Functions in R

Setup

- Go to your CM515 directory
- Run 'git pull'

- You should get a new directory inside 12_Programming_Basics called **functions:**
- **Lecture_examples.R**
- **functions_practice.md**

Why write functions

1. 🔁 **Reusability -** "Write once, use anywhere."
    - *Functions allow you to **encapsulate a process** so that you don't have to rewrite the same logic again and again.*
    - Reusability also means that if the underlying calculation needs to change (say, a new standard factor), you **only change the function once**, not in 20 different places.

2. ✨ **Clean Code / Less Repetition -** "If you're copying and pasting more than twice, you probably need a function."
    - Repeating similar blocks of code makes your script harder to read and more error-prone.

    Functions help you:
    - Avoid clutter
    - Organize your work logically

3. 🧩 **Modular Thinking**
    - Functions promote **modular design** – you can build complex analyses by chaining together small, well-defined pieces.
    - Modular code is easier to **test**, **maintain**, and **collaborate on**. If something breaks, you can pinpoint which part needs fixing.

Why Functions Matter — Protein Concentration Example

You're working in a lab and need to calculate protein concentration from a Bradford assay using a standard formula:

$$\textbf{Concentration}_{\textbf{protein}}\ (\mu g/\mu L) = A_{595} \times F$$

- A_{595} = Absorbance measured at 595 nm
- F = Standard factor, derived from the slope of a standard curve

In this assay, a dye (Brilliant Blue) binds to proteins and changes color. You then measure the color change using a spectrophotometer, which gives you an absorbance value at 595 nm. The more protein, the darker the color, and the higher the absorbance.

You've derived from a standard curve, where you measure absorbance for samples with known protein concentrations. You then fit a line, and the slope of that line becomes your conversion factor –

`1.45 µg/µL per unit absorbance.`

Why Functions Matter — Protein Concentration Example

🔴 No Functions (Manual + Repetitive)

```r

# Sample 1
abs1 <- 0.45
conc1 <- abs1 * 1.45
conc1

# Sample 2
abs2 <- 0.89
conc2 <- abs2 * 1.45
conc2

# Sample 3
abs3 <- 0.32
conc3 <- abs3 * 1.45
conc3
```

✅ Simple Function

```r


calc_conc <- function(absorbance) {
  return(absorbance * 1.45)
}
```

🧩 Key Takeaway:

- Only write the formula once.
- You can reuse it on any number of values—and if the factor changes, you just update the function once.

You've already been using some pre-defined functions!

- ***typeof()***

- In R, a "closure" is just the technical term for a user-defined function.

- What happens if we type:

```
typeof(calc_conc) #?

typeof(calc_conc(80)) #?

typeof(calc_conc()) #?
```

- **c(12.5, 14.2, 13.8, 15.0)**
- **data.frame()**
- **mean()**

```
library(ggplot2)

# Create a sample data frame
set.seed(42)
cell_sizes <- data.frame(
  condition = rep(c("Control", "Treatment"), each = 10),
  size = c(rnorm(10, mean = 15, sd = 2), rnorm(10, mean = 18, sd = 2))
)

# Plot
ggplot(cell_sizes, aes(x = condition, y = size)) +
  geom_boxplot(fill = "skyblue", alpha = 0.7) +
  stat_summary(fun = mean, geom = "point", color = "red", size = 3) +
  labs(title = "Cell Size by Condition",
       y = "Cell Size (µm)",
       x = "Condition") +
  theme_minimal()
```

Anatomy of a function in R

- a **function** is a block of code that you can give inputs (called **arguments**) and get an output (called a **return value**).

```r
my_function <- function(argument1, argument2) {
  # code that uses the arguments
  result <- argument1 + argument2
  return(result)
}
```

🧩 **Key Takeaway:**

- The ***`function()`*** keyword defines a function.
- Arguments are **placeholders** for values you'll pass in - They can be numbers, strings, vectors, or even data frames.
- The `return()` function specifies **what comes out** of the function.

```r
greet <- function() {
  return("Hello, CM515 students!")
}
```

Anatomy of a function in R – how to run functions

- Let's say you're calculating total DNA mass from concentration and volume:

```r
calc_dna_mass <- function(concentration_ng_per_uL, volume_uL) {
  mass <- concentration_ng_per_uL * volume_uL
  return(mass)
}
```

- Call the function - "execute the function after you've defined it"

```r
calc_dna_mass <- function(concentration_ng_per_uL, volume_uL) {
  mass <- concentration_ng_per_uL * volume_uL
  return(mass)
}

calc_dna_mass(50, 20)
```

```r
greet <- function() {
  return("Hello, CM515 students!")
}
greet()
```

🧩 **Key TakeAway:**

- The function CALL comes **after** the function definition

DEMO: calc_dna_mass, greet() function

Local and Global Scope

- **Global scope** refers to variables that exist **outside** of any function – they're defined in the "main" R environment.

```r
concentration <- 80  # Global variable
volume <- 1.5        # Global variable

calculate_protein_mass <- function() {
  mass <- concentration * volume  # Uses global variables
  return(mass)
}

calculate_protein_mass()
```

- **Local scope** refers to variables defined **inside** a function – these exist only while the function is running.

DEMO:

calculate_protein_mass()

```r
# Slide 9 --  local and Global scope demo
calculate_protein_mass <- function(concentration, volume) {
  mass <- concentration * volume  # 'mass' exists only here
  return(mass)
}
```

Control Flow (decision making) in functions

```r
is_positive <- function(x) {
  if (x > 0) {
    return("Positive")
  } else {
    return("Not positive")
  }
}
is_positive(-2)
```

```r
classify_number <- function(x) {
  if (x > 0) {
    return("Positive")
  } else if (x < 0) {
    return("Negative")
  } else {
    return("Zero")
  }
}
```

Practice:
Improve classify_number : use !is.numeric(x) || is.na(x)
Complete flag_low_proteins()

Control Flow – Functions with multiple conditions

```r
grade_student <- function(score) {
  if (score >= 90) {
    return("A")
  } else if (score >= 80) {
    return("B")
  } else if (score >= 70) {
    return("C")
  } else if (score >= 60) {
    return("D")
  } else {
    return("F")
  }
}
```

```r
#improve grade student to return "Missing" if no grade is passed
#and to return "Invalid" if score is < 0 or > 100
```

Practice:
Improve grade_student()

Rewrite grade_student()
to use for loop

Advanced Functions – Vectorized functions

- Instead of looping over each element one at a time (like we did with `for`), a vectorized function automatically handles the entire vector efficiently and concisely. This is a core strength of R.

Advantages:
- Simpler, cleaner code
- Often much faster (especially on large datasets)

Vectorized version!

```r
double_values <- function(vec) {
  return(vec * 2)
}
nums <- c(1,3,8)
double_values(nums)
```

```r
double_values <- function(vec) {
  out <- c()
  for (val in vec) {
    out <- c(out, val * 2)
  }
  return(out)
}
```

```
> double_values <- function(vec) {
+   return(vec * 2)
+ }
> nums <- c(1,3,8)
> double_values(nums)
[1]  2  6 16
```

Practice: what does "out" look like at every stage of for-loop execution if we call *double_values(c(1,3,8))*?

Advanced Functions – Vectorized functions

- With vectorized operations, **no explicit loop or intermediate buildup** (like `out <- c(out, ...)`) happens. Instead, **the entire vector is operated on at once** under the hood.

Advantages:

- More efficient run time

```r
double_values <- function(vec) {
  return(vec * 2)
}
nums <- c(1,3,8)
double_values(nums)
```

Stage	Operation	Value
1	Input `vec`	`c(1, 3, 8)`
2	Multiply by 2	`vec * 2` → `c(2, 6, 16)`
3	Return	`c(2, 6, 16)`

Advanced Functions – Using . . . (Ellipsis)

- The `...` allows your function to accept an arbitrary number of arguments and pass them along to other functions inside.

Advantages:

• It makes your functions more flexible, especially when wrapping or modifying existing functions like `plot()` or `mean()`.

```r
custom_mean <- function(x, ...) {
  mean(x, ...)
}
```

```r
custom_mean <- function(x, ...) {
  mean(x, ...)
}

custom_mean(c(1, 2, 3, 4, NA), na.rm = TRUE)
# [1] 2.5
```

Advanced Functions – Nested functions

- You can place the **result of one function *inside* another**, allowing you to build more complex logic.

Advantages:
•Reduces intermediate variables
•Makes pipelines more readable when used thoughtfully

```r
sqrt(mean(c(4, 16, 36)))

rounded_avg <- function(x) {
  round(mean(x))
}

rounded_avg(c(1.1, 2.5, 3.9))  # returns 2
```

Why don't we need a return statement here?

Practice: identify all the functions in this block

```r
#### how many functions are in this block ?
library(ggplot2)

df <- data.frame(
  group = c("A", "B", "C"),
  value = c(4, 7, 2)
)

ggplot(df, aes(x = group, y = value)) +
  geom_col(fill = "steelblue") +
  labs(title = "Bar Plot", x = "Group", y = "Value") +
  theme_minimal()
```

solution: identify all the functions in this block

```r
#### how many functions are in this block ?
library(ggplot2)

df <- data.frame(
  group = c("A", "B", "C"),
  value = c(4, 7, 2)
)

ggplot(df, aes(x = group, y = value)) +
  geom_col(fill = "steelblue") +
  labs(title = "Bar Plot", x = "Group", y = "Value") +
  theme_minimal()
```

```r
## identify all the functions in this block answer:
| #  | Function          | What it does                                                                |
|----|-------------------|-----------------------------------------------------------------------------|
  | 1  | `library()`       | Loads the **ggplot2** package                                               |
  | 2  | `data.frame()`    | Creates the data frame `df`                                                 |
  | 3  | `c()`             | Combines individual values into a vector (used twice)                       |
  | 4  | `ggplot()`        | Initializes the plot with data and aesthetics                               |
  | 5  | `aes()`           | Specifies the aesthetic mappings (e.g., `x` and `y`)                        |
  | 6  | `geom_col()`      | Adds bar geometry (bars whose heights represent values)                     |
  | 7  | `labs()`          | Adds labels to the plot (title and axis labels)                             |
  | 8  | `theme_minimal()` | Applies a minimalist visual theme to the plot                               |
```