Functions, iteration, and control flow

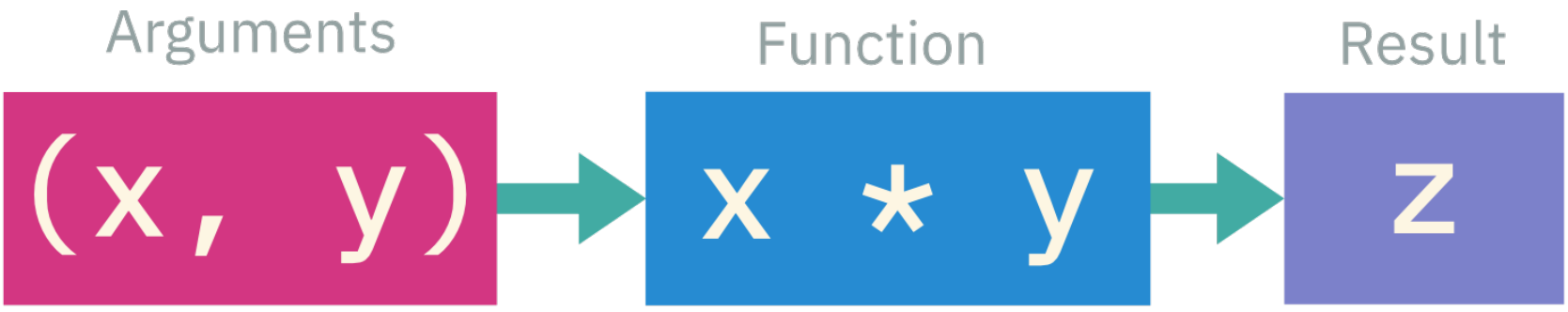
Session overview

1. Functions in R
    - What is a function?
    - How to use functions
    - Writing your own functions
2. Iteration
3. Control flow

Functions in R

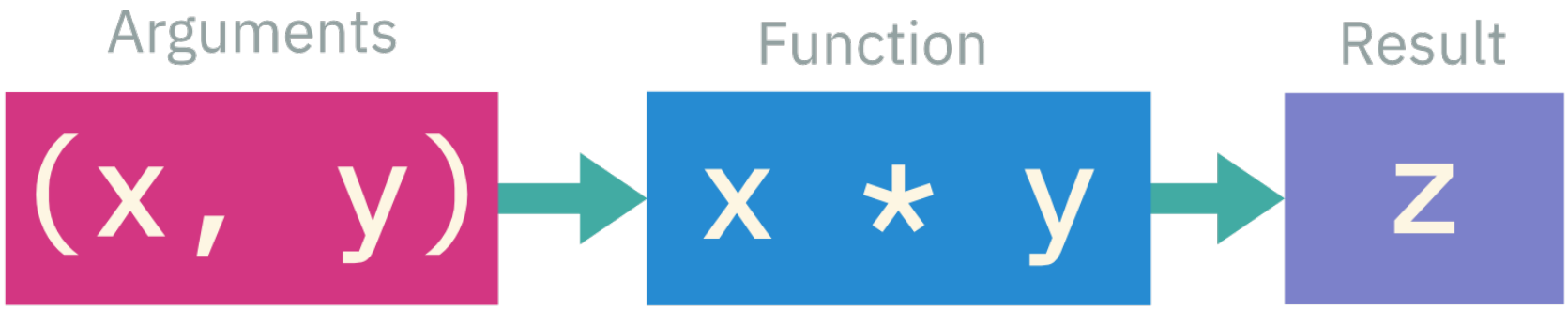
Functions

A function takes inputs, does something with them, and returns the result.



Functions

A function takes inputs, does something with them, and returns the result.



Many functions come with R or are added via packages (e.g., `min`, `mean`, `head`).

But the ability to write your own functions is very powerful.

```r
add_one <- function(x) {
  return(x + 1)
}

add_one(2)
[1] 3
```

```r
multiply <- function(x, y) {
    return(x * y)
}

multiply(2, 3)
[1] 6
```

"A good rule of thumb is to consider writing a function whenever you've copied and pasted a block of code more than twice (i.e. you now have three copies of the same code)."

Hadley Wickham, *R for Data Science (2e)* (2023).

Things you should know about functions

1. Functions are objects.
2. We call a function by typing its name followed by parentheses.
3. Arguments can be named or positional.
4. Arguments can have defaults.
5. We can use `return` within our function to return a result. (But we don't have to).

```
my_exciting_function <- function(x, y) {
    ...
}
```

Things you should know about functions

1. Functions are objects.
2. We call a function by typing its name followed by parentheses.
3. Arguments can be named or positional.
4. Arguments can have defaults.
5. We can use `return` within our function to return a result. (But we don't have to).

```
my_exciting_function()

# What happens if we omit ()?
my_exciting_function
```

Things you should know about functions

1. Functions are objects.
2. We call a function by typing its name followed by parentheses.
3. Arguments can be named or positional.
4. Arguments can have defaults.
5. We can use `return` within our function to return a result. (But we don't have to).

```
# These are equivalent:
rnorm(100, mean = 0, sd = 1)

rnorm(100, 0, 1)
```

Things you should know about functions

1. Functions are objects.
2. We call a function by typing its name followed by parentheses.
3. Arguments can be named or positional.
4. Arguments can have defaults.
5. We can use `return` within our function to return a result. (But we don't have to).

```
my_function <- function(x = 1, y = TRUE) {
    ...
}

rnorm()
```

See help (e.g. `?rnorm`) to learn about default arguments.

Things you should know about functions

1. Functions are objects.
2. We call a function by typing its name followed by parentheses.
3. Arguments can be named or positional.
4. Arguments can have defaults.
5. We can use `return` within our function to return a result. (But we don't have to).

```
my_exciting_function <- function(x, y) {
  return(x * y)
}

my_exciting_function <- function(x, y) {
  x * y
}
```

What's the difference between these three functions?

```
f1 <- function(x, y) {
  return(x * y)
}

f2 <- function(x, y) {
  print(x * y)
}

f3 <- function(x, y) {
  cat(x * y)
}
```

Rolling a dice

```r
roll_dice <- function() {
  sample(1:6, 1)
}

roll_dice()
```

Calculating BMI

```r
calculate_bmi <- function(weight, height) {
  weight / (height^2)
}

# Example: 70kg and 1.75m
calculate_bmi(70, 1.75)
```

A function to format the
coefficients from a regression model

```r
p <- function(x) { sprintf("%.1f", x) }

format_coef <- function(est, lo, hi) {
  glue::str_glue(
    "{p(est)} [{p(lo)}, {p(hi)}]"
    )
}

format_coef(1.4188, 1.2103, 1.6120)
# 1.4 [1.2, 1.6]
```

A function to import Qualitrics extracts (`*.xlsx`):

```
import_qualtrics <- function(path) {
  metadata <- readxl::read_xlsx(path,
    n_max = 2,
    col_names = FALSE,
    .name_repair = "minimal"
  )
  d_name <- metadata[1, ] |>
    as.character() |>
    janitor::make_clean_names()
  d_label <- metadata[2, ] |> as.character()
  data <- readxl::read_xlsx(path, skip = 2,
                            col_names = d_name)
  labelled::var_label(data) <- setNames(d_label,
                                        d_name)
  return(data)
}
```

Iteration

Computers are good at repeating things

Like any programming language, R provides many tools for iteration.

We'll look at two today:

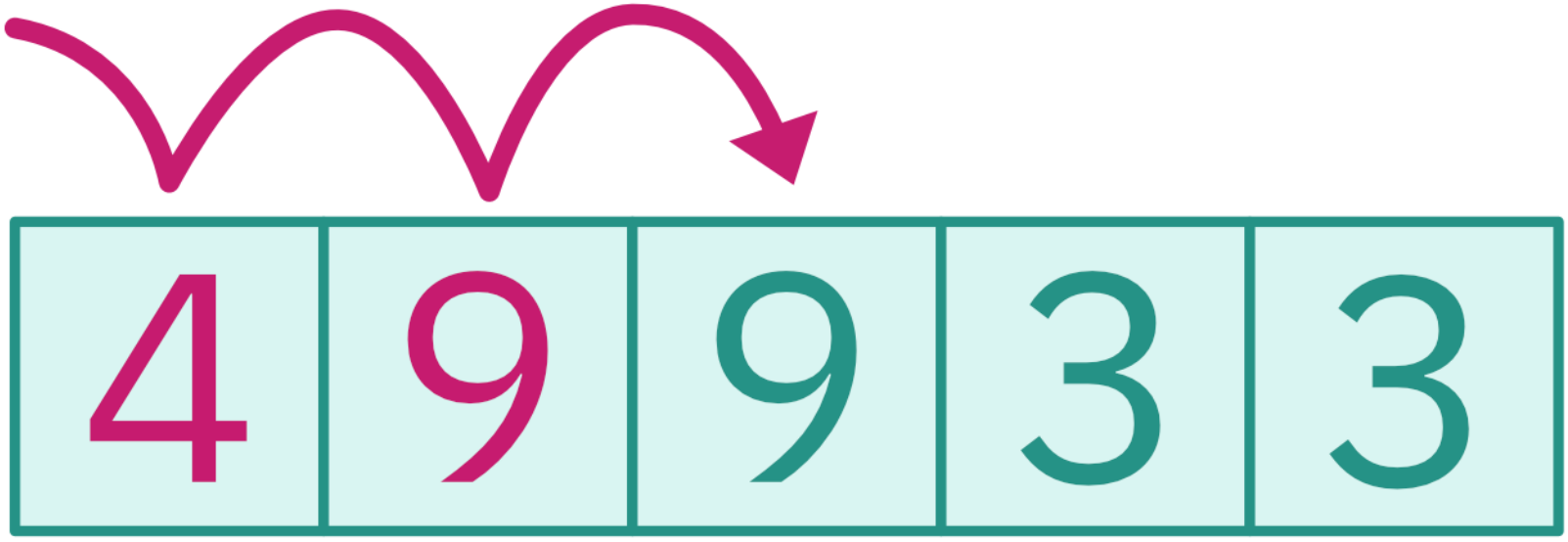
1. **Loops**
2. **Vectorised functions**

In Session 4, we'll extend this to explore the functional programming tools in R — another approach to iteration.

Loops

Iterate through a sequence of things, and do some operation at each iteration.

For example, we might loop through a numeric vector and perform some calculation on each element.



Loops

In R, we use the **`for`** statement:

```
for (THING in SEQUENCE) {
  DO SOMETHING
}
```

Loops

In R, we use the **`for`** statement:

```
for (THING in SEQUENCE) {
  DO SOMETHING
}
```

For example:

```
for (i in 1:10) {
    print(i)
}
```

```
> for (i in 1:10) {
+   print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

`i` refers to the current element of the sequence.

You don't have to use `i`:

```
for (thing in 1:10) {
  print(thing)
}
```

```r
# Loop over elements of a vector
months <- c("January", "February", "March")
for (i in months) {
    print(i)
}

# Nested loops
for (i in 1:10) {
    for (j in seq(5, 25, 5)) {
        print(i * j)
    }
}
```

Using loops to update a list or vector

e.g., iterate through a sequence, perform a calculation, and store the result in a new vector.

```
# Define an empty vector to store results
> results <- vector(mode = "numeric",
>                   length = 10)
> for (i in 1:10) {
>   results[i] <- sqrt(i)
> }

> results
 [1] 1.000000 1.414214 1.732051
 [4] 2.000000 2.236068 2.449490
 [7] 2.645751 2.828427 3.000000
[10] 3.162278
```

'Vectorised' functions

- Many functions in R are vectorised.
- This means they operate on all elements of a vector without the need for looping.

For example, `mean()` or `min()`:

```
> x <- c(8, 1, 12, 5, 21, 4)
> mean(x)
[1] 8.5

> min(x)
[1] 1
```

Use vectorised functions wherever possible.

```
> library(rbenchmark)
> x <- sample(1:100, 1e4, replace = TRUE)
> head(x)
[1] 79  4 13 25 16 30

> my_mean <- function(x) {
>   total <- 0
>   for (i in x) {
>     total <- total + i
>   }
>   return(total / length(x))
> }

> benchmark("My function" = { my_mean(x) },
>           "Vectorised function" = { mean(x) },
>           replications = 1e5)

                 test replications elapsed relative
1         My function       100000  10.361    8.563
2 Vectorised function       100000   1.210    1.000
```

Loops are almost never the right answer.

- Use vectorised functions.
- Use functional programming (see *Session 4*).

It is extremely rare that you need to loop over rows in your data.

Example:

1. Calculating a mean.
2. Updating a column.

Control flow

Control flow

1. `if`/`else` statements

   If condition is `TRUE`, then perform an action.

2. `while`

   Perform an action while the condition remains `TRUE`.

3. `repeat`

   Repeat an action forever.

We're only going to look at `if`/`else`.

`if` / `else`

```
if (CONDITION is TRUE) {
   DO SOMETHING
} else {
   DO SOMETHING ELSE
}
```

`if` / `else`

```
if (CONDITION is TRUE) {
   DO SOMETHING
} else {
   DO SOMETHING ELSE
}
```

```
if (format(Sys.time(), "%H") < 12) {
   print("Good morning")
} else {
   print("Good afternoon")
}
```

```r
if (score >= 50) {
    grade <- "pass"
} else {
    grade <- "fail"
}
```

```
if (score >= 50) {
    grade <- "pass"
} else {
    grade <- "fail"
}
```

We can have multiple conditions:

```
if (score >= 70) {
    grade <- "A"
} else if (score >= 60) {
    grade <- "B"
} else if (score >= 50) {
    grade <- "C"
} else {
    grade <- "fail"
}
```

But don’t overuse this — it quickly becomes unreadable.

We’ll introduce `case_when` and `recode_values` in a later session. These are a better option if you have multiple conditions to evaluate.

For example:

```
library(dplyr)

case_when(
  score >= 70 ~ "A",
  score >= 60 ~ "B",
  score >= 50 ~ "C",
  .default ~ "Fail"
)
```

A common use case: Input validation

It's good practice to verify that function inputs are correct.

- Required inputs are not missing.
- Inputs are of the correct type (e.g., numeric, character).

```
calculate_bmi <- function(weight, height) {
  if (is.na(weight) || is.na(height)) {
    stop("Inputs must not be missing")
  }
  if (!is.numeric(weight) ||
      !is.numeric(height)) {
    stop("Inputs must be numeric")
  }
  weight / (height^2)
}
```