Foundations of Data Science for Biologists

Functions and flow control in R

BIO 724D

09-OCT-2023

Instructors: Greg Wray, Jesse Granger, Paul Magwene

Functions in R

Functions: motivation

Why use existing functions?

- Saves time compared to writing your own code
- Far more likely to run without errors and produce expected results
- Much easier for others to understand your code
- Access to procedures that would otherwise be impractical

Why write your own functions?

- Makes your code more readable
- Reduces errors and incorrect results
- Makes it easier to debug code
- Makes it easier to re-use code

Flexibility of R functions

R functions allow for quite a lot of flexibility and power

- Zero, one argument, or many arguments
- Optional arguments with default values
- Arbitrary number of arguments (indicated as `...` ; **variadic** function in CS terminology)
- Functions with no name (**lambda** functions in CS terminology)

Some examples

- No arguments (the argument is assumed): `date()`, `getwd()`
- One scalar argument (vector of length 1): `sqrt()`, `is.logical()`
- One vector or list argument (require length > 1): `max()`, `sum()`, `sort.list()`
- Multiple arguments of the same data type: `paste0()`, `rbind()`
- Multiple arguments of different types: `c()`, `dplyr::mutate()`

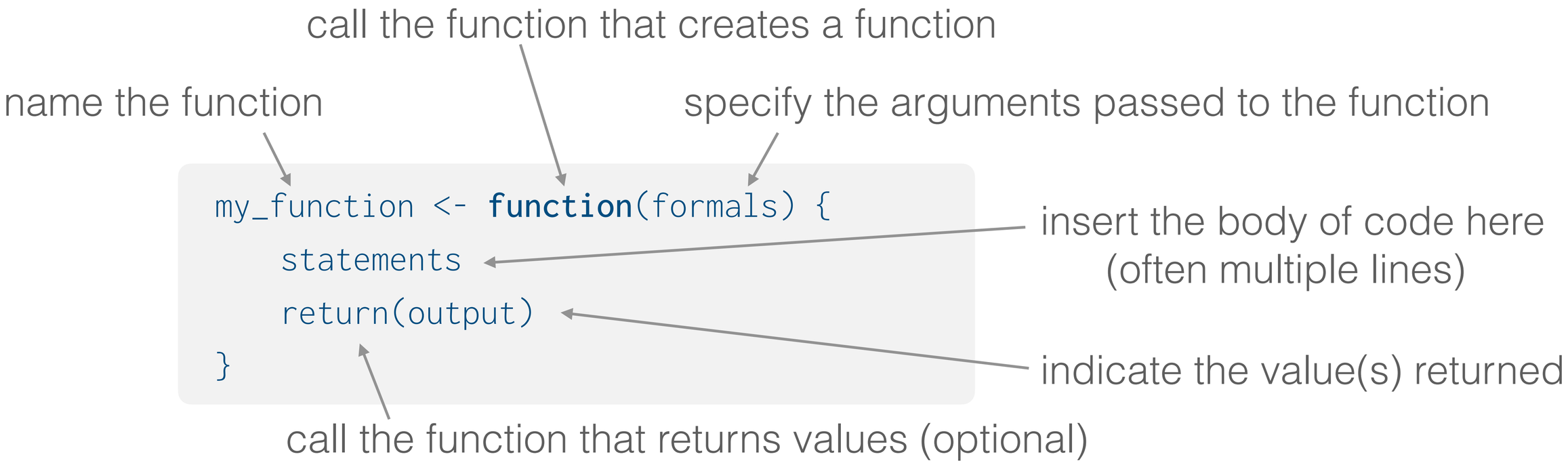
Functions: definition

Functions in R have three components

Formals: the list of arguments that determine how you call the function

Body: the code inside the function that tells it what to do

Environment: the data structure that determines how the function finds values



Understanding arguments

The first argument is usually the data

- E.g, most TidyVerse verbs expect a data frame as the first argument
- Notable exceptions: many modeling functions (e.g., linear models)

Default values can be assigned to arguments

- Provide a name and default value; e.g., `na.rm = FALSE`
- In general, default values should be the most common or enforce safety
- Any argument with a specified default value is automatically **optional**

Variadic functions can be specified using … (dot dot dot; also called varargs)

- Indicates that an arbitrary number of arguments are allowed

E.g., R's help page for the function `sum()` shows usage as: `sum(…, na.rm = FALSE)`

Returning results and halting execution

Implicit: by default, the value of the last expression evaluated is returned

- Useful for simple functions where it is clear what is being done

Explicit: to specify the value(s) that are returned, use a `return()` statement

- Clearer when returning multiple values and required for returning intermediate values
- Useful for functions where the code block is more complex
- Note that `return()` can only return 1 data object
- For multiple values, first create a vector, list, or other structure pass it to `return()`

To raise an error while executing the code block, use a `stop()` statement

- Immediately halts execution of the program; optionally, prints an error message
- Useful for debugging and for trapping incorrect input during runtime

Environments

In computing terms, an **environment** is:

- A list of all variables, objects, and functions available to your code
- A virtual space where where those variables, objects, and function are available
- Shown in the Environment panel in RStudio

Every time you start R, it sets up a new environment called the **global environment**

- That environment starts empty, but you can create objects

Every time you run code, R sets up a new environment

- That job by default inherits the top-level environment

Every time you call a function, R sets up a new environment

- That function by default inherits the environment of the program that calls it

Working with environments

New environments have access to everything in their parent environment

- But the reverse is not true!

Much of what happens in a function remains exclusive to that environment by default:

- Any existing variable that is assigned a new value
- Any newly defined variable
- Any function created within the function

To override these behaviors, include the variable or function in a `return()` list

Why do we need environments?

- To avoid accidental changes in variables due to name collisions
- To insulate the calling program from events inside functions (especially errors)

Comment your code

Use comments freely and often

- Re-running and debugging is much easier
- Re-using your code is much simpler
- Sharing your code is less painful
- Your future self will thank you!!

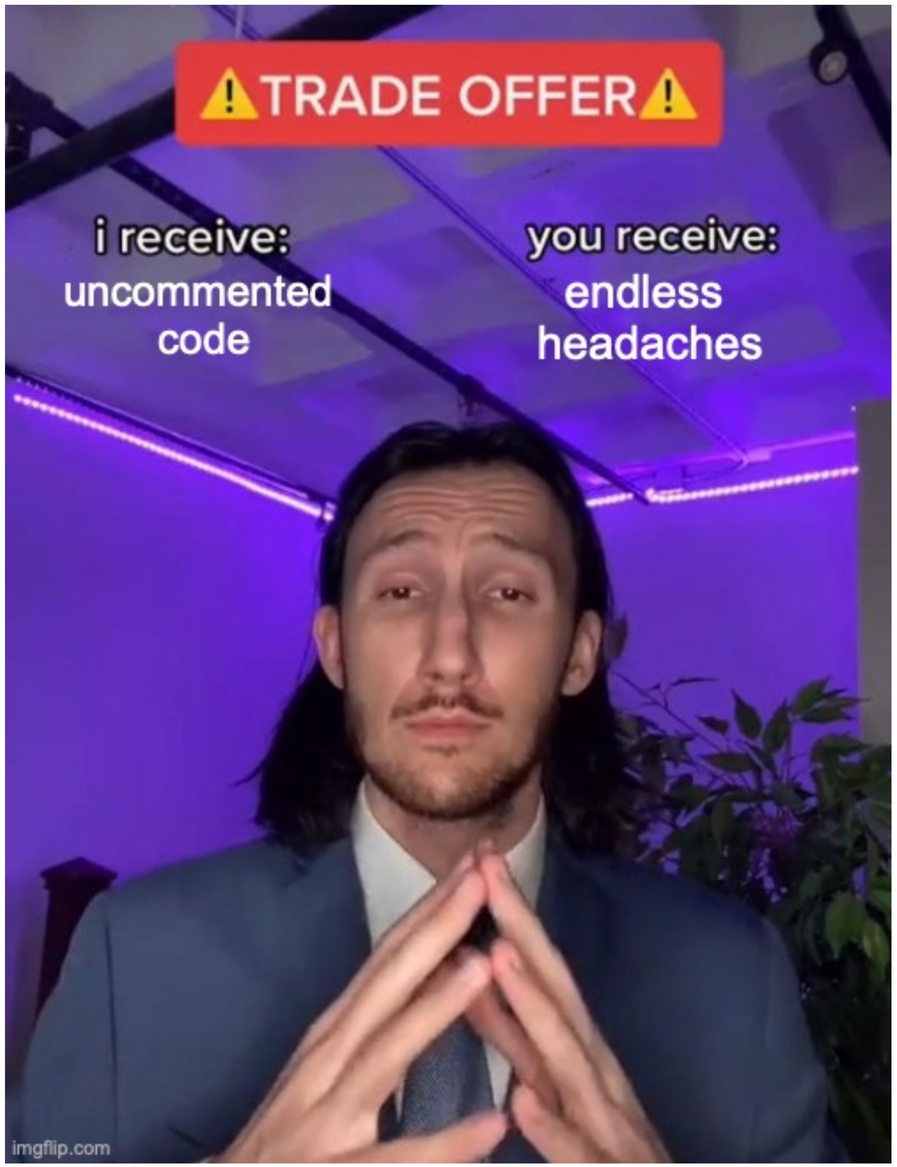
Use headers for programs and functions

- Input file requirements and formats
- Outputs: variables, files, actions

Break your code up into sections

- E.g.: file wrangling, analysis, plots/tables

Said no one: I used too many comments



Exercise

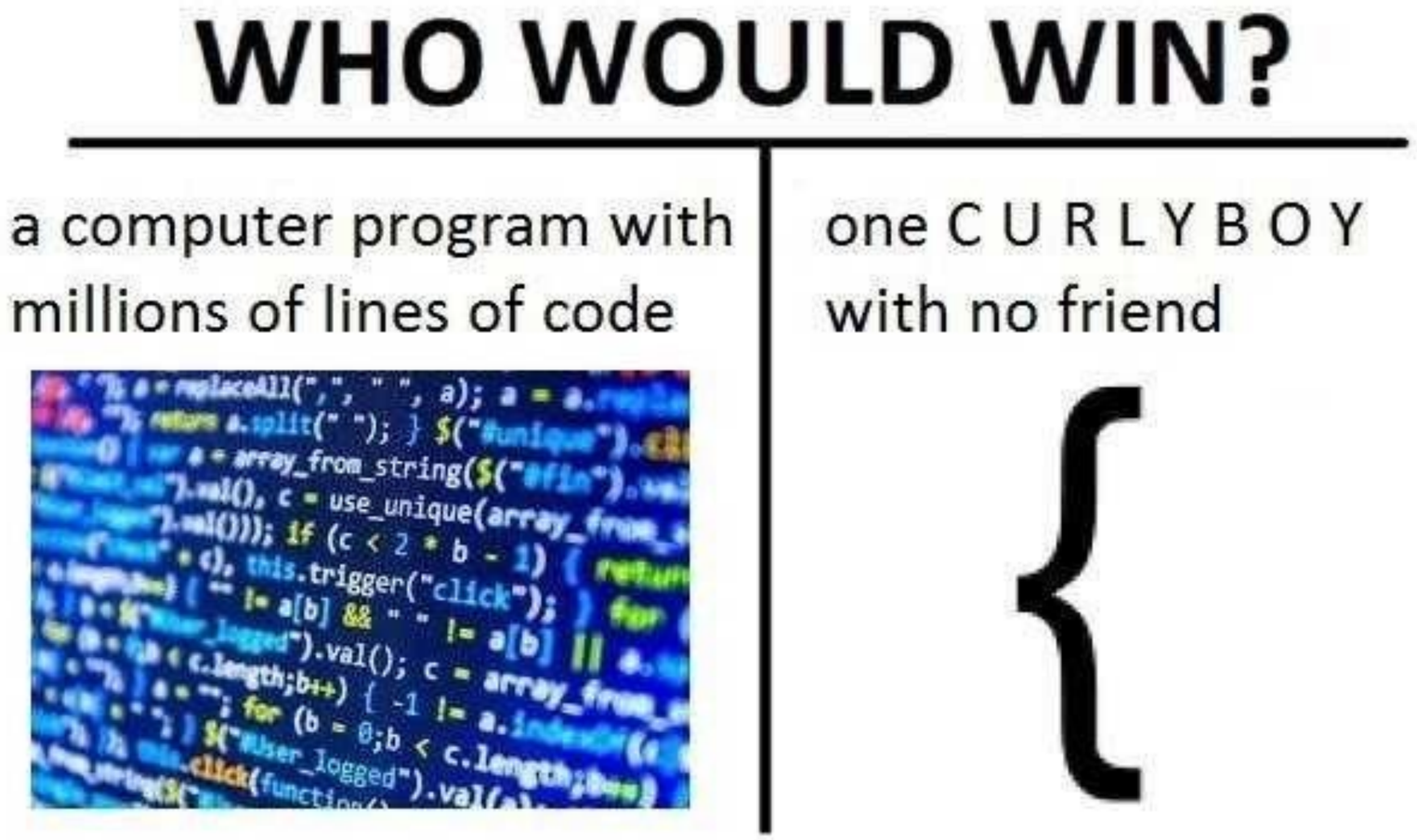
Write a function that returns the area of a circle based on radius (input value)

Write a function that returns both the area of a circle and the volume of a sphere
for the same radius

Flow control

Goals for Today

- Gain an intuitive understanding of if and if-else statements
- Gain a better understanding of for-loops



If and if else

```
if (condition = TRUE) {
  “Do task”
} else {
   “Do task”
}
```

If and if else

```
if (your shirt == black) {
 stand up
}
```

If and if else

```
if (your shirt == black) {
  stand up
} else  {
  touch your nose
}
```

If and if else

```
if (your shirt !=black) {
  stand up
} else {
  touch your nose
}
```

If and if else

```
if (your shirt !=black && your hair==brown) {
  stand up
} else {
  touch your nose
}
```

If and if else

```
if (your shirt != black || your hair==brown) {
  stand up
} else {
  touch your nose
}
```

If and if else

```
if (the number of letters in your first name is > 5) {
  stand up
} else {
  touch your nose
}
```

If and if else

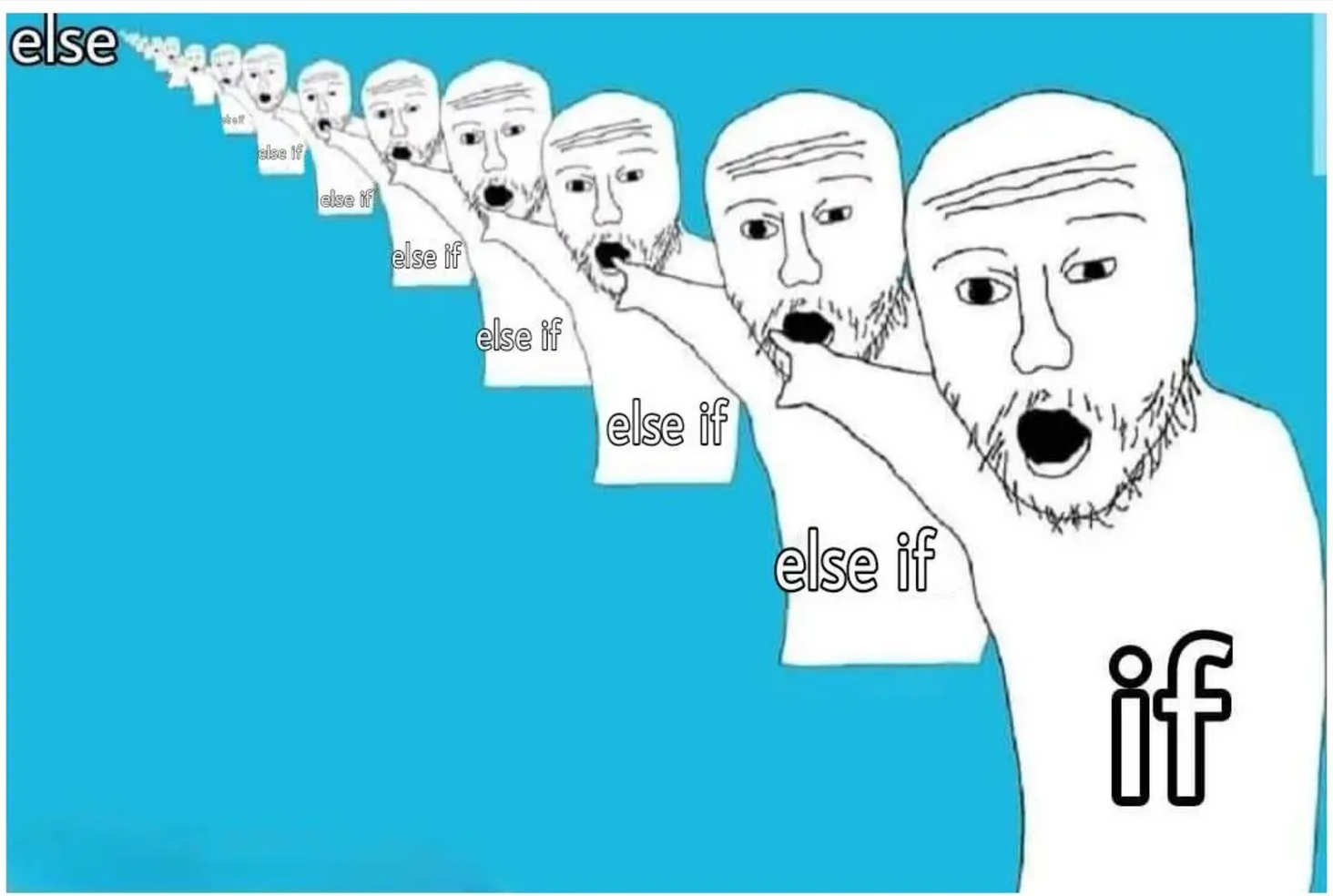
```
if (the number of letters in your first name is >= 5) {
  stand up
} else {
  touch your nose
}
```

If and if else

```
if (the number of letters in your first name is < 5) {
  stand up
} else if (the number of letters in your first name <= 7) {
  touch your nose
} else {
  raise your hand
}
```

If and if else

```
if (the number of                                    sses) {
  stand up
} else if (the num
  touch your nose
} else if (the num                                    hair != brown) {
  raise your hand
} else {
  sit on the floor
}
```



In class exercise – groups of 4

Building a “coin-toss” if-else statement:

runif(1) will generate a random floating-point number between 0 and 1

print() will output to the console

Write an if-else statement that simulates a “coin flip” using x=runif(1) and print() where it prints “heads” if x is less than 0.5 and “tails” if x is 0.5 or larger.

For loops

```
for ( variable in vector ) {

}
```

For loops

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

}
```

For loops

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

For loops—Indexing

```
class=c(Emma, Madison, Runlong, Katie, Miao, Bertram, Hannah, Jimmy, Desireé,
Annabelle, Amanda, Litong)

for ( i in 1:12 ) {
  STANDUP(class[i])
}
```

For loops—Indexing

```
class=c(Emma, Madison, Runlong, Katie, Miao, Bertram, Hannah, Jimmy, Desireé,
Annabelle, Amanda, Litong, Emily)

for ( i in 1:12 ) {
  STANDUP(class[i])
}
```

For loops—Indexing

```
class=c(Emma, Madison, Runlong, Katie, Miao, Bertram, Hannah, Jimmy, Desireé,
Annabelle, Amanda, Litong, Emily)

for ( i in 1:length(class) ) {
  STANDUP(class[i])
}
```

Combining for loops and if else statements

```
class=c(Emma, Madison, Runlong, Katie, Miao, Bertram, Hannah, Jimmy, Desireé,
Annabelle, Amanda, Litong, Emily)

for ( i in 1:length(class) ) {
  if (class[i] first name <=5 letters) {
   stand up
 } else {
   touch your nose
 }
}
```

Combining for loops and if else statements

```
class=c(Emma, Madison, Runlong, Katie, Miao, Bertram, Hannah, Jimmy, Desireé,
Annabelle, Amanda, Litong, Emily)

for ( i in 12 ) {
  if (class[i] first name <=5 letters) {
   stand up
 } else {
   touch your nose
 }
}
```

Combining for loops and if else statements

```
class=c(Emma, Madison, Runlong, Katie, Miao, Bertram, Hannah, Jimmy, Desireé,
Annabelle, Amanda, Litong, Emily)

for ( i in 2:5 ) {
  if (class[i] first name <=5 letters) {
   stand up
 } else {
   touch your nose
 }
}
```

Combining for loops and if else statements

```
class=c(Emma, Madison, Runlong, Katie, Miao, Bertram, Hannah, Jimmy, Desireé,
Annabelle, Amanda, Litong, Emily)

for ( i in c(3,12,1,5,7,9,8,10,12)) {
  if (class[i] first name <=5 letters) {
   stand up
 } else {
   touch your nose
 }
}
```

for loops for repeating a task and saving the output

```
new.vector = c()

for ( i in 1:10) {
 x = mean(runif(10))
 new.vector=c(new.vector,x)
}
```

break statements

```
new.vector = c()

for ( i in 1:10) {
 x = mean(runif(10))
 new.vector=c(new.vector,x)
 if (x > 1) {
   break
 }
}
```

Words of Wisdom

You can accidentally get trapped in an infinite for loop or, have a broken loop that takes FOREVER to run. SO:

1) ALWAYS, ALWAYS, ALWAYS save your code before you run a loop

2) Add a “progress bar” to your loop using print(i) to make sure it is running and isn’t trapped in an error (this also lets you see how far you’ve gotten). For Example:

```
new.vector = c()
for ( i in 1:10) {
 x = mean(runif(10))
 new.vector=c(new.vector,x)
 print(i)
}
```

In class exercise – groups of 4

Write a for loop that uses the coin-toss if-else statement you wrote before and repeats it 20 times, saving the output into a new vector called “flips”.

The output should be a vector of length 20, of class “character”, containing the words “heads” and “tails”

Extra Notes for the Future:

- Make sure to take a look at while statements, and mapping in your post-class readings.
- There are two vectorized if else statements in r. These allow you to effectively loop your if else statements across a vector or column (combining a for loop and an if else statement for you):
  * Base R: ifelse(test, “output if TRUE”, “output if FALSE”)
  * dplyr: if_else(condition, “output if TRUE”, “output if FALSE”).The dplyr version allows you to handle missing values.

```
x=c(-5:5,NA)
ifelse(x<0,“Neg”,“Pos”)
#"Neg" "Neg" "Neg" "Neg" "Neg" "Pos" "Pos" "Pos" "Pos" "Pos" "Pos" NA
if_else(x < 0, “Neg", “Pos", missing = "missing")
#"Neg" "Neg" "Neg" "Neg" "Neg" "Pos" "Pos" "Pos" "Pos" "Pos" "Pos" "Missing"
```

Extra Notes for the Future:

- Make sure to take a look at while statements, and mapping in your post-class readings.
- There are two vectorized if else statements in r. These allow you to effectively loop your if else statements across a vector or column (combining a for loop and an if else statement for you):
    * Base R: ifelse(test, “output if TRUE”, “output if FALSE”)
    * dplyr: if_else(condition, “output if TRUE”, “output if FALSE”).The dplyr version allows you to handle missing values. Often used in combination with mutate.

```
df=data.frame(height=c(30:50))
df |>
  mutate(category=if_else(height < 40, “short”, “tall”))
```

Putting it together; also debugging and validation

Strategies for debugging your code: Documentation matters!

- RTFM: Take the time to read the documentation for other functions your code calls. At a minimum m make sure you understand required arguments (and their order), the type and nature of outputs, and possible warnings or errors that might be generated.
- WTFM: Document your own code! Take the time to write concise, precise descriptions of required inputs, the form and type of outputs, constraints and limitations, etc. Described in sufficient detail what the code and how it's supposed to work. Sometimes writing the documentation *before* you write the code can be really helpful!

Strategies for debugging your code: Tests and intermediate state

- Test your code with known inputs and compare generated outputs to expected results.
- Test your code with incorrect inputs. Does your code fail silently or are there warnings or errors generated?
- Are there corner cases or non-allowable values you need to consider?
- In complex pipelines or flow control statements examine the state of intermediate data objects using strategies such a `print()` statements (simple) or learning to use the debugger in R (complex)