If/else, loops, and functions

Lecture 2

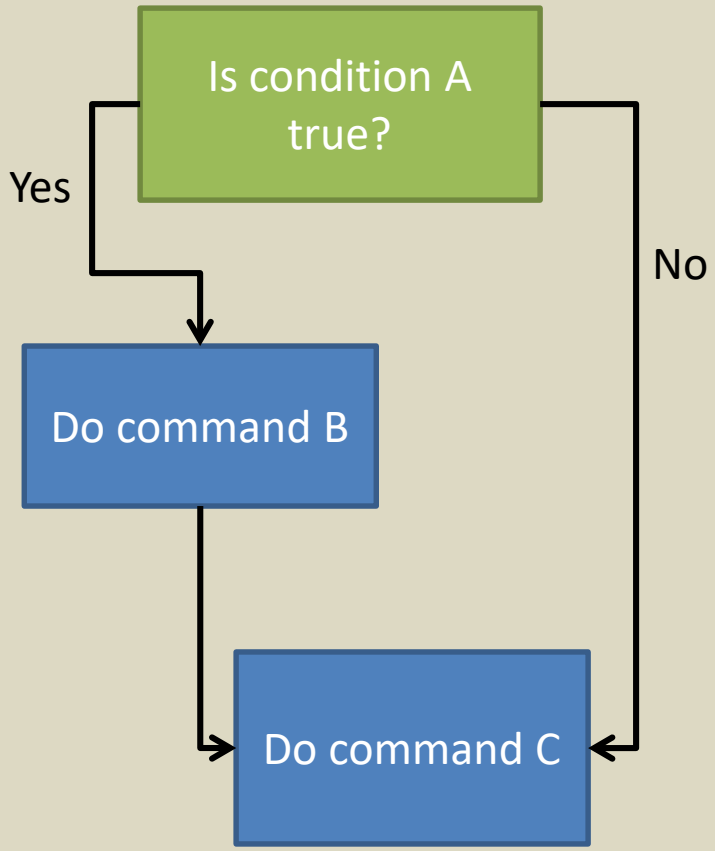
Flow control

- Commands that control the flow of a program – should a certain section of code be skipped, repeated, etc.
- Type `?Control` to get the help page in R

Grouping lines of code

- In R, any time we need to group a set of commands (for example, a section of code that we want to skip under certain circumstances) we use **curly braces** {}
- Indentation within curly braces generally makes code more readable

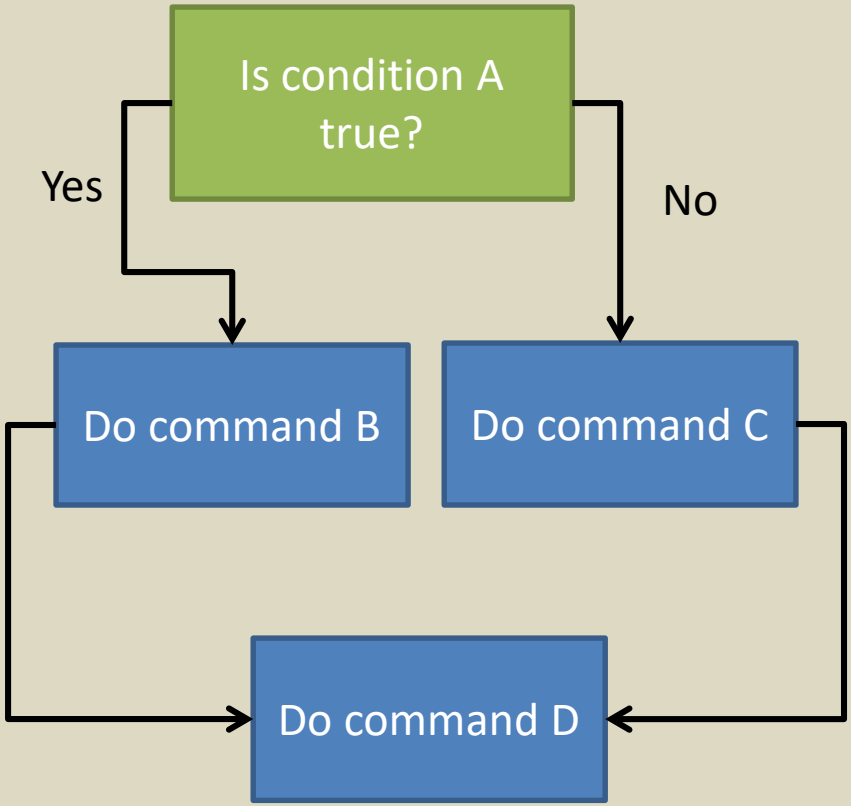
Overview of an `if` statement




```
if(condition A){
   command B
}
command C
```

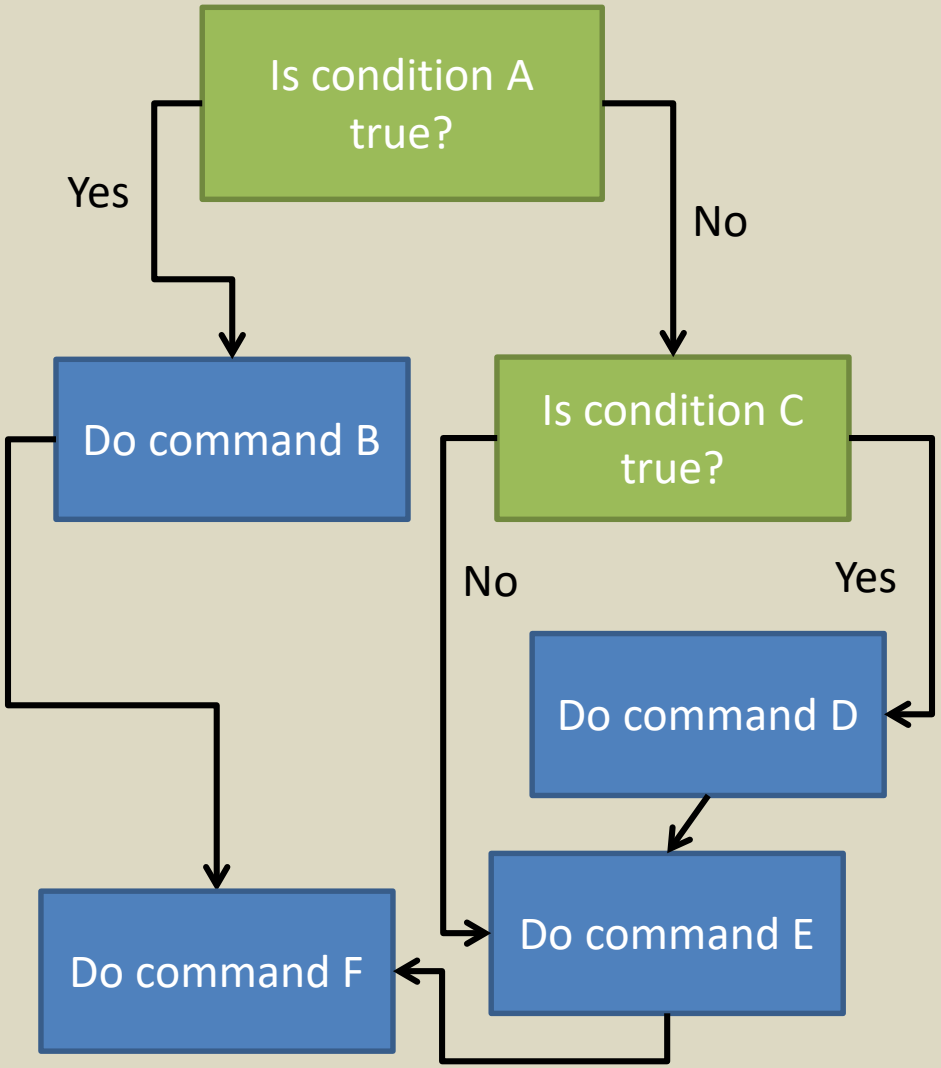
(This is **pseudocode**, a mix of actual code and English that you can use to plan out an algorithm.)

if-else




```
if(condition A){
   command B
} else {
   command C
}
command D
```

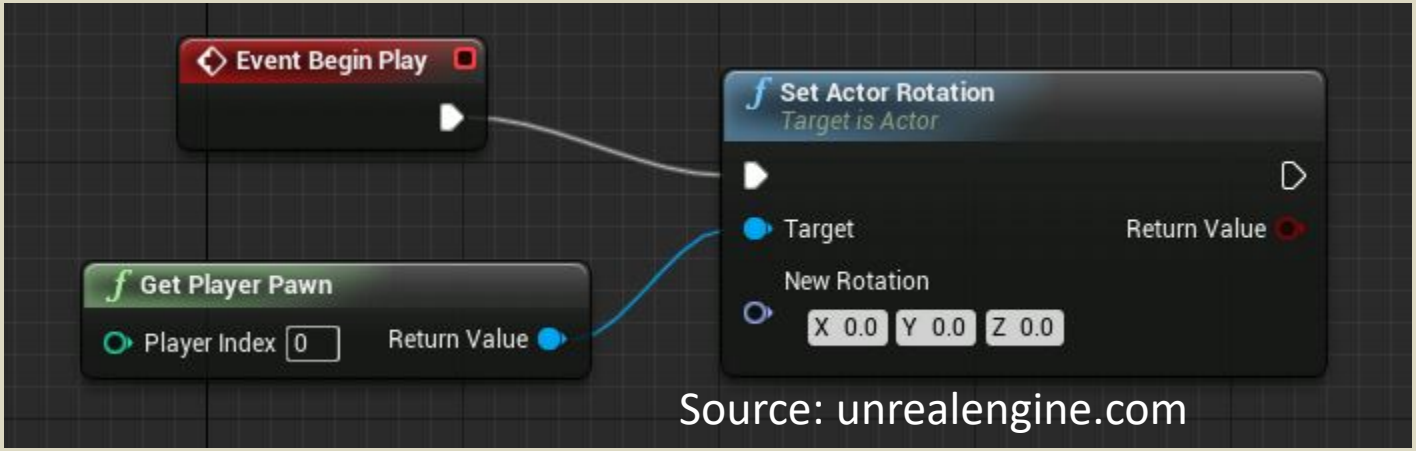
Nested flow control




```
if(condition A){
   command B
} else {
   if(condition C){
      command D
   }
   command E
}
command F
```

Quick side note on the future of coding…

- **Visual scripting** or **visual programming** is where instead of writing code, you draw a flow chart, and a computer program writes the code for you
- Already used in video games, aviation, Galaxy bioinformatics
- Sadly, no replacement for R yet



Source: unrealengine.com

Mini exercise

- Say you want Biomass.yield to always be in g, but sometimes you get a file where it is in kg.
- Add some code to the script that will multiply Biomass.yield by 1000, but only if the median is less than 10.

Using AND and OR with IF statements

- `&` and `|` are the vectorized AND and OR
- In an `if` clause, it is preferred to use `&&` and `||`
- The “doubled” versions only evaluate one element, not a whole vector
- Evaluation stops as soon as a false is found for `&&`, or as soon as a true is found for `||`
- Useful if evaluation of an expression might cause an error

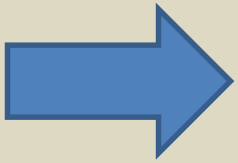
```
TRUE && TRUE = TRUE;   FALSE && never mind… = FALSE
```

Mini-exercise

- Add some code to the script to print a message if plant height OR number of stems is missing

Abstraction

```
Code <- code
Code code code
Code{
  code
}
Code code code
Code code
Code(code){
  code
  code
}
```



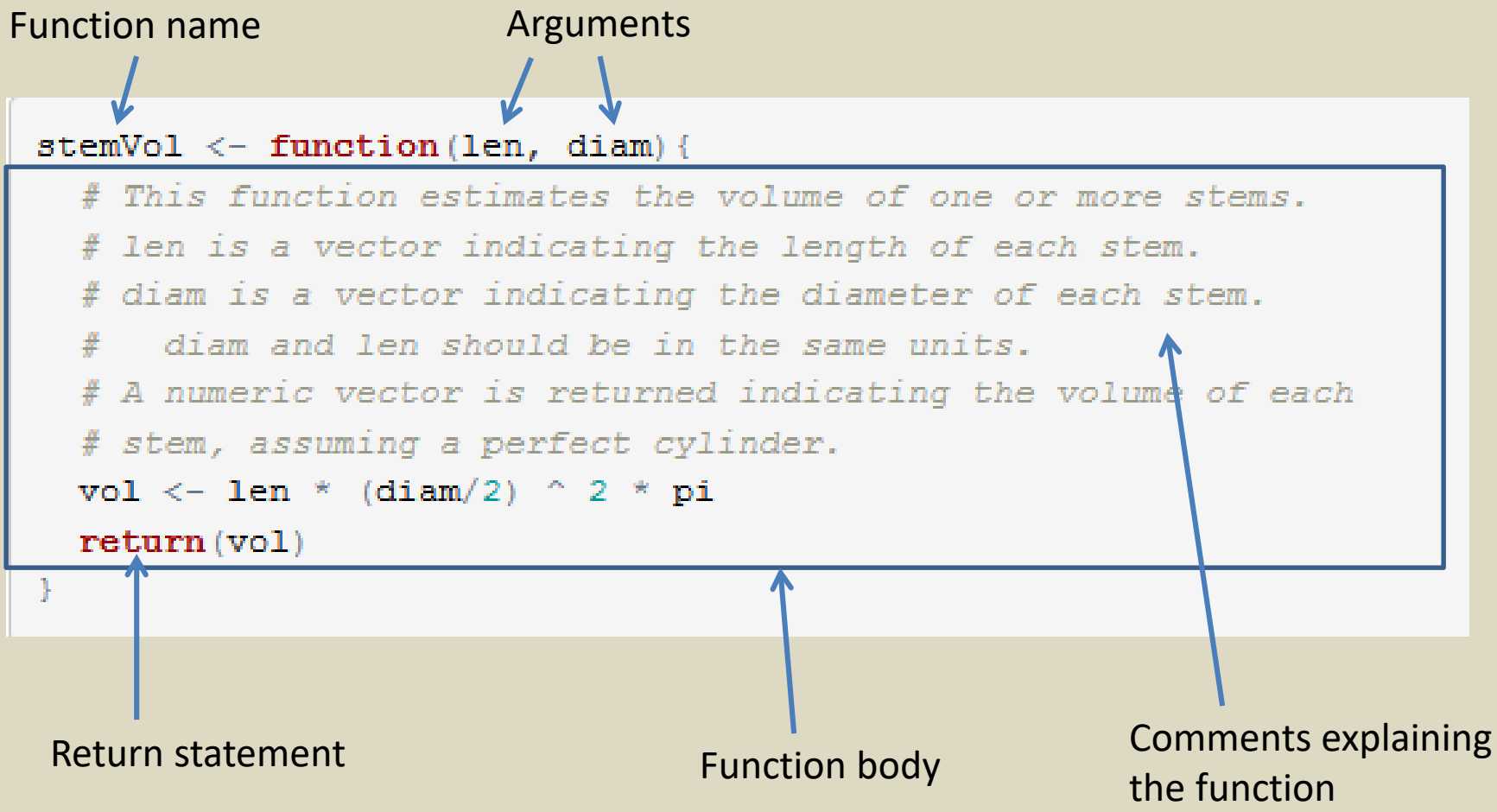
Magic box that just does what we need. (We can open it up again if it behaves unexpectedly.)

Programming is impossible without abstraction. There are already many layers of abstraction between you and what is happening inside your computer. You can make your life easier by creating more layers of abstraction.

Creating your own functions

- Very helpful for **abstraction**: figure out how to do something once, then you don't have to think about it any more
- If you have to fix or change something, you only have to change it one place
- Can add error checking to the function
- Generally makes code easier to read

Anatomy of a function definition



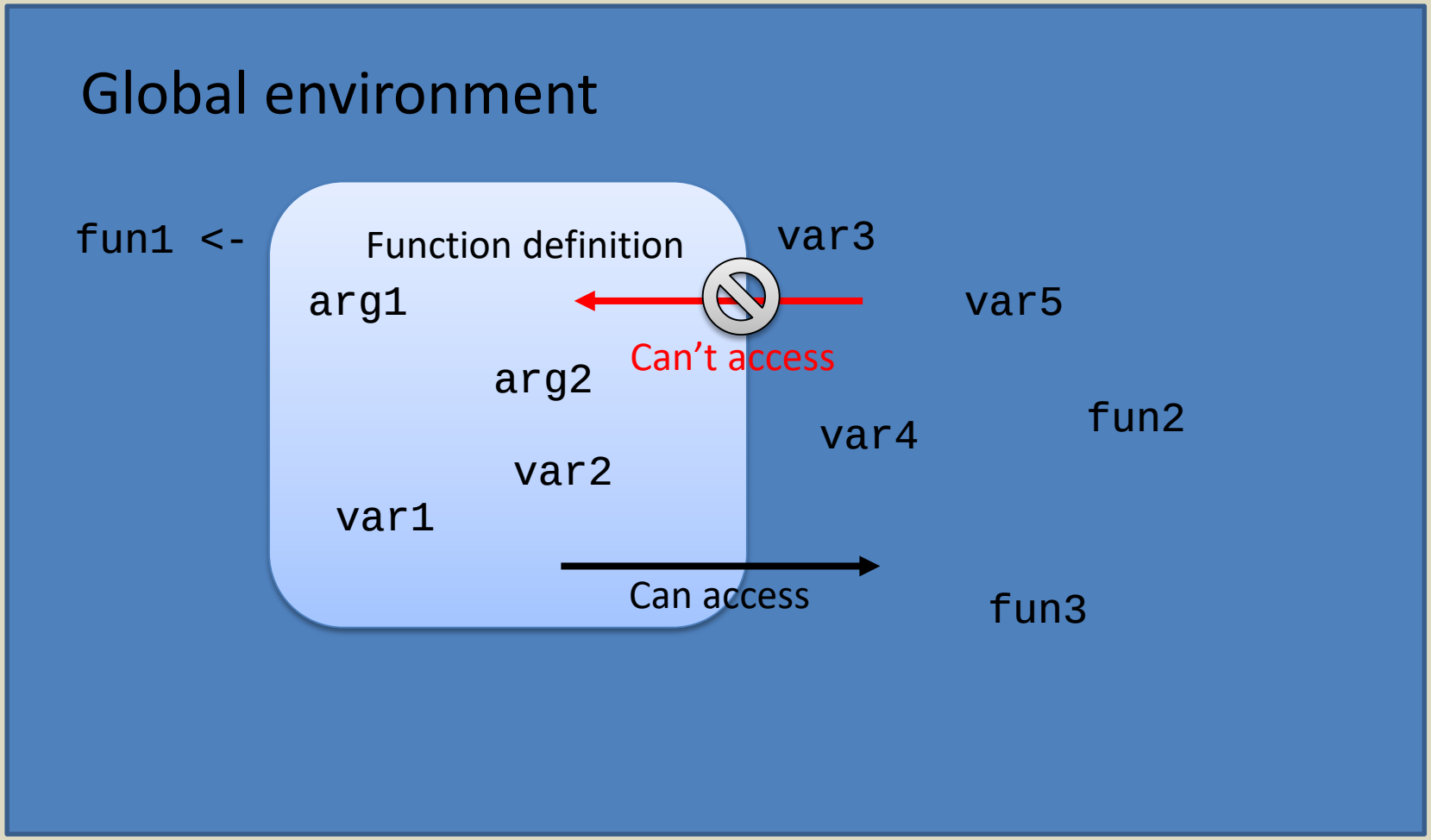
Adding arguments with default values

- Much like it appears in help pages
- `argument = default_value`
- Common to have arguments that are true/false to control behavior of the function (like `na.rm` in `mean`) – pass the argument to an `if` statement in the function body

Mini exercise

- Make a function that converts miles to km (there are 1.61 km in a mile)
- Give it an argument called `nautical` with a default value of `FALSE`
- Make it convert nautical miles to km if `nautical = TRUE` (there are 1.85 km in a nautical mile)

Variable scope



Variable scope

- Argument names become variable names within the body of the function
- Arguments and other variables created in the function body are only accessible within the function body (not within the global environment)
- Helps prevent clutter in the environment

Variable scope cont'd

- If you use a variable in the function body that wasn't defined in the function body, it will look for that variable in the global environment
- Helpful for creating functions that use other functions you have made
- Also dangerous – **we always want a function to return the same result if given the same arguments**

Adding errors and warnings

- `stop` : causes function to **throw** an error (function immediately stops executing)
- `warning` : causes the function to throw a warning (will still finish executing and return the value)

- Both of these are typically put after `if` statements

Adding messages

- The `message` function prints a message in the same category (and color) as errors and warnings
- Helpful for any other info the user might want to know (like progress for functions that take a long time to run)

Mini exercise

- In your function body, use `is.numeric` to check if the distance supplied to your miles-to-km function is a number
- If it isn't give an error (one that is maybe more user-friendly than the standard R error)

Making source code

myscript.R

```
source("myfunctions.R")

x <- 5

y <- plusone(x)
```

myfunctions.R

```
plusone <- function(num){
  return(num + 5)
}
```

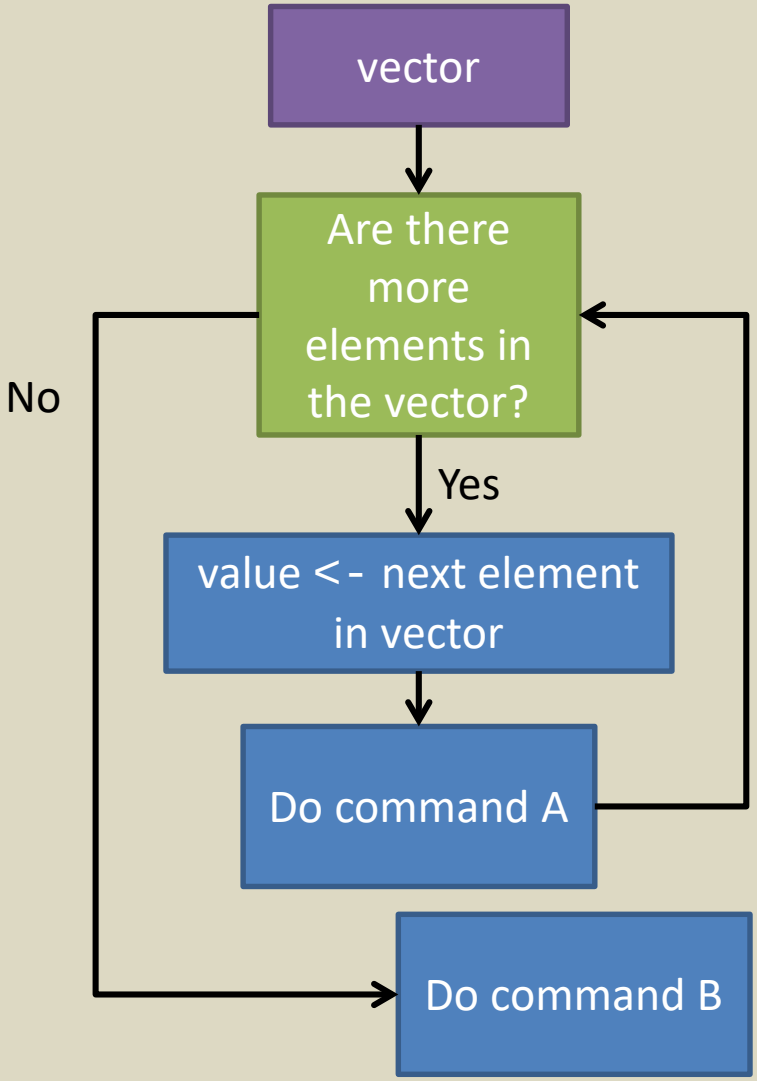
Have both of these files in your RStudio project folder.

(You can also “source” from a URL or somewhere else on your computer)

Making source code

- Generally it is a good idea to make one or more .R files that only contain function definitions
- You can then use the `source` function to load those functions in to your script
- (`source` takes any file of R code and executes the whole thing)
- Call `source` at the top of the script with any `library` calls, to show script dependencies

`for` loops




```
for(value in vector){
   command A
}
command B
```

You can't alter a vector part-way through the loop. You also can't move backwards.

Like if/else, for loops are *flow control*

More on `for` loops

- They are more commonly used in other languages than in R
- In R, vectorized arithmetic and vectorized functions are much faster than loops
- You will still probably use them very frequently

You can loop through any type of vector

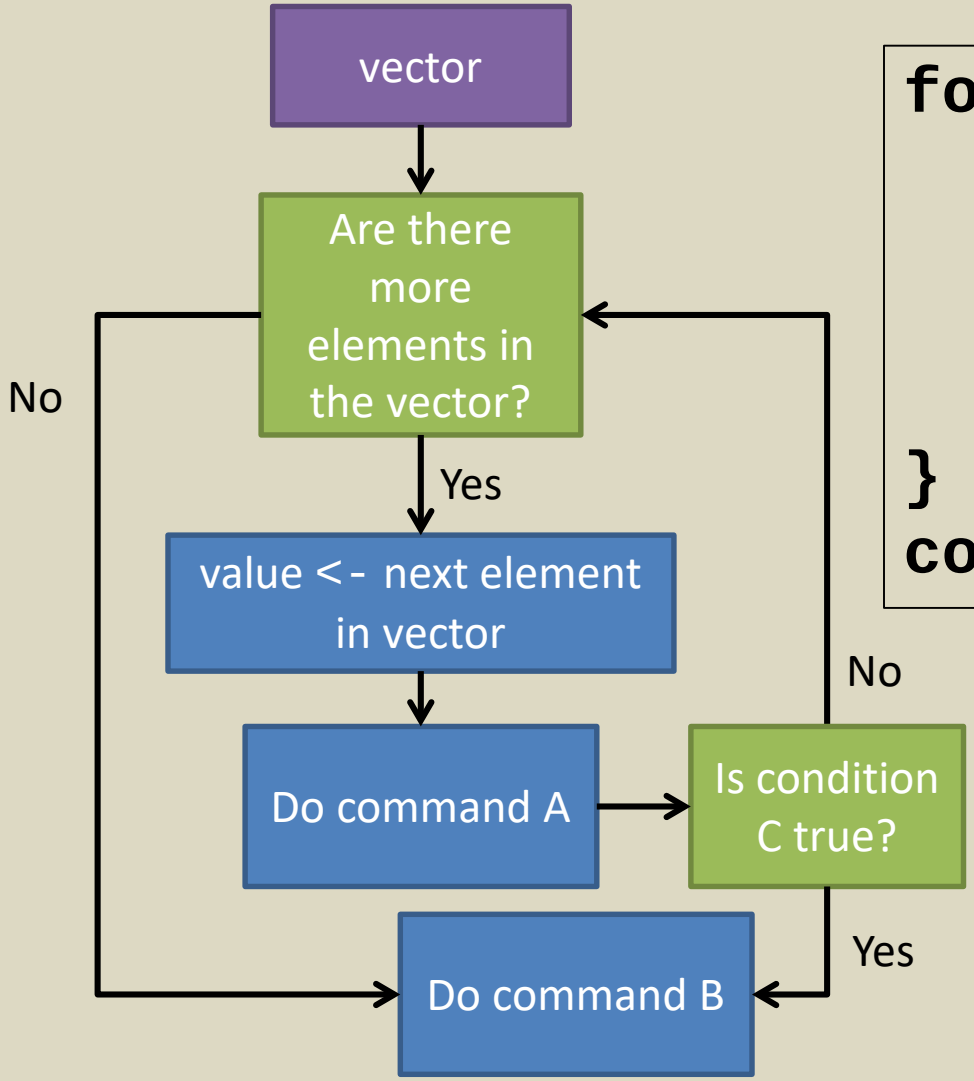
- Atomic vectors
  - Most common: numbers in a series
  - Any vector of numbers
  - Vector of character strings
  - Vector of Booleans
  - etc.

- **Lists**: vectors that can contain any type of object

Constructing a list

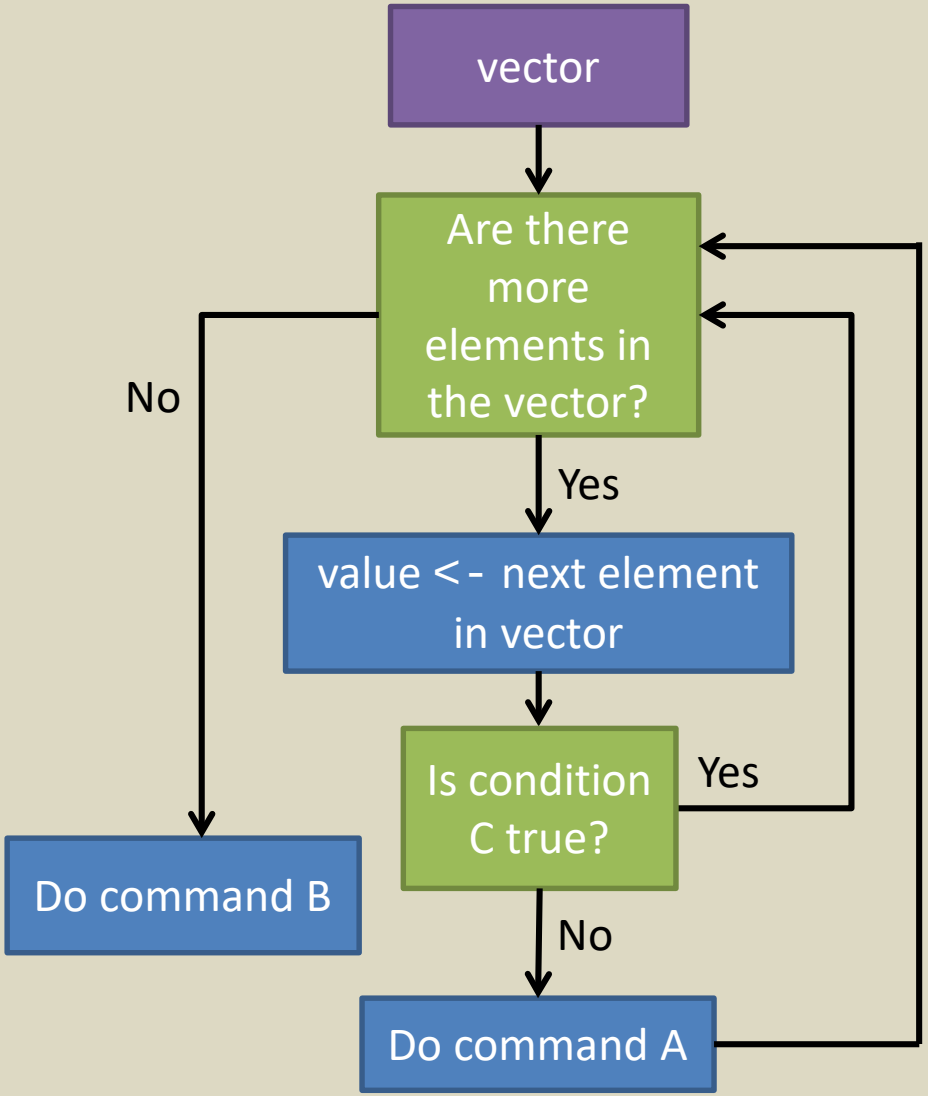
- Instead of `c` for atomic vectors, use `list` for lists
- `mylist <- list(item1, item2, item3)`
- To combine two lists into one list, use `c`
  - This is because `list` will give you a list of lists
- Indexing
  - Use `[]` to get multiple elements in a list
  - Use `[[]]` to get a single element

Interrupting a loop




```
for(value in vector){
   command A
   if(condition C){
      break
   }
}
command B
```

Skipping ahead in a loop

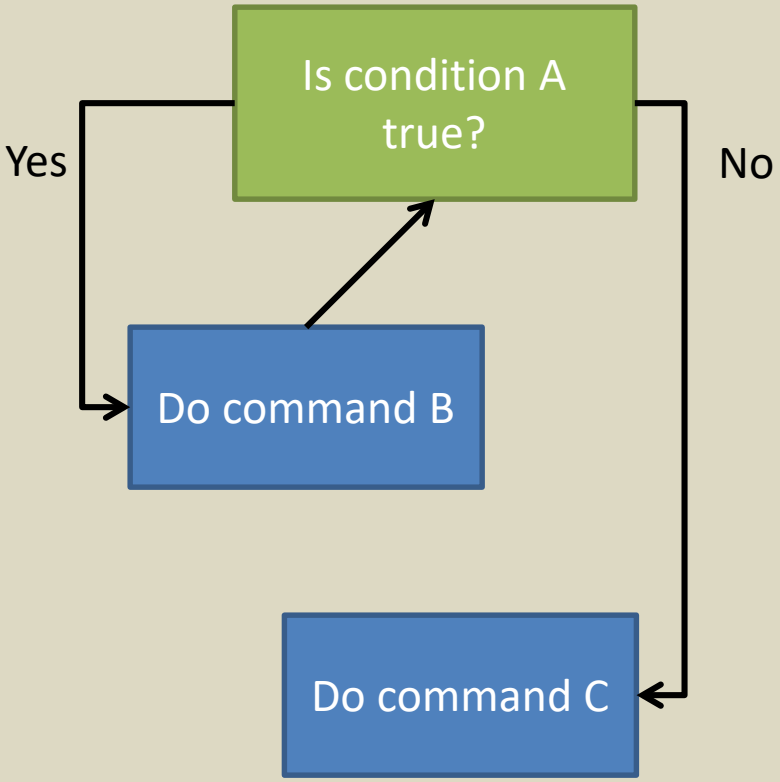



```
for(value in vector){
  if(condition C){
    next
  }
  command A
}
command B
```

Mini exercise

- Use `next` to prevent a data frame from being added to the list if it has fewer than 30 rows

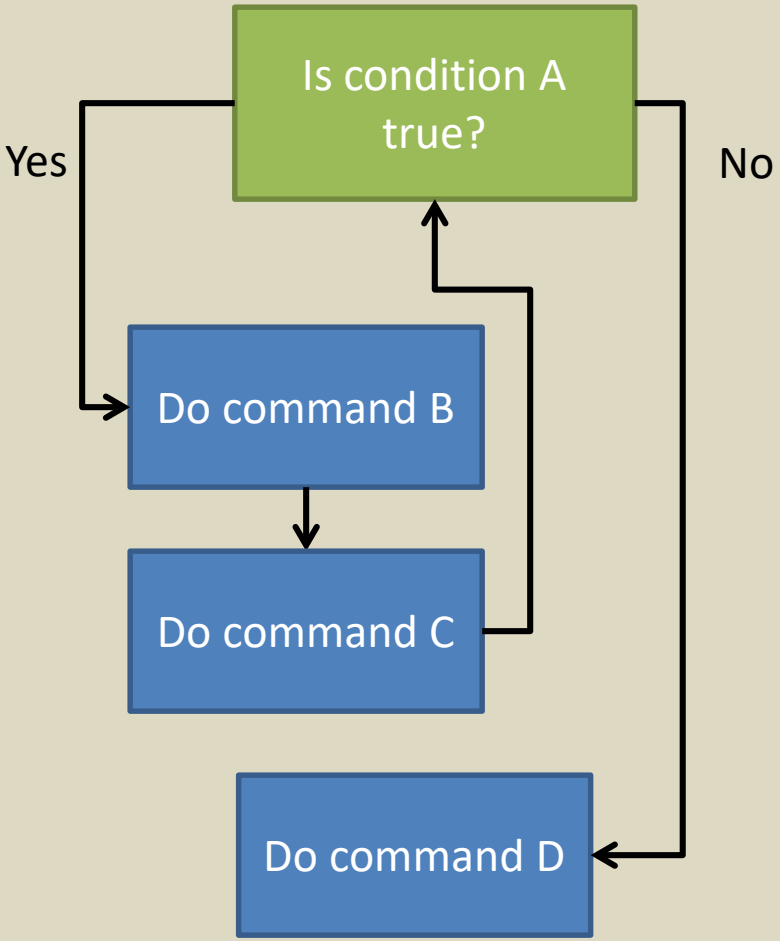
`while` loops




```
while(condition A){
   command B
}
command C
```

Only difference from `if` is that after executing the commands in the curly braces, `while` checks again to see if the condition is still true.

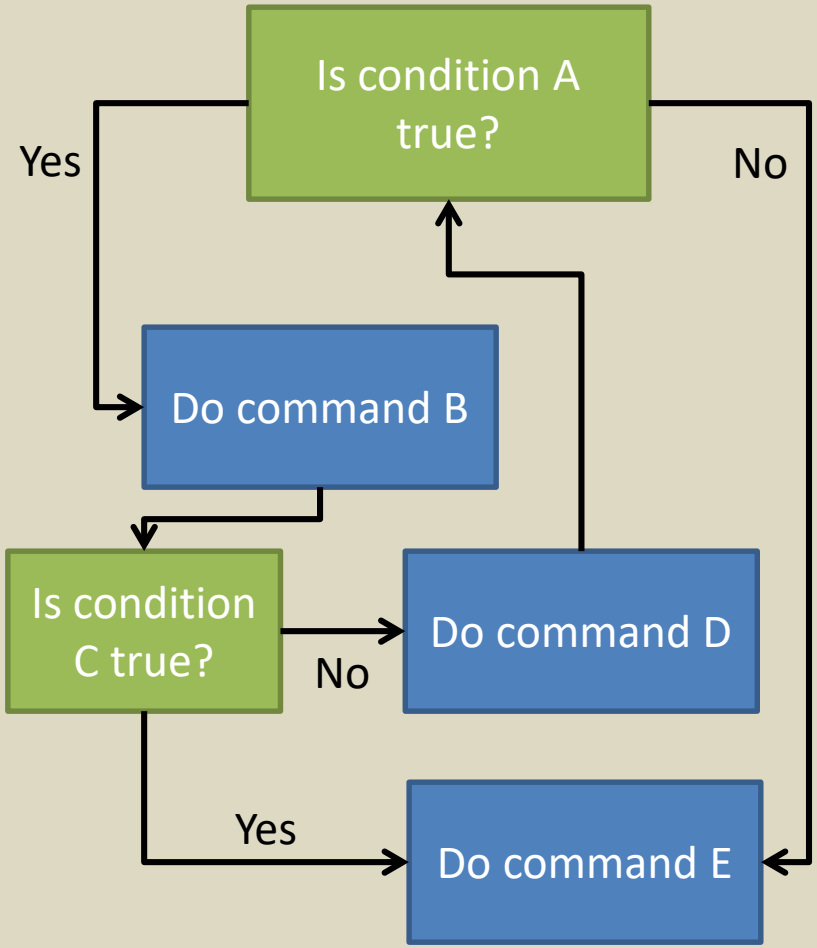
`while` loops




```
while(condition A){
   command B
   command C
}
command D
```

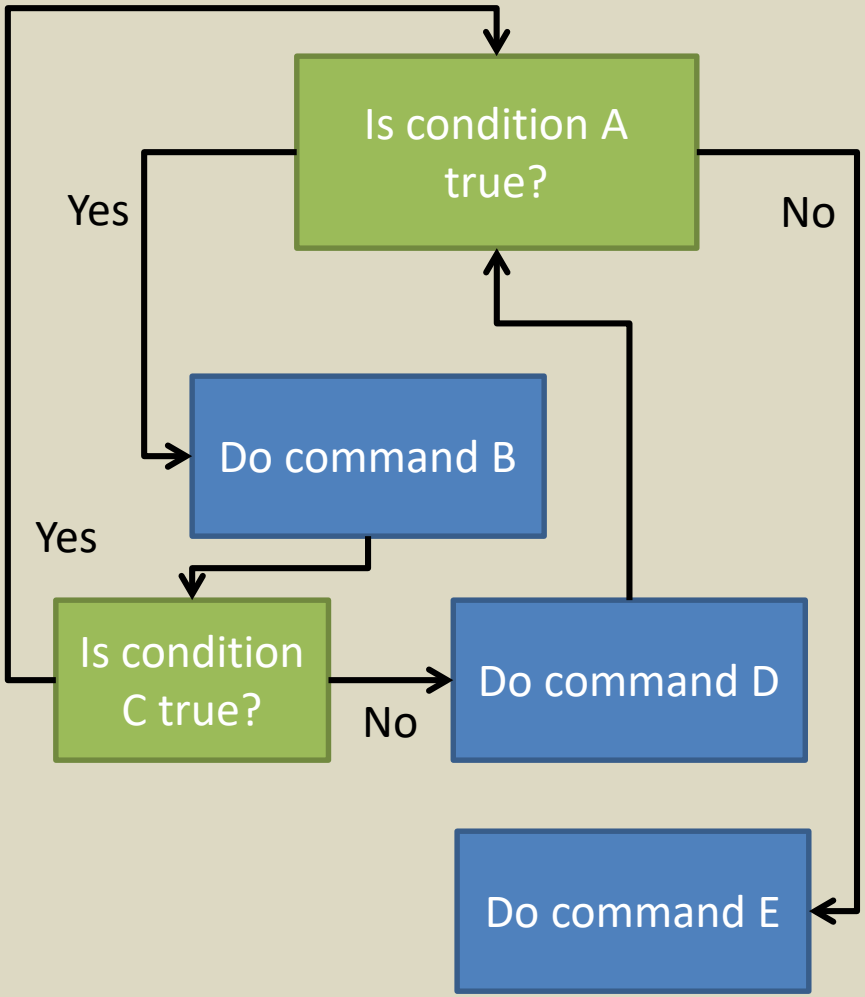
The condition for a while loop isn't checked again until all commands have been executed

Interrupting a `while` loop




```
while(condition A){
   command B
   if(condition C){
      break
   }
   command D
}
command E
```

Skipping ahead in a `while` loop




```
while(condition A){
   command B
   if(condition C){
      next
   }
   command D
}
command E
```

`While` loops in data analysis

- Useful in R if there is a file that is too big to read all at once; more on this in the data import lecture
- Iterative algorithms like estimation-maximization (EM) that keep repeating until convergence is reached
- Not a lot of other routine uses in data analysis