Why create your own functions?

- Cut down on repetitive code (easier to fix things!)
- Organize code into manageable chunks
- Avoid running code unintentionally
- Use names that make sense to you

Writing your own functions

The general syntax for a function is:

```
function_name <- function(arg1, arg2, ...) {  
  <function body>  
}
```

Writing your own functions

Here we will write a function that multiplies some number x by 2:

```
div_100 <- function(x) x / 100
```

When you run the line of code above, you make it ready to use (no output yet!).
Let's test it!

```
div_100(x = 600)
```

```
[1] 6
```

Writing your own functions: { }

Adding the curly brackets - {} - allows you to use functions spanning multiple lines:

```
div_100 <- function(x) {  
  x / 100  
}  
div_100(x = 10)  
  
[1] 0.1
```

Writing your own functions: `return`

If we want something specific for the function's output, we use `return()`:

```
div_100_plus_4 <- function(x) {  
  output_int <- x / 100  
  output <- output_int + 4  
  return(output)  
}  
div_100_plus_4(x = 10)
```

```
[1] 4.1
```

Writing your own functions: multiple inputs

Functions can take multiple inputs:

```
div_100_plus_y <- function(x, y) x / 100 + y  
div_100_plus_y(x = 10, y = 3)
```

```
[1] 3.1
```

Writing your own functions: multiple outputs

Functions can return a vector (or other object) with multiple outputs.

```
x_and_y_plus_2 <- function(x, y) {  
  output1 <- x + 2  
  output2 <- y + 2  
  
  return(c(output1, output2))  
}  
result <- x_and_y_plus_2(x = 10, y = 3)  
result  
  
[1] 12  5
```

Writing your own functions: defaults

Functions can have “default” arguments. This lets us use the function without using an argument later:

```
div_100_plus_y <- function(x = 10, y = 3) x / 100 + y  
div_100_plus_y()
```

```
[1] 3.1
```

```
div_100_plus_y(x = 11, y = 4)
```

```
[1] 4.11
```

Writing another simple function

Let's write a function, `sqdif`, that:

1. takes two numbers x and y with default values of 2 and 3.
2. takes the difference
3. squares this difference
4. then returns the final value

Writing another simple function

```
sqdif <- function(x = 2, y = 3) (x - y)^2
```

```
sqdif()
```

```
[1] 1
```

```
sqdif(x = 10, y = 5)
```

```
[1] 25
```

```
sqdif(10, 5)
```

```
[1] 25
```

```
sqdif(11, 4)
```

```
[1] 49
```

Writing your own functions: characters

Functions can have any kind of input. Here is a function with characters:

```
loud <- function(word) {  
  output <- rep(toupper(word), 5)  
  return(output)  
}  
loud(word = "hooray!")  
  
[1] "HOORAY!" "HOORAY!" "HOORAY!" "HOORAY!" "HOORAY!"
```

Functions for tibbles - curly braces

```
# get means and missing for a specific column
get_summary <- function(dataset, col_name) {
  dataset |>
    summarise(mean = mean({{col_name}}), na.rm = TRUE),
    na_count = sum(is.na('{{col_name}}")))
}
```

Functions for tibbles - example

```
er <- read_csv(file = "https://daseh.org/data/CO_ER_heat_visits.csv")  
  
get_summary(er, visits)  
  
# A tibble: 1 × 2  
  mean na_count  
  <dbl>   <int>  
1 7.19     303  
  
yearly_co2 <-  
  read_csv(file = "https://daseh.org/data/Yearly_CO2_Emissions_1000_tonnes.csv")  
  
get_summary(yearly_co2, `2014`)  
  
# A tibble: 1 × 2  
  mean na_count  
  <dbl>   <int>  
1 175993.      0
```

Summary

- Simple functions take the form:
 - `NEW_FUNCTION <- function(x, y){x + y}`
 - Can specify defaults like `function(x = 1, y = 2){x + y}`
 - `return` will provide a value as output
- Specify a column (from a tibble) inside a function using `{}{double curly braces}{}{double curly braces}`

Lab Part 1

□ [Class Website](#)

□ [Lab](#)

Functions on multiple columns

Using your custom functions: **sapply()**- a base R function

Now that you've made a function... You can "apply" functions easily with **sapply()**!

These functions take the form:

```
sapply(<a vector, list, data frame>, some_function)
```

Using your custom functions: `sapply()`

Let's apply a function to look at the CO heat-related ER visits dataset.

¶ There are no parentheses on the functions!¶

You can also pipe into your function.

```
sapply(er, class)
```

```
  county      rate   lower95cl   upper95cl     visits      year
"character" "numeric" "numeric" "numeric" "numeric" "numeric"
```

```
# also: er |> sapply(class)
```

Using your custom functions: `sapply()`

Use the `div_100` function we created earlier to convert 0-100 percentiles to proportions.

```
er |>  
  select(ends_with("cl")) |>  
  sapply(div_100) |>  
  head()
```

	lower95cl	upper95cl
[1,]	NA	0.09236776
[2,]	0.02848937	NA
[3,]	0.04359735	0.09313561
[4,]	0.01711087	0.04846996
[5,]	0.01892912	0.05232461
[6,]	0.06124961	0.11572046

Using your custom functions “on the fly” to iterate

Also called “anonymous function”.

```
er |>  
  select(ends_with("c1")) |>  
  sapply(function(x) x / 100) |>  
  head()
```

	lower95cl	upper95cl
[1,]	NA	0.09236776
[2,]	0.02848937	NA
[3,]	0.04359735	0.09313561
[4,]	0.01711087	0.04846996
[5,]	0.01892912	0.05232461
[6,]	0.06124961	0.11572046

Anonymous functions: alternative syntax

```
er |>  
  select(ends_with("c1")) |>  
  sapply(\(x) x / 100) |>  
  head()
```

	lower95cl	upper95cl
[1,]	NA	0.09236776
[2,]	0.02848937	NA
[3,]	0.04359735	0.09313561
[4,]	0.01711087	0.04846996
[5,]	0.01892912	0.05232461
[6,]	0.06124961	0.11572046

across

Using functions in `mutate()` and `summarize()`

Already know how to use functions to modify columns using `mutate()` or calculate summary statistics using `summarize()`.

```
er |>
  summarize(max_visits = max(visits, na.rm = T),
            max_rate = max(rate, na.rm = T))

# A tibble: 1 × 2
  max_visits max_rate
  <dbl>      <dbl>
1        48     89.3
```

Applying functions with **across** from **dplyr**

`across()` makes it easy to apply the same transformation to multiple columns.
Usually used with `summarize()` or `mutate()`.

```
summarize(across(<columns>, function))
```

or

```
mutate(across(<columns>, function))
```

- List columns first: `.cols =`
- List function next: `.fns =`
- If there are arguments to a function (e.g., `na.rm = TRUE`), use an anonymous function.

Applying functions with `across` from `dplyr`

Combining with `summarize()`

```
er |>
  summarize(across(
    c(visits, rate),
    mean # no parentheses
  ))
```

```
# A tibble: 1 × 2
  visits   rate
  <dbl>   <dbl>
1     NA     NA
```

Applying functions with **across** from **dplyr**

Add anonymous function to include additional arguments (e.g., `na.rm = T`).

```
er |>
  summarize(across(
    c(visits, rate),
    function(x) mean(x, na.rm = T)
  ))
```

```
# A tibble: 1 × 2
  visits   rate
  <dbl>   <dbl>
1     7.19  2.43
```

Applying functions with `across` from `dplyr`

Can use with other tidyverse functions like `group_by`!

```
er |>
  group_by(year) |>
  summarize(across(
    c(visits, rate),
    function(x) mean(x, na.rm = T)
  ))
```

A tibble: 12 × 3

	year	visits	rate
	<dbl>	<dbl>	<dbl>
1	2011	5.20	1.49
2	2012	5.89	1.75
3	2013	5.63	1.83
4	2014	4.12	1.41
5	2015	6.4	1.96
6	2016	10.1	5.28
7	2017	7.24	2.13
8	2018	11.7	3.28
9	2019	9.12	4.09
10	2020	6.26	1.73
11	2021	8.06	2.08
12	2022	9.29	3.21

Applying functions with `across` from `dplyr`

Using different `tidyselect()` options (e.g., `starts_with()`, `ends_with()`, `contains()`)

```
er |>
  group_by(year) |>
  summarize(across(contains("c1"), mean, na.rm=T))

# A tibble: 12 × 3
  year lower95cl upper95cl
  <dbl>    <dbl>    <dbl>
1 2011     0.836    2.12
2 2012     1.06     2.41
3 2013     1.07     2.62
4 2014     0.810    2.11
5 2015     1.21     2.77
6 2016     3.05     7.99
7 2017     1.28     3.08
8 2018     2.17     4.41
9 2019     2.32     6.21
10 2020    1.02     2.52
11 2021    1.30     2.92
12 2022    1.93     4.71
```

Applying functions with `across` from `dplyr`

Combining with `mutate()` - the `replace_na` function

Let's look at the yearly CO₂ emissions dataset we loaded earlier.

`replace_na({data frame}, {list of values})` or `replace_na({vector}, {single value})`

```
yearly_co2 |>
  select(country, starts_with("194")) |>
  mutate(across(
    c(`1943`, `1944`, `1945`),
    function(x) replace_na(x, replace = 0)
  ))
```

A tibble: 192 × 11

	country	`1940`	`1941`	`1942`	`1943`	`1944`	`1945`	`1946`	`1947`	`1948`
	<chr>	<dbl>								
1	Afghanistan	NA	NA	NA	0	0	0	NA	NA	NA
2	Albania	693	627	744	462	154	121	484	928	704
3	Algeria	238	312	499	469	499	616	763	744	803
4	Andorra	NA	NA	NA	0	0	0	NA	NA	NA
5	Angola	NA	NA	NA	0	0	0	NA	NA	NA
6	Antigua and B...	NA	NA	NA	0	0	0	NA	NA	NA
7	Argentina	15900	14000	13500	14100	14000	13700	13700	14500	17400
8	Armenia	848	745	513	655	613	649	730	878	935
9	Australia	29100	34600	36500	35000	34200	32700	35500	38000	38500
10	Austria	7350	7980	8560	9620	9400	4570	12800	17600	24500
	# 182 more rows									
	# 1 more variable: `1949` <dbl>									

GUT CHECK!

Why use `across()`?

- A. Efficiency - faster and less repetitive
- B. Calculate the cross product
- C. Connect across datasets

purrr package

Similar to `across`, `purrr` is a package that allows you to apply a function to multiple columns in a data frame or multiple data objects in a list.

While we won't get into `purrr` too much in this class, its a handy package for you to know about should you get into a situation where you have an irregular list you need to handle!

Multiple Data Frames

Multiple data frames

Lists help us work with multiple tibbles / data frames

```
df_list <- list(AQ = airquality, er = er, yearly_co2 = yearly_co2)
```

select() from each tibble the numeric columns:

```
df_list <-
  df_list |>
  sapply(function(x) select(x, where(is.numeric)))
```

Multiple data frames: sapply

```
df_list |> sapply(nrow)
```

AQ	er	yearly_co2
153	768	192

```
df_list |> sapply(colMeans, na.rm = TRUE)
```

\$AQ

Ozone	Solar.R	Wind	Temp	Month	Day
42.129310	185.931507	9.957516	77.882353	6.993464	15.803922

\$er

rate	lower95cl	upper95cl	visits	year
2.431466	1.449322	3.526338	7.189247	2016.500000

\$yearly_co2

1751	1752	1753	1754	1755	1756	1757
9360.000	9360.000	9360.000	9370.000	9370.000	10000.000	10000.000
1758	1759	1760	1761	1762	1763	1764
10000.000	10000.000	10000.000	11000.000	11000.000	11000.000	11000.000
1765	1766	1767	1768	1769	1770	1771
11000.000	12300.000	12300.000	12300.000	12300.000	12300.000	13600.000
1772	1773	1774	1775	1776	1777	1778
13600.000	13600.000	13600.000	13600.000	15000.000	15100.000	15100.000
1779	1780	1781	1782	1783	1784	1785
15100.000	15100.000	16900.000	16900.000	16900.000	16900.000	8451.835
1786	1787	1788	1789	1790	1791	1792
9601.835	9601.835	9601.835	9601.835	9601.835	10701.835	7290.890
1793	1794	1795	1796	1797	1798	1799
7294.557	7315.890	7316.890	7646.223	8051.223	8359.890	8810.223
1800	1801	1802	1803	1804	1805	1806
5631.934	5590.134	5262.667	6299.534	5730.945	6691.334	7019.534
1807	1808	1809	1810	1811	1812	1813
6153.112	7019.134	7022.134	6231.445	6603.112	6845.945	6874.445
1814	1815	1816	1817	1818	1819	1820

Summary

- Apply your functions with `sapply(<a vector or list>, some_function)`
- Use `across()` to apply functions across multiple columns of data
- Need to use `across` within `summarize()` or `mutate()`
- Can use `sapply` (or `purrr` package) to work with multiple data frames within lists simultaneously

Lab Part 2

- [Class Website](#)
- [Lab](#)
- [Day 9 Cheatsheet](#)
- [Posit's purrr Cheatsheet](#)

Research Survey

<https://forms.gle/jVue79CjgoMmbVbg9>



Image by [Gerd Altmann](#) from [Pixabay](#)