

# **Coding in R I: syntax; variables, arrays, and data frames; for loops, apply**

Part 1: Naming objects & entering data

Part 2: Coding considerations

Part 3: Tidy data

Part 4: Data structures

Part 5: Loops and Apply

# First, some terms:

The diagram illustrates the components of the R code snippet `sloth_speed <- sloth_df %>% select(id, age, speed)` using handwritten annotations and brackets:

- OBJECT** (pink): A bracket above `sloth_speed`.
- assignment operator** (purple): A label with an arrow pointing to the `<-` operator.
- STARTING DATA FRAME** (green): A bracket above `sloth_df`.
- pipe operator** (orange): A bracket above the `%>%` operator.
- FUNCTION** (blue): A bracket below `select`.
- ARGUMENTS** (red): A bracket below `(id, age, speed)`.

```
sloth_speed <- sloth_df %>% select(id, age, speed)
```

# **Part 1: Naming things & entering data**



“Call him Voldemort, Harry. Always use the proper name for things.” - Albus Dumbledore, *Harry Potter and the Sorcerer’s Stone* by JK Rowling

# Naming things

When naming variables, observations, data frames, or files, make them:

1. Meaningful
2. Consistent
3. Concise
4. Code & coder friendly

# Naming things

When naming variables, observations, data frames, or files, make them:

1. Meaningful

2. Consistent

3. Concise

4. Code & coder friendly

- Names of variables, data frames, and files should not be so generic/vague that a user must need a glossary to know what they contain
- Names should be specific to the data/experiment/project, and the more intuitive their interpretation the better
- **Bad examples:** File-1.xlsx, file-2.csv, indicator1, indicator2, ExperimentA.R, ExperimentB.R
- **Better examples:** taco\_nutrients.csv, ca-demographics, mice\_1a\_mass, sb\_channel\_spatial.shp

# Naming things

When naming variables, observations, data frames, or files, make them:

1. Meaningful

2. Consistent

3. Concise

4. Code & coder friendly

- Keep names **perfectly identical** for identical entries (e.g. “burrito-32” and “Burrito 32” are completely different things to R)
- Be consistent **across data frames** - your life will be easier if you have year called ‘year’ in both sets, instead of ‘year’ in one and ‘YEAR\_NEW’ in the other
- Use logical suffixes (if necessary), consistently formatted. Like: temp\_water\_surface, temp\_water\_sub, temp\_water\_bottom

# Naming things

When naming variables, observations, data frames, or files, make them:

1. Meaningful
2. Consistent
3. Concise
4. Code & coder friendly

- Balance meaningfulness w/conciseness
- Better to be descriptive than not know what a variable is
- Longer names = tedious coding, but less effort to look through metadata for column/identifier names
- **Bad examples:** 'First dive temp readings Celsius', greatblueheron\_observations\_2019\_09\_20, 'Allison final figures version 4.xlsx'
- **Better examples:** goleta\_temp, USTotalPop, PercClay



# Naming things

When naming variables, observations, data frames, or files, make them:

1. Meaningful
2. Consistent
3. Concise

4. Code & coder friendly

- Avoid punctuation (% , ! , ~ , ( ) , #) in names - more challenging to type & can mean things in code that you don't want it to (or just break it)
- Avoid spaces (makes coding much more difficult)
- Generally, avoid starting object names with numbers (but could be useful for file names in sequence)
- Pick (and be consistent with) a choice of case, like:
  - lowercase\_snake\_case
  - camelCase (my favorite)
  - UpperCamelCase
  - SCREAMING\_SNAKE\_CASE

# Other naming considerations:

- Avoid object names that are common/used function names (e.g., don't name something 'filter')
- Make a name uniquely searchable (would I be able to find this if I searched for it, e.g. in a GitHub repo?)
- Consider making object names nouns, & function names verbs
- It's never the end of the world if you give something a bad name, but it will save you time & effort to strive for good names

# Entering things

We'll consider three bins for now:

- **Quantitative data:** numeric observations, can be continuous (measured) or discrete (usually counts or ordinal data)
- **Nominal data:** labels, usually in words (e.g. “purple”, “blue”, or “orange”)
- **Date/times:** a time variable with weird formatting that Excel is determined to mess with

# Entering things

- The **outcomes** for a variable (whether values or descriptions) should exist alone in a column
- Be **very consistent** when entering descriptions (e.g. “Purple” v. “purple” v. “purple\_”)
- Avoid formatting and symbols (if it’s hard to type, then it’s hard to type...and might cause issues beyond that)
- Put any additional information (units, notes, etc.) in columns separate from the value/description
- If there are missings, enter the **same exact thing** for each missing value (common: - 9999, NA, -- or -)

Bad:

Species	Berry Mass
manzanita	0.24 g
Manzanita	0.31 grams
currant	0.15 g (note: disease observed)
currant	0.17 g
manzanita	not measured (equipment malfunction)
currant	0.12 grams
CURRENT	-9999

Better:

species	berry_mass_g	notes
manzanita	0.24	NA
manzanita	0.31	NA
currant	0.15	disease observed
currant	0.17	NA
manzanita	NA	equipment malfunction
currant	0.12	NA
currant	NA	NA

# Entering dates/times



- International Organization for Standardization (ISO) [8601](#)
  - Dates: eliminate ambiguity with **yyyy-mm-dd** format (this is called 'extended format' - unextended is just `yyymmdd`)
    - 2019-09-30
  - Times as **hh:mm:ss.ffff** (hours, minutes, seconds, fractions of seconds)
    - 06:30:22.4033
  - Datetimes: **yyyy-mm-ddThh:mm:ss.ffff**
    - 2001-03-12T14:38:02.8725

# Recap: Data types

```
# Load library for datetime
library(lubridate)

# Make fake dataframe about dogs
dogs <- data.frame(
  birth_date = as.Date(c("2020-01-15", "2019-06-23", "2021-11-05")), # Date
  vet_visit = ymd_hms(c("2023-05-01 10:30:00", "2023-05-03 14:15:00", "2023-05-07 09:45:00")), # Datetime
  name = c("Buddy", "Luna", "Max"), # Character
  breed = factor(c("Labrador", "Beagle", "Poodle")), # Factor
  weight_kg = c(30.5, 12.2, 8.7), # Numeric
  vaccinated = c(TRUE, FALSE, TRUE) # Logical
)

print(dogs)
str(dogs)
```

# Recap: Data types

```
> print(dogs)
```

	birth_date	vet_visit	name	breed	weight_kg	vaccinated
1	2020-01-15	2023-05-01 10:30:00	Buddy	Labrador	30.5	TRUE
2	2019-06-23	2023-05-03 14:15:00	Luna	Beagle	12.2	FALSE
3	2021-11-05	2023-05-07 09:45:00	Max	Poodle	8.7	TRUE

```
> str(dogs)
```

```
'data.frame': 3 obs. of 6 variables:
```

```
$ birth_date: Date, format: "2020-01-15" "2019-06-23" "2021-11-05"
```

```
$ vet_visit : POSIXct, format: "2023-05-01 10:30:00" "2023-05-03 14:15:00" "2023-05-07 09:45:00"
```

```
$ name      : chr  "Buddy" "Luna" "Max"
```

```
$ breed     : Factor w/ 3 levels "Beagle","Labrador",...: 2 1 3
```

```
$ weight_kg : num  30.5 12.2 8.7
```

```
$ vaccinated: logi  TRUE FALSE TRUE
```



## Part 2: Some coding considerations

See: [The tidyverse Style Guide](#) by Hadley Wickham

## Part 2: Some coding considerations

*“Good coding style is like correct punctuation: you can manage without it, but it sure makes things easier to read.”*

[The tidyverse Style Guide](#) by Hadley Wickham

# Why?

- Consistent syntax and strict organization will help you and your collaborators work with, update, and provide feedback on your code.
- Consistency is also useful when troubleshooting - if something looks weird, you'll start spotting it more quickly

# Who is a collaborator?

- Your future self ([Lowndes et al. 2017](#))
- Current collaborators/coworkers
- Future collaborators

- You will develop your own style of coding. But there are some standards that will make it easier for you & everyone else to follow what you did.
- Some of these things are true for coding in general, & some are specific to tidyverse-style functional programming

## Some important things:

- Functionality (...it has to work correctly)
- Reproducibility (.Rproj, {here}, scripts, R Markdown)
- Organization (subsections, line spacing)
- Annotation (clear, useful comments)
- Consistency (spacing, syntax, naming)
- Elegance (%>%, {purrr}, automating repeated processes, etc.)

# Code structure

- When possible, avoid lines of code  $> 80$  characters long
- For extended code, consider vertical structure instead of paragraph style

## EW:

```
object_name <- df %>% filter(col_a == "yes", col_b ==  
  "burritos", col_c != "eggplant") %>% select(col_b:col_e) %>%  
  mutate(new_col = col_f + col_g) %>% group_by(col_b, col_d) %>%  
  summarize(new_col_2 = mean(new_col))
```

## PHEW:

```
object_name <- df %>%  
  filter(col_a == "yes",  
         col_b == "burritos",  
         col_c != "eggplant") %>%  
  select(col_b:col_e) %>%  
  mutate(new_col = col_f + col_g) %>%  
  group_by(col_b, col_d) %>%  
  summarize(new_col_2 = mean(new_col))
```

# Consider pressing 'Return':

- After any pipe operator %>%
- For functions containing a lot of arguments, after the comma following each argument

```
ggplot(df, aes(x = temp, y = salinity)) +  
  geom_point(color = "blue",  
             size = 2,  
             pch = 18,  
             alpha = 0.2)
```

- After any + sign when creating a graph with ggplot2

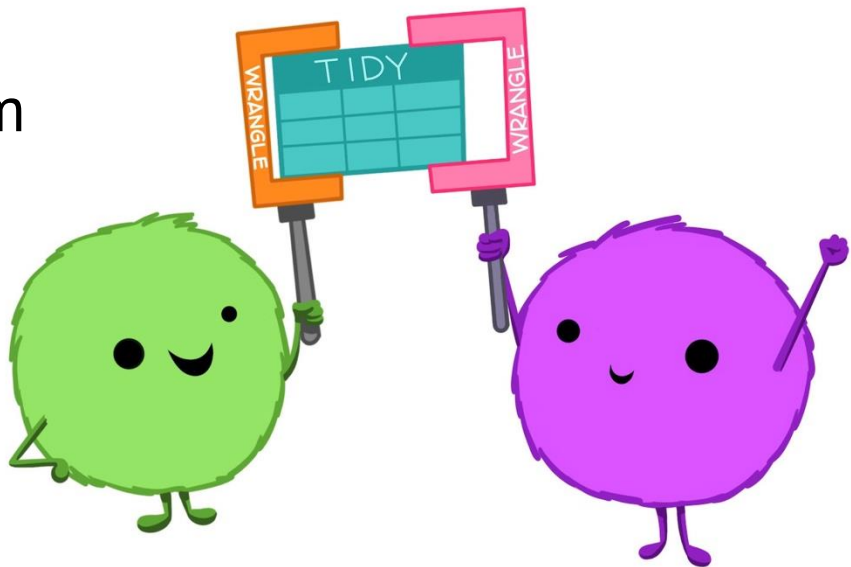


# Code spacing (from the [Tidyverse style guide](#))

- *“Always put a space after a comma, never before”*  
**Rough:** `select(data = df, temp , ph , salinity)`  
**Better:** `select(data = df, temp, ph, salinity)`
- *“Don’t put spaces inside or outside parentheses for regular function calls.”*  
**Rough:** `filter ( data = df, temp, ph, salinity )`  
**Better:** `filter(data = df, temp, ph, salinity)`
- *“Most infix operators (==, +, -, <-, etc.) should be surrounded by spaces.”*  
**Rough:** `pika_mass <- age*4.3+0.2`  
**Better:** `pika_mass <- age * 4.3 + 0.2`

# Part 3: Tidy data

[Tidy Data](#) by Hadley Wickham



# What is tidy data?

From [R for Data Science](#) by Grolemund & Wickham:

To be “tidy”:

1. Each **variable** is a column.
2. Each **observation** is a row.
3. Each **value** in its own cell.

A **variable** is a characteristic that is being measured, counted or described with data. Like: **car type**, **salinity**, **year**, **population**, or **whale mass**.

An **observation** is a single “data point” for which the measure, count or description of one or more variables is recorded. For example, if you are recording variables *height*, *mass*, and *color* of dragons, then **each dragon** is an observation.

A **value** is the recorded measure, count or description of a variable.

Tidy data schematic, from [R for Data Science](#) by Grolemund & Wickham:

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	128042583

variables

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	128042583

observations

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	128042583

values

# Part 4: Common data structures in R

- **Vector:** a combined list of elements
- **List:** a combination of vectors (class of vectors can differ)
- **Data frame:** a list of vectors w/ same # elements

**We often read in external data.**  
**We can also make vectors & DFs in R.**

Make a vector by combining elements with `c()`:

```
burrito_lbs <- c(0.8, 1.1, 1.3, 0.6)
```

# There are a number of ways to make DFs:

```
species <- c("cat", "dog")  
pet_color <- c("orange", "brown")  
pet_age <- c(4, 2)
```

You can use `rbind()` - “row bind” to bind vectors together in rows, but usually you want to combine vectors as columns. Use `data.frame()` to bind vectors (of same length) together as columns into a single df:

```
pets_df <- data.frame(species, pet_color, pet_age)
```

```
> pets_df
```

	species	pet_color	pet_age
1	cat	orange	4
2	dog	brown	2



# What in the world is a tibble?

1. Tibbles are data frames
2. Usually used interchangeably
3. Some updated functionality
4. Mostly don't worry about it
5. Coerce w/ `as_tibble()` ...



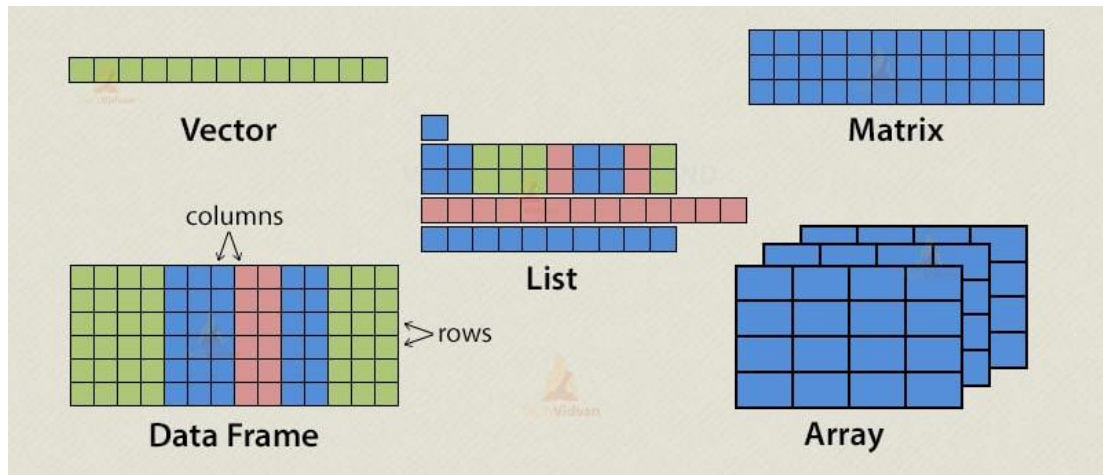
# Recap

## Variables/Objects

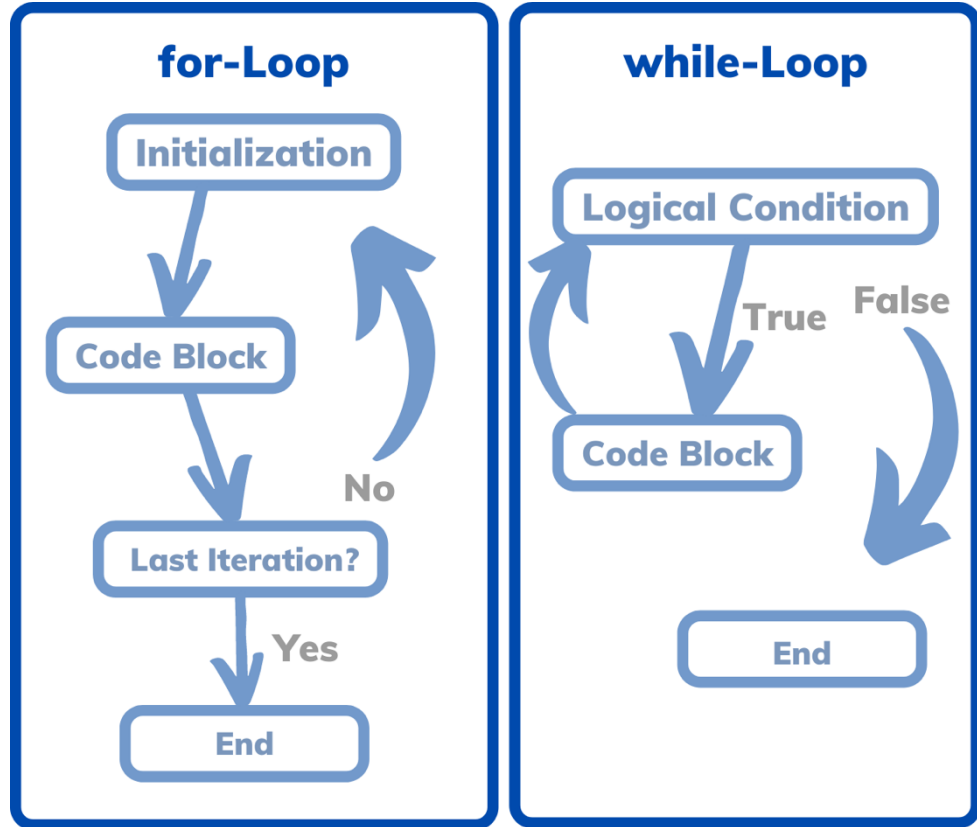
### Vectors

Matrix/Arrays

Dataframes/Tibbles



# Part 4: Loops



# For Loops

A for loop in R is used to repeat a block of code a fixed number of times.

General structure:

```
for (variable in sequence) {  
  # code to execute  
}
```

*My rule of thumb is, once I copy+paste code 3 times, I convert it to a loop/apply/function*

```
dog_names <- c("Buddy", "Luna", "Max")  
  
for (name in dog_names) {  
  print(paste("Good dog:", name))  
}
```

```
> for (name in dog_names) {  
+   print(paste("Good dog:", name))  
+ }  
[1] "Good dog: Buddy"  
[1] "Good dog: Luna"  
[1] "Good dog: Max"
```

# For Loops

A for loop in R is used to repeat a block of code a fixed number of times.

General structure:






```
for (variable in sequence) {  
  # code to execute  
}
```

```
dog_birth_years <- c(2020, 2018, 2019)  
current_year <- 2025
```

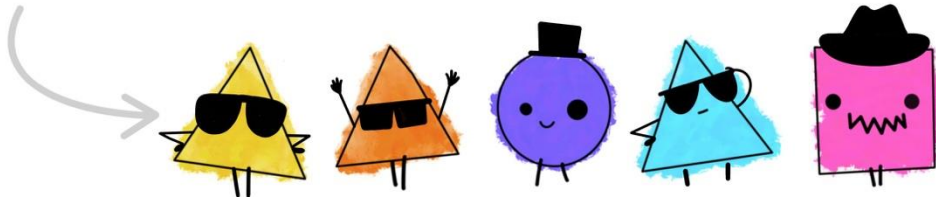
```
for (year in dog_birth_years) {  
  age <- current_year - year  
  print(paste("Dog age:", age))  
}
```

```
dog_weights <- c(30.5, 12.2, 8.7)  
total_weight <- 0
```

```
for (w in dog_weights) {  
  total_weight <- total_weight + w  
}
```

`parade = c(`  `,`  `,`  `,`  `,`  `)`

```
for (monster in parade) {  
  if (shape(monster) == triangle) {  
    monster_style = monster + sunglasses  
  }  
  else {  
    monster_style = monster + hat  
  }  
  print(monster_style)  
}
```



# For Loops

- Repeating actions for each item in a list or vector
- Iterating over datasets
- Performing cumulative calculations

For loops are *very easy* to troubleshoot

# Apply Statements

The apply family of functions in R are used to apply a function to elements of data structures.

- `apply()`: works on matrices/arrays
- `lapply()`: returns a list
- `sapply()`: returns a simplified vector or matrix

```
dog_names <- c("Buddy", "Luna", "Max")

lapply(dog_names, function(name) {
  paste("Good dog:", name)
})
```





## Reasons to use apply, lapply, sapply, etc.

### 1. Cleaner, more concise code

```
dog_names <- c("Buddy", "Luna", "Max")

result = lapply(dog_names, function(x) paste("Good dog:", x))
# vs.

result <- list()
for (i in seq_along(dog_names)) {
  result[[i]] <- paste("Good dog:", dog_names[i])
}
```

### 2. Functional style fits R better

R was built around vectors, lists, and applying functions. The apply family aligns with that design

### 3. Less bookkeeping

You don't need to pre-allocate and index into results manually (`result[i] <- ...`). `lapply()` handles it.

### 4. Readability

Other R users will immediately recognize that you're "applying a function to each element" when they see `lapply()`.

### 5. Integration with data frames

Functions like `apply()` and `tapply()` make it easy to work with rows, columns, or groups in a way that's harder with raw for loops.

## ✅ Reasons to use a for loop

### 1. Step-by-step debugging

Easier to insert `print()` or `browser()` inside a loop if something goes wrong.

### 2. Complex logic

If the operation isn't just "apply this function to each element," but involves conditionals, accumulating multiple results, or breaking early, `for` can be clearer.

### 3. Speed (sometimes)

With small vectors, `lapply()` is as fast as `for`. With very large, more complex operations, differences shrink — and sometimes a well-written `for` loop can be faster.



- A **for loop** is like *walking each dog one by one yourself*.
- An **apply** function is like *sending them all through the same doggy daycare routine automatically*. Less micromanaging, but you trust the system.