PHW251 Week 2 Reader

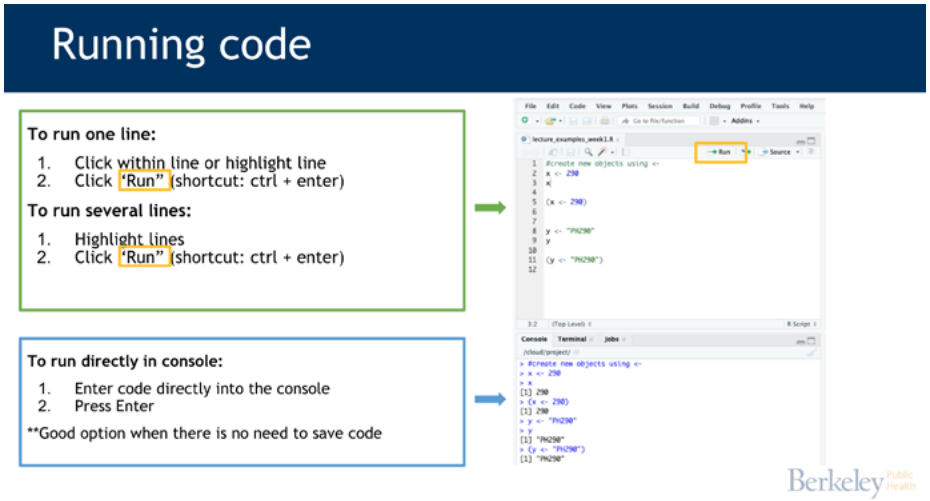
Topic 1: Foundations of Programming

Topic 2: Data Types and Objects

Topic 1: Foundations of Programming

Foundations of Programming

Running Code




```
2 + 3
```

```
[1] 5
```

```
2 + 2
```

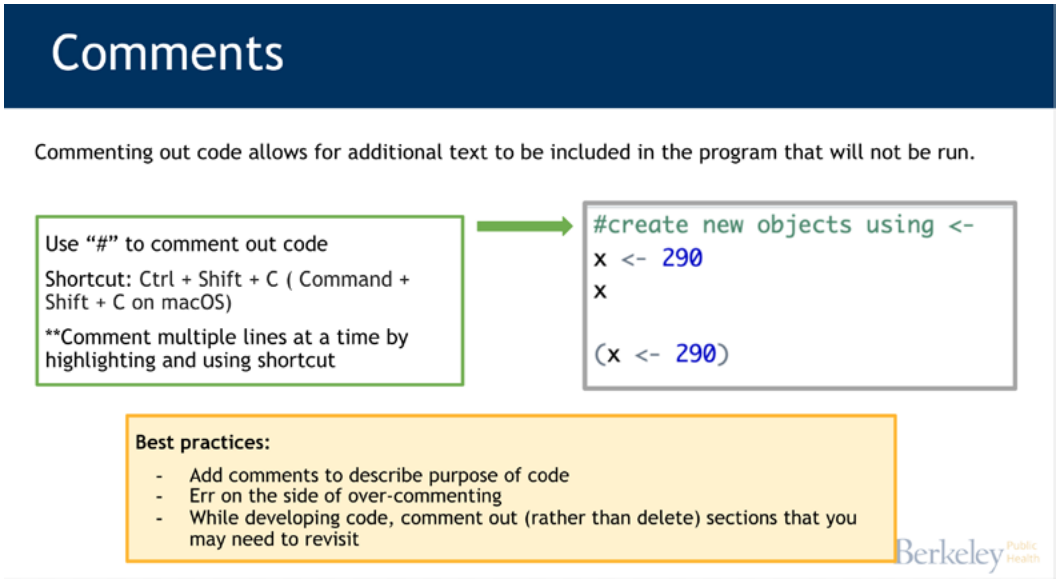
```
[1] 4
```

In this video, we're going to cover a few different topics that we consider to be foundations of programming. We're going to touch on some basics that have likely already been mentioned in previous weeks that are good to just make sure everyone understands how this works including running code, commenting out code, and assigning objects within R. We're also going to talk about some programming concepts. We're going to cover operators, both arithmetic and relational and logical operators. If else, logic and loops, including for and while loops.

The objectives are really to continue our familiarity with writing and running code in R, understand commonly used operators in R, understand logic of an if else statement, and learn how to use loops. Let's start with just a recap of running code within R. There are a handful of different ways that code can be run either one line at a time or several lines at a time. You can also run code within the console, which is down here. Rather than spend a lot of time on this slide, let's just look at this chunk with a couple of different examples.

If we wanted to run this whole chunk because we're working in a QMD could actually just click this Run Current Chunk button. It runs everything that chunk. You can see we get a 5 and 4. You can also run all chunks above this chunk if you need to do in this case, we don't have any chunks above. Don't need to do that. But you can also be just within this chunk. Have your mouse in there and go up to this Run option lists and you can run the current chunk, you can run the next chunk. You can run chunks above or chunks below. You could even restart and run all chunks. Or you can run selected lines. That would work best if you have a line selected, it Run Selected Lines.

There's also easier ways to do that. If I wanted to just run one line, I could use Control Enter to run that. I'm using a Mac and I can also use Command Enter and you could do that one line at a time. Sometimes I do that just to see how each line work. You could even highlight everything and do the same Control Enter to run wherever you can, fit here, 2+3 and get 5. Print it out in our console. If we do that, it's not going to show up within our QMD, but doesn't always need to.




```
#this is a comment

# this is another comment

# this
# is
# several
# comments

# 2 + 4
```

Going on from there, commenting, commenting is super important within code because it allows for additional texts to be included in the program. That text will not be run. We can use the hashtag or pound sign to comment out code. There are also some shortcuts for doing that as well. Here we have a bunch of different comments written. We run this. Nothing. You can see in the console all of those statements have been printed, but this 2+4 is not being evaluated.

There's different ways we can do comments. I think the most efficient ways to use the shortcuts. But you could also just go through and type in the hashtag sign for every line you can do it with or without a space. Or you can select everything you want to comment and do Control Shift C. That will comment it. If you do Control Shift C, again, it will uncomment it. That's really, I think the most efficient way to connect out any code, whether it's a comment describing the purpose of the code, or you have a piece of code that you don't want to run, but you want to keep it in case you want to reference it later.

Object Assignment

To assign an object a value (or values), use "<-":

Object <- value(s)

Assign a value (or values) to an object

Print object contents to console

Assign AND print object contents to console

Best practices:

- R is case sensitive
- Naming objects/variables
  - Use lowercase text
  - Separate words with underscores (my_object_name)

```
#create new objects using <-
x <- 290
x

(x <- 290)


y <- "PH290"
y

(y <- "PH290")
```

Berkeley Public Health

```
x <- 500
x
```

```
[1] 500
```

```
(y<-39+23)
```

```
[1] 62
```

```
current_month <- "August"
```

Object assignment. I believe this has also been mentioned previously, but just to reiterate, there are different ways to create objects. How we recommend doing it is using basically the less than sign and the dash to create that left facing arrow, where on the left-hand side you have your new object name and on the right-hand side you have your values, whether that's a single value or you're creating a table or a vector, that information we'd go on the right-hand side. It will say it's possible to do it the opposite way. But for kind of best practice and cleanliness, we recommend doing it this way.

Let's do a couple of examples. We have a line here where we're just creating a new object, x with a value 500. Let's just run that one line. Once you do that, you can see we have R-value showing up in our environment tab, and it has a value of 500. If we just type out x, we can print that value to our console and below R code chunk in our session here. One way to do both of those things at the same time is to utilize parentheses. So you could do parentheses. We're creating a new object called y, and we're assigning it the value of 39+23. When we run this line, it's actually going to create the y object over to the right and print the value in your console or within your document.

Then we can also create a new object with just a string. If we wanted to say current month is August, we can do that and you can see that show up on the right here as well. You'll see this through all of our examples and you've probably seen them in examples already. But this is how we will create a new object and store information related to that object. Again, that could be a whole table that would probably be more commonly seen as we move throughout the course.

Arithmetic operators

- R can be used as a calculator
- Objects can be created based on calculated values

Example:

```
> 200+90
[1] 290
> x<-290
> x
[1] 290
> y<-200-90
> y
[1] 110
> x*y
[1] 31900
```

Calculation	Operator	Example
Addition	+	200+90
Subtraction	-	200-90
Multiplication	*	2*90
Division	/	90/2
Exponent	^	90^2

...and more!

Berkeley Public Health

```
a <- 200
b <- 90

a/b
```

```
[1] 2.222222
```

```
b^2
```

```
[1] 8100
```

```
new_calc <- a * b
```

Let's move into some of the newer concepts. Arithmetic operators is really meaning like R can be used as a calculator. You can create objects based on calculated values. The operators should look very familiar. There's plus, minus, there's the star or asterisk. multiplication or division, we use a forward slash and then for an exponent we use the little carrot. There's many more operators, these are just some of the most common. I encourage you to look if you're interested.

But one way to start getting familiar with R is to really use it as a calculator, more than just a place to read in a table and you start doing analyses. Here's some examples here we're going to create two new objects, a with a value of 200 and b with a value of 90 and just show that we can do some calculations on those. So we can do a/b, we get 2.22 b to this second, or b^2. We get 8,100. Those were just running their printing in our console. But we could also create a new object called new calc that has a*b as its value. Because we're creating it as a new variable. We don't see it printed, but you can see over here it's 18,000. We could also then our console just type in new underscore calc enter and we get 18,000. We could also within the console do a-b and get 110. If you're wanting to play around with some numbers, you could write out the code for all of this, or you could just use your console to get some practice doing those operations.

Relational & Logical operators

- R can be used to compare values
- Returns a Boolean - TRUE or FALSE

Example:

```
> a<-200
> b<-90
> a==b
[1] FALSE
> a!=b
[1] TRUE
> a<=b
[1] FALSE
> a>=b
[1] TRUE
```

Comparison	Operator
Equal	==
Not equal	!=
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=
Is value in object (vector/list/table)	%in%

Logic	Operator
AND	&
OR	|
NOT	!

Berkeley Public Health

```
a <- 2
b <- 7
x <- 1000
y <- 999

a==b
```

```
[1] FALSE
```

```
x==y
```

```
[1] FALSE
```

```
x<y
```

```
[1] FALSE
```

```
a<b
```

```
[1] TRUE
```

```
x<y & a<b
```

```
[1] FALSE
```

```
x<y | a<b
```

```
[1] TRUE
```

Another type of operator in R relational or logical operators. R can be used to compare values. It will return a Boolean, so meaning TRUE or FALSE. Here's a list of common comparisons we might be interested in. The operator here is actually two equal signs, so ==. Then not equal uses this exclamation point, meaning not, and then just a single equal sign for that not equal. We have less than, greater than, less than or equal to, greater than or equal to, and then another one that might be new is this %in%. This is helpful if we want to see if a value is within an object. If you have a list of counties and you want to say is Contra Costa in that list of counties, you can use this in operator.

Additionally, we have these logic operators that help when we have more complex logic, so if we want to say, are these two values equal and these other two values equal, we would use AND, we could use this to do OR, so are a AND b equal OR c AND d equal, and then NOT. We talked about this with not equal, but basically using that exclamation point will always do the inverse of whatever your other operator is.

Let's look at some examples here. We're creating four new objects here, a, b, x, and y, and you can see they showed up in our environment tab to the right. Let's just run through some of these and see what happens. Here we're comparing is a==b, and we return a FALSE. That makes sense. Is x==y? That also returns a FALSE. We can do x<y, FALSE. There's a trend here, is a<b? There we have a TRUE. All of those are just TRUE or FALSE. Is this TRUE or is it not? Here we get a little more complex, so we're looking for if x<y and a<b, we all expect a TRUE otherwise a FALSE. Then here we have the same but instead of an AND we have an OR, so, if x is less than y OR a is less than b will return a TRUE, so it is true.

```
c <- "hi"
d <- "hello"

c == d
```

```
[1] FALSE
```

```
#using %in%
v <- c("Monday","Tuesday","Wednesday")

"Friday" %in% v
```

```
[1] FALSE
```

```
condition_met <- "Friday" %in% v
```

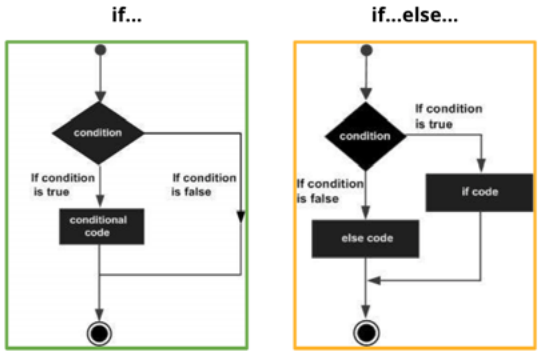
We can do the same thing with strings, so let's create a c and d with hi, hello, saved, so is c==d? No, FALSE. Then introducing the %in% would just see a lot more, but if we have a vector of the values Monday, Tuesday, Wednesday, if we want to know is Friday in that vector, we can say Friday %in% v, and that is FALSE, but if we did Monday, it would be TRUE. You don't have to store this within a new vector value, you can create the vector here as well, so you could do is Friday %in% Monday, Tuesday, Wednesday, and that's false. You can also create a new object with the TRUE or FALSE value, so is Friday %in% v, condition_met, you get a new object, condition_met is FALSE, so let's create that object here.

You might be thinking this is interesting, but I'm not really sure when you use this, but this often comes up in conditional programming, and there are some times where you might want to evaluate some situation, and if it returns a TRUE value, you want to do one set of actions, and if it doesn't, so if it returns a FALSE, then you want to do a different set of actions. One way we use these logical or relational operators a lot is in an if... else statement.

If...else...

If...else...

- Decision making in R
- Based on conditional statement that evaluates to a boolean (T/F) -
  - Include multiple conditions (using & or |)
  - Use ! to get inverse
- Multiple if...else's can be combined for more complex or multi-leveled logic
- if's can be embedded within other if (or else statements)
- ifelse() function - useful tool for data cleaning and creating new fields



Berkeley Public Health

```
##if code structure

# if(condition){
#    if code
# }

##ifelse code structure
```

If...else is really a tool for decision-making in R, it's based on a conditional statement that evaluates to a Boolean, so TRUE or FALSE, so those types of statements that we just discussed above. They can include multiple conditions using AND, or, OR, and then you can also use the exclamation point to get the inverse. You can combine multiple if...else is together for a more complex or multilevel logic. if's can also be embedded within other if statements or within else statements, and then there is an ifelse function that is a useful tool for data cleaning and creating fields. We'll talk about that briefly today, but it will definitely be coming up as we get into data cleaning. Let's look at the if...else's.

There's some code commented out here just to share the structure. If you are just interested in an if statement, so if you're really only evaluating one condition and you only want to do a set of actions if that condition is met, if that condition isn't met, you don't want to do anything, then you can use an if statement, so you would have if and then in parentheses a condition, this condition needs to evaluate to a TRUE or FALSE and then whatever action you want to take within the body of the code chunk, so between these brackets.

```
##ifelse code structure
```

```
# if(condition){
#    if code
# } else if(condition){
#   else code
# } else{
#   else code
# }

day <- "Friday"

if(day=="Sunday"){
  print("Run weekly report")
} else if(day %in% v){
  print("Run SHORT daily report")
} else{
  print("Run FULL daily report")
}
```

```
[1] "Run FULL daily report"
```

Similarly, you could expand this to create some if...else criteria, so this is the same as what we have above, but if this condition is false, then we can add an else and continue evaluating for it to be more complex. We could say if Condition 1, then we run this if code. Otherwise, we keep evaluating and we'll do else if, and then there's another condition, and there's code here, so if Condition 2 is true, this code will evaluate if Condition 2 is false, it will enter the last else and run whatever's in the else code here.

What's important to note is that in these multilevel condition statements, if Condition 1 is true, this code evaluates, it will exit and not run anything else. If there was a situation where Condition 1 and Condition 2 were true, it would not evaluate Condition 2 because it basically exits this if else structure after getting a TRUE or one of the conditions.

Let's do an example. One applied application for this is you're generating some report and on different days of the week you want a different report. We have day. Let's say it's Friday. In this fictitious example, the day is Friday. Then we have these if elses, and they are basically providing different instructions for each day. If the day is Sunday, it's going to print run weekly report. If the day is in v, which is that vector we created earlier that said Monday, Tuesday, Wednesday. We're going to run a short daily report. Otherwise, we'll run the full daily report.

For this example, it may or may not be that you actually want to print run weekly report or or short daily report. In a real-life example, you may replace this with what the weekly report is. If it's Monday to Wednesday, you could put the code here for the short daily report and then otherwise the code for the full daily report. Let's go ahead. We saved our day value and then if we run this, it'll evaluate the whole chunk. We get the run full daily report because it's not Sunday and it's not a day within this vector. We could try changing our day value to Monday. When we run this, we get run short daily report. Another applied example could be something like evaluating a case rate and putting it into a risk category.

```
[1] "Run FULL daily report"
```

```r
#numeric
case_rate <- 45
risk <- "low"

if(case_rate>50){
  risk <- "high"
} else if(case_rate>25){
  risk <- "medium"
} else{
  risk <- "low"
}

#ifelse(condition, if TRUE, if FALSE)
case_rate_gp <- ifelse(case_rate>50,"high","low")
```

Examples of practical applications: + Specifying a chunk of code that should only be run on a certain day (e.g. perhaps you want a weekly summary produced every Friday) + Creating new columns based on conditions (using ifelse() function)

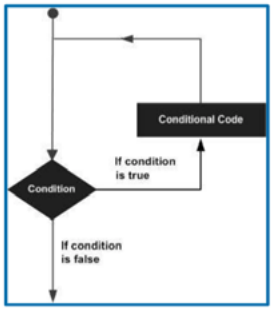
Actually, we don't need this line, but we have a case rate of 45. We're going to compare our case rate to these conditions. If case rate is over 50, our risk is high. Otherwise, if case rate is over 25, it's medium and otherwise it's low. We can print out our risk value, it's medium. We have that in there. We could say, what if it's 10, then our risk value changes to low. Just to introduce this final concept, this is a bit more concise than what we have written above, but there is an if-else function within this R that consolidate that information a bit. It's primarily useful if you're creating a new object or a new column.

But right now, we're just creating a single object called case rate group. We're using this if-else function. The first argument is our conditions, so case rate greater than 50. If that is true, we will assign the value of high to this case rate group object. Otherwise, else, we will have the value low. Let's run that line and print the results. We have low because our case rate value is still 10. If it is 60, we get high.

We will definitely be using the concept of if else throughout the semester. If you have any questions on how this logic works, please make sure to ask. But yes, some practical applications are like the examples we already did. Maybe specifying a chunk of code that should only be run on a certain day. Perhaps you want a weekly summary produce every Friday, or creating new columns based on conditions, and that would be using that last if-else function.

Loops

- Execute a block of code multiple times as long as a condition is met
- Different types of loops that handle conditions differently



For loop executes the same code again and again until a stop condition is met.

- Iterate over items of a sequence
- Best used when number of iterations is known

While loop executes the same code again and again until a stop condition is met. Evaluates the condition at start of the loop.

- Good option when # iterations isn't necessarily known
- Caution: can run in infinite loops

Berkeley Public Health

```
#for
#example
v <- LETTERS[1:4]

for ( i in v) {
   print(i)
}
```

```
[1] "A"
[1] "B"
[1] "C"
[1] "D"
```

The last thing we're going to talk about is loops. Loops are tools for executing a block of code multiple times as long as the condition is met. There are different types of loops that handle conditions differently. We're going to talk about two types of loops. We're going to talk about a for loop which executes the same code again and again until a stop condition is met. It's good for iterating over items in a sequence, and it's best used when the number of iterations is known.

Maybe, you want to run through it 10 times or you want to run through it for every state in the country or something like that. A while loop executes the same code again and again until a stop condition is met. But it evaluates the condition at the start of the loop. It's a good option when the number of iterations isn't necessarily known. But a major caution is that you could end up running infinite loops. That's the danger with the while loop.

Let's look at a couple of examples. I guess, I'll add it. I think we're not going to be talking about loops in depth in this course. But I think just in terms of concepts of programming, it's helpful to have an understanding of what they are doing behind the scenes. Let's do an example with for loops. We're going to create a vector called v that has the letters a, b, c, and d in it. That's what these letters one to four is doing, creating the a, b, c, d. Within our for loop, the structure setup we have for tells R we're using a for loop. We have instructions within the parentheses that are basically telling R for what. I is the placeholder here. We're saying for i in v. It will start with one, that means it will start with the first position of v. Then it will run through and go to the second position in v. It will print the second value. Then it'll go to the third and then to the fourth.

Let's just run that. You can see we have printed out a, b, c, and d. They're not printing out like a vector. They're printing out four separate values. R is doing this really quickly, but you could think about it as a gets printed first, and then b, as it goes through a second time, c the third time, and d, the fourth time.

```r
#example
county <- c("Alameda","Contra Costa","Sacramento")
case_rate <- c(54,23,46)

for (i in 1:length(county)){
   print(paste0(county[i]," case rate: ",case_rate[i]))
}
```

```
[1] "Alameda case rate: 54"
[1] "Contra Costa case rate: 23"
[1] "Sacramento case rate: 46"
```

Another example is maybe we have a vector of counties, Alameda, Contra Costa, Sacramento, and a vector of case rates. Alameda is 54, Contra Costa is 23, and Sacramento is 46. Another way to set up a for loop would be to say for i in one through the length of county. So the length of the county vectors. Let's actually just print what length county is. I guess I haven't created the county after here. Let's try that link. Length county and run that line so it's one through three so that means I is going through each item in, they'll go through three times through the for loop and I'll move through that vector.

We're going to print this paste0 is something I'm not sure you've seen before, but it's basically a way to concatenate information from objects in r with texts so basically we're saying for i, so let's say the first time we go through this loop, i will have a value of 1 and it will print county1, which is a way to index factors. Print Alameda, and they'll print the texts case_rate and then it will actually print the value from the case_rate vector so case_rate1 is 54.

We'll run through this again with i being value of 2 instead of print Contra Costa 23, and then its final time, it will be the value of 3 and it will print for Sacramento in case_rate of 46. Then it stops because it's gone through three times and we told it we want to run one through three. Let's run this, and at the bottom you can see we get our little Alameda case rate 54, etc. That's cool. It's a practical application of this.

```
#while
count <- 1
```

```
while (count < 7) {
   print(paste0("Hello, this is round ",count))
   count <- count + 1
}
```

```
[1] "Hello, this is round 1"
[1] "Hello, this is round 2"
[1] "Hello, this is round 3"
[1] "Hello, this is round 4"
[1] "Hello, this is round 5"
[1] "Hello, this is round 6"
```

Let's talk a little bit about a while loop, and I personally make for loops are more intuitive. But again, there are situations where while loops are helpful. One thing that's usually needed for a while loop is some counter or else you're going to get stuck in an infinite while loop. Let's make an account object with the value of 1, and our while loop, we want it to evaluate as long as count is less than seven, and we're going to print his hello, this is round one. This is round two.

The important part of a while loop is after you do whatever action you are wanting to do, you need to bump up your counter. We're going to overwrite our count field with count plus 1. That means the count value will be 2 and the loop will start again so the count value will be two, two is still less than 7, so it will circle through. We'll go to 3, 4, 5, 6.

Once count value is 6, it will still go through one more time and then it will get to count value of seven, and that's when it will stop because seven is not less than seven. Let's run this while loop, and you can see we have six rounds that we ran through. Hello, this is round 1, 2, 3, 4, 5, 6. If you didn't have this line, the count plus 1, this would run forever because the count value would always be one and never be tuning and so the while loop will just keep running infinitely.

```r
#how they differ
counter <- 1

while(counter <= 10) {
        print(counter)
        counter <- counter + 1
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

```r
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
```

Examples of practical applications:

- Generating a value or report for each county in CA
- Read in multiple files
- Simulations

Let's do one more example. This is a pretty simple example, looks like the one we just did, but showing how these groups could be used to achieve the same output. Really we're just looking for the output 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 so for the while loop, we're going to use the counter value again. This time instead of count, just to be different, they have a while loop that's going to run as long as the counter is less than or equal to 10, we're going to print the counter value, and then add one to our counting. There we go. We've printed 1 through 10.

Then there's two different options for how to do a for loop. We can still use the counter. This line is creating a vector of length 10. Up here, our counter was just one value is 1, and then this counter is a vector from 1-10. Actually, let's see what happens when you run this. Our vector actually wasn't values 1-10, it just has 10 instances of zero. But this is running through the length of counter so it's running through 10 times, so it doesn't really matter what's in this counter vector. It's more about how many elements are in the vector. For I going from one to the length of counter, we can print it here, but should it be 10, and then it's going to print i, so i is going to start at one and then move on to two, all the way to 10 and that's why we get the values printed 1 through 10.

Simpler way to do this without the counter is to just say for i in 1 through 10, that's what the colon here does print i. Then we get 1 through 10 printed as well. Some practical applications of these could be generating a value or a full report or table for each county in California or another state. They could also be used to read in multiple files if you had your files with some naming convention that were either just like day 1, day 2, day 100, or even dates like August, 13th, August 14th, August 15th. You could utilize that to read those in a more efficient way than having to read them in individually, and then they're also good for our simulations.

Again, there may be some problems that questions on for and while loops, they will not be the focus of the majority of the semester, but an important programming concepts so again, feel free to reach out to the teaching team with any clarifying questions on this.

Lecture 2.1: Data Types

Data Types and Objects in R

Overview

- Data types
  - Logical
  - Character/string
  - Numeric
  - Special forms
    - Dates
    - Factors
- Objects
  - Vectors
  - Lists

Objectives

1. Identify different data types in R
2. Understand how knowledge of data types will help you write better R code
3. Understand how to use dates and factors
4. Learn how to use vectors and lists

Hey everyone. In this video we're going to be talking about different data types in R, including logical character or string values, numeric data, and a couple of special forms including dates and factors. Then we're going to dive a little bit into different types of objects. In this video, we'll cover vectors and lists. Additional objects will be covered in other lectures. The objectives are to be able to identify different types of data in R, to understand how knowledge of data types will help you write better R code, to understand how to use dates and factors, and to learn how to use vectors and lists.

Data Types in R

We'll focus on 4 main types:

Data Type	Description
Logical	True / False
Character	Alphabetic, special characters (!@#$%^&*, etc.), designated between " " (single quotes also work, but it's better to leave them for situations when a quoted work exists within a string (e.g. "This is 'useful' information")
Numeric	Whole numbers between -2,147,483,647 and + 2,147,483,647 (designated in code with

Data Type	Description
(Integer)	the letter L)
Numeric (Double)	Whole, decimal, or scientific notation - most numbers in R are converted to double

There are four main types of data in R that we're going to start with. That's logical, meaning, just true or false. Boolean might be another term you'll hear for this data-type. We have character data that's alphabetic special characters typically designated between double quotes. Single quotes do also work, but it's better to leave them for situations when you need to quote something within a string. For example, if you wanted to say, this is useful information as a string but put single quotes around the useful. For this course is our advice to use double-quotes whenever creating strings.

For numeric data, there's two different ways, it's stored in R. There's integer format, which are basically whole numbers from very large to very small and it's designated in the code or in your environment with the letter L, so you would see like a number 7 followed by an L to indicate it's an integer. That might seem like a strange detail, I wouldn't get hung up on that because it doesn't necessarily really matter.

But we also have the double numeric type, which this could be whole decimal or scientific notation. Most numbers in R are converted to a double. But you might notice when you import data or looking at different data elements, some might import as integer and some might import as double and that may just be related to the types of numbers in the field. Doesn't really matter though again.

Logical

To assign an object as a logical value use either `TRUE` or `FALSE` or `T` or `F`.

```
x <- TRUE
x
```

```
[1] TRUE
```

```
y <- T
y
```

```
[1] TRUE
```

`True` and `False` aren't recognized.

```
z <- False
```

```
Error in eval(expr, envir, enclos): object 'False' not found
```

There are four main types of data in R that we're going to start with. That's logical, meaning, just true or false. Boolean might be another term you'll hear for this data-type. We have character data that's alphabetic special characters typically designated between double quotes. Single quotes do also work, but it's better to leave them for situations when you need to quote something within a string.

For example, if you wanted to say, this is useful information as a string but put single quotes around the useful. For this course is our advice to use double-quotes whenever creating strings.

Any expression comparison in R is effectively a function that returns a logical value.

```
7 == 14
```

```
[1] FALSE
```

```
7!= 14
```

```
[1] TRUE
```

```
7L == as.double(7) #integer compared to a double
```

```
[1] TRUE
```

```
0 == 0.00000000000000000000000000001
```

```
[1] FALSE
```

```
"Good" == "Bad"
```

```
[1] FALSE
```

Comparing a vector to another vector (or list to another list) returns a vector of logical values. This kind of evaluation can get very useful later on!

```
c(1, 2, 5, 7, 9) == c(1, 2, 2, 9, 6)
```

```
[1]  TRUE  TRUE FALSE FALSE FALSE
```

Any expression comparison in R is effectively a function that returns a logical value. What these different operators need will be talked about in detail in another video. But basically, if you run each of these lines, it will return a true or false, so this is evaluating is 7=14 and the evaluation is false. This is saying is 7 not equal to 14. That is true. Here is the notation for a integer. We have 7L, and then we are comparing it to the value 7 and we're forcing that to be stored as a double. If we run that, we do get a true.

This is an example of whether or not your data stored as an integer or a double, they will be equivalent if they are whole numbers. For this 7, it doesn't really matter if it's an integer or a double. Can do this comparison, a very small number equal to zero. It is false and then we have these two strings.

This is actually a typo in here. This created a new object called the Good with the value of Bad. You'll notice it's different than what we recommend for doing the kind of left facing arrow as an assignor. You can also use equals, but it does get really confusing so we recommend using the left-handed operator there. We wanted to actually compare if Good equals Bad, we need to make sure we have two equal signs. Let's run that line and then we get a FALSE.

Similarly to comparing single values, we could compare a vector to another. This is an example and we'll do more of these later on, and this might seem like a random example, but it really can get useful later. Now we have two vectors. One vector with the values 1, 2, 5, 7, and 9 and another vector with the value 1, 2, 2, 9, 6. If we run this, we're going to get a vector back of logical values comparing each position in this vector. We have TRUE, these both are the same in both, and then they diverge. They're all FALSE from there or not. We'll actually be talking about more of this towards the end of this video, so this might seem like a brand-new looking thing, but we will talk about it more shortly.

Characters

Character data type (aka strings, text) in R include letters, symbols, and can even be numbers.

To create an object as a character you surround the text in "quotes" as follows:

```
this_char <- "apple"
this_char
```

```
[1] "apple"
```

Can include any characters:

```
another <- "apple123!@#"
another
```

```
[1] "apple123!@#"
```

Can also convert from other data types:

```
numbers <- as.character(1:10)
numbers
```

```
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

Be careful though – if you convert numbers to characters, you'll get odd results when sorted.

```
sort(numbers)
```

```
 [1] "1"  "10" "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"
```

But let's move on to character data. Another word for character data types are strings or text. In R this include letters and symbols. It can even be numbers. To create an object as a character you want to surround the text in quotes. Here's an example or creating a new object called this care with the apple as the string value. We can also create a string that has characters in it. We have our new object called another. Then we can also convert from other datatypes. One to 10 here is going to be a vector of values 1-10. By default, that would be numeric but by using this as character, we can change it to be a character. Let's actually first run this using 1 through 10.

You can see we get a vector over here called numbers, as integer values 1-10. If we replace that with the as.character, it's going to convert all of those values to character. You can see now we have a numbers vector that has the 10 numbers from 1-10. But they're all in quotes, meaning they're being stored as strings.

So there might be times do you want to do this, maybe for year values or something else. But you do want to be careful if you do convert your numbers to characters, you will get odd results when you're sorting. For example, we're using the sort here and we're sorting our numbers vector. You can see that it goes by the first digit, so we get one and then 10, and then 2, 3, 4, et cetera. Could maybe a reason you want to do this but can have some weird behavior, especially if you're sorting or like a final table or final presentation.

String Manipulation

There are lots of ways to manipulate strings and lots of reasons why you might want to. Here's a few examples of key functions. A few of these functions are part of the stringr package, which is in the tidyverse. There's a nice cheatsheet for the stringr functions as well.

Function(x = string)	Package	Description	Example
`substr(x, start, end)`	Base R	Pull out a key suffix, prefix, or something in between	`substr("sem1_grade", 1, 4) == "sem1"`

`gsub(pattern, replacement, x)`	Base R	Replace some portion of the string	`gsub("sem1_", "", "sem1_grade") == "grade"`
			
			`gsub("-", "_", "field-name") == "field_name"`

Let's talk a little bit about string manipulation. We will talk about this in depth in a video later on in the semester. But just for some brief orientation, there are a lot of ways to manipulate strings and there are also a lot of reasons why you might want to do that. Maybe you're doing basic cleaning. Maybe you're trying to extract some information from a string. One of the helpful packages for manipulating strings is the string R package, which is part of the Tidyverse.

There's a nice cheat sheet for that which we recommend checking out. But here are some both functions from base R and in this string R package that are helpful for manipulating strings in different ways. Again, we'll talk about this a lot more in depth later on when we get to the data cleaning portion of the semester. But there is a function substring or sub str where you specify a string and the start position and end position to pull out pieces of information.

There's an example here. If you use substring on the string, sem1_grade and you want to extract from one position 1-4, you'll end up with a string of just sem1.

Gsub is a great tool for replacing a portion of a string. In this example, we are looking for a pattern. The pattern we're looking for is sem1_. We're specifying a replacement which in this case is nothing. It's just the double quotes with nothing in between. Then x here would be the string that we're evaluating. We're evaluating the sem1_grade and this would return just the value of grade. It's replacing that sem1_ with nothing.

Another common use for this might be if you want to change a dash to an underscore. In this case we're looking for the dash will replace with an underscore, and the field we're looking at is this field name with the dash in between. What you'd get out of here is the fields_name.

`str_to_upper(x)`
`str_to_lower(x)`
`str_to_title(x)`
	stringr	Convert to all lower, all UPPER, or Title Case. Really helpful when trying to join datasets by strings	`str_to_upper("try_this") == "TRY_THIS"`
			
			`str_to_lower("tbl_YELLING") == "tbl_yelling"`
			
			`str_to_title("ALAMEDA CO") == "Alameda Co"`

`str_detect(x, pattern)`	stringr	Returns a logical depending on whether the pattern exists in the string	`clothes <- c("tie", "shirt", "dress")`
			
			`str_detect(clothes, "e") == TRUE FALSE TRUE`

A set of functions within string or that are super helpful are these str to upper, to lower, to title. This will convert all text to lower, upper, title case. It's really helpful when trying to join datasets. Here we have an example of string to upper. This string is all lowercase but when you enclose it in the string to upper, you get all uppercase. We can do the same string to lower. Then for title case as well. Here's Alameda County in all caps. If you enclose it in the string to title, you'll get Alameda CO with just the capitalization for the first letters.

In the last time we'll mention now is string detect. It will return a logical depending on whether the pattern exists in the string. Here's an example where we're creating a vector of three strings, tie, shirt, and dress. If we do string detect on clothes, we're looking for E. It will return a vector with the values of true for each of these strings that have E and M, and false if it doesn't have yes or true, false, true.

```
        library(stringr)

        substr("sem1_grade", 1, 4)
```

```
[1] "sem1"
```

```
        gsub("sem1_", "", "sem1_grade")
```

```
[1] "grade"
```

```
        gsub("-", "_", "field-name")
```

```
[1] "field_name"
```

```
        str_to_upper("try_this")
```

```
[1] "TRY_THIS"
```

```
        str_to_lower("tbl_YELLING")
```

```
[1] "tbl_yelling"
```

```
        str_to_title("ALAMEDA CO")
```

```
[1] "Alameda Co"
```

```
        clothes <- c("tie", "shirt", "dress")

        str_detect(clothes, "e")
```

```
[1]  TRUE FALSE  TRUE
```

This code chunk actually has all of those examples we just looked at above. I'm not going to go through them again, but feel free to access them here and play around with them a bit.

Numeric

Computers can store and retrieve numbers more efficiently than we can. There are two main types of numbers in R. For the most part, the differences don't matter a whole lot as R will convert types to match what is needed.

However, it's still good to know the difference.

Integer - Take up less memory and can sometimes be preferred for indexing and iterating through a sequence.

Double - Are best for calculations and displaying results.

Special Forms

Dates

Dates are a special form of numeric that are stored as numeric but display in formats we're familiar with. Similar to how Microsoft (1/1/1990) and SAS (1/1/1960) anchor their dates, R anchors to a specific date (1/1/1970).

- Dates are stored as the number of days before or after 1/1/1970
- Date/times are stored as the number of seconds before or after 1/1/1970

The default display format in R is `yyyy-mm-dd` (e.g. `2020-09-17`).

Let's walk through some examples of dates, starting with a base R function `Sys.Date()` that returns the current date and `Sys.time()` which returns the current date AND time.

```
Sys.Date()
```

```
[1] "2024-06-21"
```

```
Sys.time()
```

```
[1] "2024-06-21 10:13:25 PDT"
```

Let's talk briefly about numeric values. Computers can store and retrieve numbers more efficiently than we can. As I mentioned earlier, there are two main types of numbers in R. For the most part, the differences really don't matter a whole lot because R will do some conversion on the backend to make sure types match under what is needed. But it is good to know the difference and you will see the difference pop up when you're doing exploration of your datasets and things like that.

We have the integer values, they take up less memory and can sometimes be preferred for indexing and iterating through a sequence. Double values are best for calculations and displaying results. They will have the decimal places and whatever else is needed, scientific notation, et cetera. You can be much more specific with a double. But there may be times that an integer is sufficient, maybe it's just group numbers or years or row numbers or something like that.

But again, it doesn't really matter. The memory issues are really beyond the scope of this course to discuss but you have both of those options. Let's dive into special forms a bit. Dates are a special form of numeric data that are stored on the numeric but displayed in formats that we're more familiar with. Similar to how Microsoft and SaaS anchor their dates are also anchors, their date values with specific date there's is going off of January 1st 1970. Dates are stored as the number of days before or after January 1st 1970, and then date times are stored as the number of seconds before or after that date.

The default display format in R is this four-digit year dash two-digit month, and two-digit date, like something like this. Let's walk through some examples. This is another topic that we'll cover more in depth in a later video but for now, want to make sure everyone feels comfortable with dates because I think especially in public health data we're using dates all the time so let's look at some examples.

There are a couple of functions within base R the sys. date and sys.time which will return the current date and time so let's just run both of those. I'm recording this on August 14th, 2023, you can even see what time I'm recording it and it even has my time zone. There are ways to change the time zone and things like that but we're not going to focus on that now.

We're going to use the tidyverse's lubridate package to work with dates. There's a huge selection of useful functions. We'll cover a few!

```
library(lubridate)
```

We can read in different types of date formats using any variation of **y**ear, **m**onth, **d**ay functions: `ymd()`, `ydm()`, `mdy()` ...you get the idea! These functions will return the standard `yyyy-mm-dd` format by parsing the strings you provide.

```
ymd("1865/06/19")
```

```
[1] "1865-06-19"
```

```
mdy("July 20, 1969")
```

```
[1] "1969-07-20"
```

```
dmy("8/03/1917")
```

```
[1] "1917-03-08"
```

With R, we can easily manipulate time.

```
# 90 days from today
Sys.Date() + 90
```

```
[1] "2024-09-19"
```

```
# 100 days ago
Sys.Date() - 100
```

```
[1] "2024-03-13"
```

```
# 12 miliseconds into the future
Sys.time() + 12 / 1000
```

```
[1] "2024-06-21 10:13:25 PDT"
```

Another package that we will again talk about more later but part of the tidyverse is the Lubricate package and there's a huge selection of useful functions. I encourage everyone to check out the links here but we're going to cover a few right now that are especially helpful for getting started.

Let's load the Lubricate library and some of the assumptions are equivalents in base R, but it's just honestly preferable to work within the Lubricate package. So there's functions that can help with reading in different types of date formats using any variation of year, month, and day functions. So you can see here they're in different orders, this is year, month day, this is year, day, month, month, day, year. You get the idea and these functions will return the standard format by parsing the strings provided.

Here's a few examples and this is common you might be reading in data from different sources but store the data in different formats. Here we're reading in a month, year, month day, month day year written out, and then another day month year. This is pretty flexible in terms of whether it needs to be in the typical format with the front slashes or written out as text. Let's turn these three and you can see they all return a date value that's aligned with ours standard format.

Another thing we can do in R is manipulate time. We can add or subtract from the current day or from any days. Here's a few examples. We can add 90 days to today, we can subtract under days from today and then using this time, we can go 12 milliseconds in the future if we really want to, so there's plenty of opportunities there but that's pretty specific. You don't always know how many days or milliseconds you want to move, but there are some additional functions that are helpful with that.

But what if we want to modify based on a window of time, such as a day, month, or year? Not every month has the same number of days and writing out `365` each time for a year can get messy. Using lubridate functions we can more clearly calculate these time frames.

```
        # 3 days later
        Sys.Date() + days(3)
```

```
[1] "2024-06-24"
```

```
        # 5 months before
        Sys.Date() - months(5)
```

```
[1] "2024-01-21"
```

```
        # 2 years later
        Sys.Date() + years(2)
```

```
[1] "2026-06-21"
```

We can also extract date info easily too.

```
        # grab the day from a date
        day(Sys.Date())
```

```
[1] 21
```

```
        wday(Sys.Date(), label = TRUE, abbr = TRUE)
```

```
[1] Fri
Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

```
        # grab the month from a date
        month(Sys.Date())
```

```
[1] 6
```

```
        # grab the year from a date
        year(Sys.Date())
```

```
[1] 2024
```

Super useful for future epidemiologists is determining the epiweek!

```
        epiweek(Sys.Date())
```

```
[1] 25
```

We can modify based on a window of time such as day, month, or a year, not every month has the same number of days and writing out 365 each time for each year would get messy.

Using Lubridate functions, we can more clearly calculate those timeframe. If we wanted to add three days to this day we can do that. That's very similar to what we did above but where it gets more novel is that we could subtract five months or we could add two years.

There's a lot of options there and going off of that, we can use similar functions to extract date info as well. Using today's date again, we can pull out the day, we get 14, we can pull out the weekday. Today is Monday this is handy for different purposes and pull out the month eight and then the year 2023.

One tool that's super helpful for future epidemiologist is determining the epi weeks. There is an epi week function, if you run that, we are currently in the epi week 33.

Dates are very useful tools to help organize data.

```
        df <- data.frame(date = c("1917-03-08", "1969-07-20", "1865-06-19"))

        # max date
        max(df$date)
```

```
[1] "1969-07-20"
```

```
        # min date
        min(df$date)
```

```
[1] "1865-06-19"
```

```
        # ordering by dates
        sort(df$date)
```

```
[1] "1865-06-19" "1917-03-08" "1969-07-20"
```

The last thing on dates before we move on is there are useful tools for organizing date data. Here we're creating a little data-frame, we'll talk about data-frames more in depth later but it has three date yields and we could use some of the summary statistics to pull out some information. We can get a max date, we can get a minimum date, we can order by dates. There's just different options for using the date data to our advantage for organizing our data in general.

Factors

Factors are another special form of numeric that are stored as numeric but display as text strings. In the past, factors efficiently stored character strings as integers, saving memory. Recent versions of R automatically convert character data to factors behind the scenes when needed, allowing us to not worry about memory.

Factors can be useful and even necessary in specific situations:

- Certain analytic packages require factors
- Labeling axes in visualizations

For the most part, the benefits outweigh the drawbacks of factors. For example, if you think you're operating with a character field but it's actually a factor, you might have difficulty debugging.

```
x <- factor(c("apple", "orange", "apricot", "apple"),
            levels = c("apple", "orange", "apricot"))
x
```

```
[1] apple   orange  apricot apple
Levels: apple orange apricot
```

```
as.numeric(x)
```

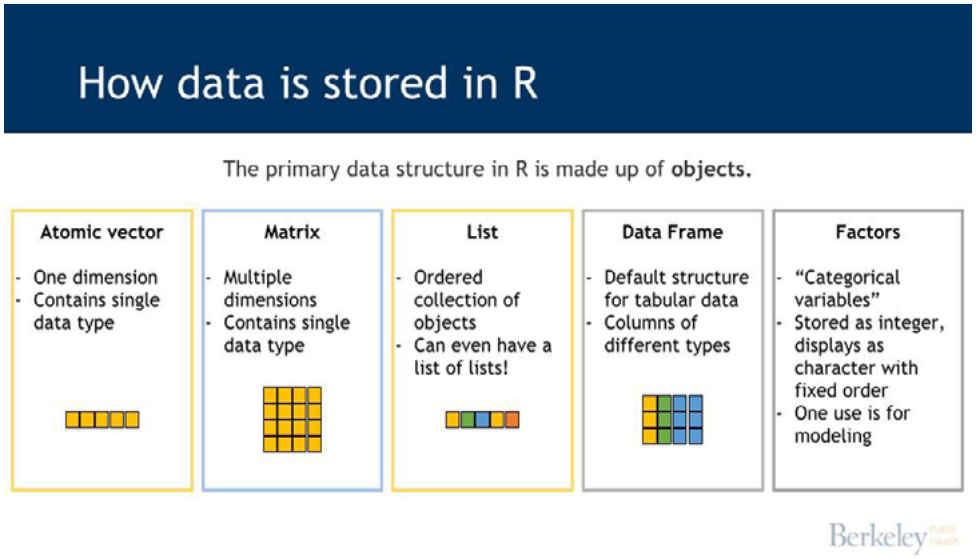
```
[1] 1 2 3 1
```

The other special form of data we want to talk about are factors. They are another type of numeric data but they are stored as numeric but displayed as text strings. In the past factors efficiently stored characters strings as integers, saving memory but recent versions of R automatically convert character data to factors behind the scenes when needed, allowing us not really to worry about memory anymore but there's a couple of times that factors are still useful. Primarily using certain analytic packages that require factors is one use case and another that may be more common to this course is labeling your axes and visualizations, or maybe wanting to control the order of categories in a table.

The strange thing about factors is that it's just good to know you're working with a factor or not, if you think you're operating with a character field but it's actually a factor, you might have a challenge debugging and vice versa. We're just going to look at one factor example and not spend too much time on this, but we're creating a new object called X and it is going to be a factor. It's a vector of four variables and then we're specifying the levels is apple, orange, and apricot.

R will take these levels in the order you give them an apple will be Level 1, orange two, apricot three. Let's go ahead and run this and then we'll print out X and we see our vector here and our levels are specified below. If we do as numeric to X, if we convert from factor to numeric, we'll actually show us the values as numeric, aligning with the levels that we specified, say 1, 2, 3 and then one again because apple is one.

Lecture 2.2: Objects



Let's move on to talk about a couple of different types of objects. The primary data structure in R is made up of objects. We have vectors, matrices, lists, DataFrames and factors. We already just talked about factors. We'll talk about DataFrames in the coming week, but today we're going to talk about atomic vectors and lists.

Vectors

Vectors are one dimension and a single data type. We can create vectors with the `c()` function.

```
c(2, 3, 4, 5, 6, 45, 2)
```

```
[1]  2  3  4  5  6 45  2
```

Or create a range of numeric values using `start:end`.

```
1:10
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

Or a sequence with `seq()`.

```
# every 2 numbers up to 10
seq(2, 10, 2)
```

```
[1]  2  4  6  8 10
```

We can also put string's in vectors, but if we combine numeric and strings together the numeric values are converted into strings. This occurs because R requires vectors to be of one type.

```
c(1, 2, 3, "I am a string")
```

```
[1] "1"           "2"           "3"           "I am a string"
```

Vectors are one dimension and a single data type. We can create vectors with the C function. You've already seen us do this in previous examples in this video and other videos. We'll go ahead and run this. This is just creating a vector. We're not assigning it a name or anything so we're not going to see it in our environment panel, but we can see that we have a vector stored with these seven values.

We can create a numeric vector using a start and end with a colon in the middle. This will create a vector of numeric values from 1-10. We can also use the sequence function. This tells us we want every two numbers up to 10. We get 2, 4, 6, 8, 10. Actually this is saying start with two, we want to go to 10 and then we want every two numbers. We could also do something like five to 50 by 10. That would give us 5, 15, 25, 35, 45.

We can put strings and vectors, but if we can bind numeric and strings together, the numeric values are converted into strings. This occurs because R requires vectors to be a one data type. You can see we have our 1, 2, and 3 all as strings.

Using vectors, we can create columns in data frames.

```
testing_pos <- data.frame(
  county = c("Alameda", "Contra Costa", "Marin"),
  total_tests = c(500, 745, 832),
  pos_tests = c(43, 32, 30)
)
```

We also have the ability to compare vectors to each other to find matches using the `%in%` operator.

```
# two list of counties
v_county <- testing_pos$county
county_list <- c("San Diego", "Marin")

# let's see how many counties in v_county are in county_list
v_county %in% county_list
```

```
[1] FALSE FALSE  TRUE
```

Vector also can be named.

```
named_v <- c("day1" = "Monday", "day2" = "Tuesday", "day3" = "Wednesday" )
named_v
```

```
      day1       day2       day3
  "Monday"   "Tuesday" "Wednesday"
```

One use for vectors is to create columns in DataFrames. Here we have a DataFrame that we're creating. We're creating it with three columns and we're using vectors to specify the values for each column. We can check out that testing pause, so you can see the values from our vectors were transported into this DataFrame. We'll talk more about DataFrames soon, and then, we also have the ability to compare vectors to each other to find matches using the in operator.

This is something that might seem random but will come in handy. If we have two lists of counties, so v_county, we're just extracting the county values from our testing pause DataFrame. You can see them show up over here. We have Alameda, Contra Costa, and Marin. We have another county list of San Diego and Marin. If we say v_county in lists, we will get a vector that's checking for every value in the county, whether or not it's in the county lists. Alameda is not in county lists. Contra Costa is not in county list, but Marin is, so we get it true. This will come in handy, I promise.

Vectors can also be named. We are creating a named vector here and printing it. We can see our first element is named Day 1. The value is Monday, Day 2, Tuesday, Day 3, Wednesday.

Indexing vectors

There are a few ways to index a vector:

- integers (positive or negative)
- logical vectors
- character vectors / names
- `which()`

Integers

```
vec <- seq(0, 50, by = 5)
vec
```

```
 [1]  0  5 10 15 20 25 30 35 40 45 50
```

```
# grab the 5th element using [brackets]
vec[5]
```

```
[1] 20
```

```
# elements 3, 6, 9
vec[c(3, 6, 9)]
```

```
[1] 10 25 40
```

```
# all elements except 3, 6, 9
vec[c(-3, -6, -9)]
```

```
[1]  0  5 15 20 30 35 45 50
```

```
# all elements except 5, 6, 7
vec[-5:-7]
```

```
[1]  0  5 10 15 35 40 45 50
```

There are a few ways to index a vector. You can utilize integers to specify the position in a vector. You can use another logical vector, character vectors or the which function.

Let's start with integers. Here we're creating a new vector that has values from 0-50 by five. If we want to grab the fifth element, we can use a single bracket and specify five. Let's do that, and the fifth element is 20. Then if we wanted to specify elements 3, 6, and 9, we can use a vector, the vector with a unit vector, but we use this vector within the brackets to say places 3, 6, and 9. That's values, 10, 25, and 40.

If we do something similar, we can say all elements except 3, 6, and 9 by using the minus sign in front of each. This is the original vector minus those three elements. Then we could also, if we wanted to drop elements that were next to each other, we could use this start and end. We're going to drop 5, 6, and 7. Let's just run that whole chunk so you can see everything at once. Different ways to index a vector.

Logical vectors

```
vec <- 5:10
vec
```

```
[1]  5  6  7  8  9 10
```

```
# keep only TRUE indexes
vec[c(FALSE, TRUE, FALSE, TRUE, FALSE, FALSE)]
```

```
[1] 6 8
```

```
# by condition
vec[vec > 7]
```

```
[1]  8  9 10
```

Character vectors / names

```
named_v <- c("day1" = "Monday", "day2" = "Tuesday", "day3" = "Wednesday" )
named_v
```

```
      day1       day2       day3
  "Monday"   "Tuesday" "Wednesday"
```

```
# see names of elements
names(named_v)
```

```
[1] "day1" "day2" "day3"
```

```
# index by name
named_v["day1"]
```

```
   day1
"Monday"
```

```
# index by multiple names
named_v[c("day1", "day2")]
```

```
    day1      day2
 "Monday" "Tuesday"
```

You can index the vector by using a logical vector. Here we're just creating another simple vector with values 5-10. Then if we wanted to, we could specify which ones we wanted to keep by providing a vector of the same length with a false and true for each element. If we do that, we just keep six and eight, and that makes sense. This is 5, 6, 7, 8, 9, 10. Or we can use a condition, so where the vector is greater than seven and we want to keep the values so we get 8, 9, and 10. I'll run all three of those so you can see them together here.

We have character vectors, gears are named vector again. Day 1, Day 2, Day 3, and that's what this looks like as a reminder. We could pull out just the names of the vector fields and that's Day 1, Day 2, Day 3 using this name's function. Because this is a named vector, we can index by name. Here we have named underscore v. Then we're just indexing the first value that's name Day 1. Then you can do something similar if you wanted to keep two of them, you could supply it a vector of the names that you want to keep. Keeping Day 1 and Day 2 returns just Monday and Tuesday.

which()

```
vec <- 95:110
vec
```

```
[1]  95  96  97  98  99 100 101 102 103 104 105 106 107 108 109 110
```

```
# find position of values that match the condition
which(vec > 99)
```

```
[1]  6  7  8  9 10 11 12 13 14 15 16
```

```
# return the elements that match the indexes
vec[which(vec > 99)]
```

```
[1] 100 101 102 103 104 105 106 107 108 109 110
```

You can also use the which function. We have another new vector with values 95-110. Print that out. You can find the position of values that match the condition using which so if we said which then vector greater than 99, we get return of the positions in the vector. This is not the actual values in the vector is the positions, but we could use those positions than to tell R which elements that we want to keep from the vector.

If we ran this line, it would just keep those that were greater than 99. If we ran all three of these, you can see we have our full vector. We have the positions that are greater than 99, starting in 6, 16 and then what are vector would look like if we use that which vec is greater than 99 to specify what we want to keep.

Modifying vectors

We have a few ways to modify vectors too:

- Change element
- Delete
- Combine vectors
- Calculate

```
# create multiple vectors to use below
v1 <- c(2,7,3,21,98)
v2 <- 1:5
v3 <- c("a","b","c","d","e")
v4 <- c(1,3)
```

Change element

```
# change an element
v1[5] <- 94
v1
```

```
[1]  2  7  3 21 94
```

```
# add an element
v1[6] <- NA
v1
```

```
[1]  2  7  3 21 94 NA
```

We can modify vectors. You can change an element, you can delete, you can combine vectors, you can calculate on vectors. There's a lot you can do. We're going to create four new vectors here. I'm going to scroll so you can see them quickly in my environment tab, there are a variety of different types of vectors.

If we wanted to change an element, we could reference for v1, and we want to change position five to 94, we can do that using the single bracket. If we print that you'll see the last value is changed to 94. We can add a sixth element. In this case we're adding an NA element. We print that, then we now have six elements. The last one is NA.

Delete

```
# delete an element
v1[-6]
```

```
[1]  2  7  3 21 94
```

```
# or
v1[1:5]
```

```
[1]  2  7  3 21 94
```

```
# clear vector values
# create vector to delete
test <- v1

# clear values from vector
test <- NULL
```

Combine vectors

```
# combine vectors
# same type
v1_2 <- c(v1,v2)
v1_2
```

```
 [1]  2  7  3 21 94 NA  1  2  3  4  5
```

```
# numeric and character, combine to character
v1_3 <- c(v1,v3)
v1_3
```

```
 [1] "2"  "7"  "3"  "21" "94" NA   "a"  "b"  "c"  "d"  "e"
```

You can similarly delete an element, so we can delete the one we just added. We can say v1, put the minus six and the bracket and we can also say which ones we want to keep, which is similar to what we just did above. We want to keep elements 1-5.

We can also create a new vector based on a previous vector. We have a new vector called test with values from v1. Then we can clear that vector if we wanted to by just saying NULL, overriding tests with NULL. You can see that changed in the environment.

You can combine vectors. Here we're going to combine what we have v1 and v2 into a new vector called v1_2. You can see that they just add to each other, start with all the values from v1 and then add the values from v2. Then if we wanted to combine a numeric and character, you would get all character values. You can see the NA isn't within quotes. That's because it doesn't really matter whether the NA is numeric or character, it's just NA.

Calculate

```
# adding vectors of different lengths will work - the short vector values will be
# align with length of longer vector
v1+v4
```

```
[1]  3 10  4 24 95 NA
```

```
# subtract
v1-v2
```

```
Warning in v1 - v2: longer object length is not a multiple of shorter object
length
```

```
[1]  1  5  0 17 89 NA
```

```
# multiply
v1*v2
```

```
Warning in v1 * v2: longer object length is not a multiple of shorter object
length
```

```
[1]   2  14   9  84 470  NA
```

```
# absolute value
v_abs <- c(-1,-4,3,7)
abs(v_abs)
```

```
[1] 1 4 3 7
```

We can do calculations on our vectors. It typically is best practice to use vectors that have the same length, but you can also operate on vectors that have different lengths. We have v1+v4. That's this first vector with six elements and before has two elements. If we do that, we get a new vector without six data elements.

Let's print all three of these. This is v one. It's adding one to two, three to seven, and then it starts over. It goes one to three, three to 21, one to 94, and three plus NA, which gives you NA.

You can also subtract, it will be similar. You'll see we're getting this warning about them being different lengths. This really isn't helpful. You can see that we have our six elements and it's doing something similar as the vectors before where it's cycling through, since the vectors are of different sizes. These are closer. So each is paired up through five and then when it hits the NA, it is trying to subtract one again. So it's cycling back through, but again that gives us a NA. We could multiply those two together. Similarly, it cycles through.

Then one calculation that can be helpful is doing the absolute value which we have. We're going to create a new vector called the abs with some negative and positive values in it, and then use the abs function to generate the absolute value. So a lot you can do with vectors. They will definitely come up through different aspects of the course.

Lists

List are ordered collections of objects

- One object
- Multiple objects
- Or even a list within a list (nested lists)

Let's create our first list.

```
list1 <- list("dog",375)
list1
```

```
[[1]]
[1] "dog"

[[2]]
[1] 375
```

We can go further by naming the list.

```
named_list <- list(pet="dog",number=375)
named_list
```

```
$pet
[1] "dog"

$number
[1] 375
```

Let's close this out talking about lists. Lists are kind of similar to vectors but more complicated because they can be an ordered collection of objects. It can be one object, multiple objects, can be a list within a list, and it doesn't have the same requirement of being only one data type. Let's create a basic list that has one string and one number. List 1 and you can see these show up near the data frames rather than the values. Portion of our environment tab and you can see that we have a character value of dog and numeric value 375 and if you click on this, it opens up something that looks like this, which is very different than the data frames.

But you can see that we're able to have different types for our different list elements. We can name the list. We can say our first element is called pet and the value dog. Then we have number is 375, so the named_list here and see them. These double bracket positions have been replaced by the name. So if you are using a list, it is helpful to name it because it is easier than to reference as you'll see a little bit later.

Complex lists

With lists, we can mix up data types: data frames, character vectors, matrix, another list.

```
testing_pos <- data.frame(
  county = c("Alameda", "Contra Costa", "Marin"),
  total_tests = c(500, 745, 832),
  pos_tests = c(43, 32, 30)
)

named_v <- c("day1" = "Monday", "day2" = "Tuesday", "day3" = "Wednesday" )

list1 <- list("dog",375)

# data frame, named vector, list
big_list <- list(testing_pos, named_v, list1)
str(big_list)
```

```
List of 3
 $ :'data.frame':   3 obs. of  3 variables:
  ..$ county     : chr [1:3] "Alameda" "Contra Costa" "Marin"
  ..$ total_tests: num [1:3] 500 745 832
  ..$ pos_tests  : num [1:3] 43 32 30
 $ : Named chr [1:3] "Monday" "Tuesday" "Wednesday"
  ..- attr(*, "names")= chr [1:3] "day1" "day2" "day3"
 $ :List of 2
  ..$ : chr "dog"
  ..$ : num 375
```

```
# character vector, matrix, list
multi_list <- list(
  "Weekend" = c("Sat","Sun"),
  matrix(c(3,9,6,12,-3,21), nrow = 2),
  list("dog",375))

str(multi_list)
```

```
List of 3
 $ Weekend: chr [1:2] "Sat" "Sun"
 $        : num [1:2, 1:3] 3 9 6 12 -3 21
 $        :List of 2
  ..$ : chr "dog"
  ..$ : num 375
```

With lists, we can mix up data types, we can store data frames, character vectors, matrices and other list. There are several items being created here. We have a data frame that we used above. We have our named vector. We have the list we just created, and we can actually create a big list which is a list that includes the data frame, a vector, and a list.

We can look at the structure of that. You can see we have all the information. We have data frame with the column information, we have our vector, and then we have another list. We can also open that up over here and because it's a little bit more dynamic, you can expand these to get more information about what's in each of the objects.

Let's do one more example of a more complicated list. We have a new object here called multi_list that has a vector, a matrix, and a list. We're naming this part of the list Weekend. The other parts don't have names. Let's go ahead and run multi_list and then look at the structure. You can see we have our vector, our matrix, and a list of two elements, dog and 375.

Indexing lists

Indexing lists is similar to indexing vectors.

```
named_list <- list(pet="dog", number=375)

# index list
named_list$pet
```

```
[1] "dog"
```

```
# compare single brackets
named_list[2]
```

```
$number
[1] 375
```

```
# to double brackets
named_list[[2]]
```

```
[1] 375
```

```
# can also return just names
names(named_list[1])
```

```
[1] "pet"
```

There's different ways to index list. It's similar to indexing vectors, but let's just start with a more basic example. We have our named_list, pet, and number. We can index this using the name of the list element. So using named_list$pet will return dog.

We can use single brackets. We do named_list 2 will return everything within this second element. So that's why we see the name and the value. If we do double brackets, we'll actually just extract the value of 375. Note it's kind of confusing, but single brackets is extracting this whole element and double brackets is kind of drilling down to the 375.

You could just return the names of the list if you wanted to. So it means the names function named_list the first element would return at. I think that's more straightforward.

```
        # indexing in list of list
        multi_list[1]
```

```
$Weekend
[1] "Sat" "Sun"
```

```
        multi_list["Weekend"]
```

```
$Weekend
[1] "Sat" "Sun"
```

```
        multi_list$Weekend
```

```
[1] "Sat" "Sun"
```

```
        multi_list[2]
```

```
[[1]]
     [,1] [,2] [,3]
[1,]    3    6   -3
[2,]    9   12   21
```

```
        multi_list[3]
```

```
[[1]]
[[1]][[1]]
[1] "dog"

[[1]][[2]]
[1] 375
```

```
        multi_list[3][[1]][2]
```

```
[[1]]
[1] 375
```

When we get into these more complicated lists like multi_list up here, it gets a little bit hard to track, to be honest. But we can index using the position or the name, if there is one. So for the first element here Weekend with the values Sat and Sun, we can use multi_list 1 and single bracket and we'll get that whole element. We can also do something similar using the name as a string and we'll get the exact same output.

If we use multi_list$Weekend, we'll actually just return the values within that vector and not the name. We can also extract multi_list 2 and multi_list 3. You can see here this list that's within a list, it's getting a little bit complicated. There's a lot of brackets here that are a little bit hard to navigate.

So if we wanted to extract this dog value, we're within the third element of our list, we're within this first part of the list. So that's where this double brackets with a one and then we want to extract the first element there. If we run this, we get dog. I believe if we change this to two, we get 375 since that's really the second element of this list.

So it's confusing, I will say that for sure. I think if you're in a position where you need to extract from a list, I would just make sure you know exactly what you're wanting to extract and play around with using single or double brackets.

Modifying lists

We can also modify list, very similar to how we modify vectors.

```
        # replace element
        multi_list[3][[1]][1] <- "cat"
        multi_list[3][[1]][1]
```

```
[[1]]
[1] "cat"
```

```
        multi_list[3]
```

```
[[1]]
[[1]][[1]]
[1] "cat"

[[1]][[2]]
[1] 375
```

```
        # add new element
        multi_list[3][[1]][3] <- "mouse"
        multi_list[3][[1]]
```

```
[[1]]
[1] "cat"

[[2]]
[1] 375

[[3]]
[1] "mouse"
```

```
        multi_list[4] <- "new"
        multi_list
```

```
$Weekend
[1] "Sat" "Sun"

[[2]]
     [,1] [,2] [,3]
[1,]    3    6   -3
[2,]    9   12   21

[[3]]
[[3]][[1]]
[1] "cat"

[[3]][[2]]
[1] 375

[[3]][[3]]
[1] "mouse"


[[4]]
[1] "new"
```

```
        # remove the last element.
        multi_list[4] <- NULL
```

We can modify lists. It's similar to how we modify vectors, but we run into the same kind of complications we just saw above. If we want to replace an element, we can use the position of it to replace the values.

In this case, we'll be replacing dog with cat and we'll just print the third list to prove that we changed dog to cat. If we want to add a new element, I think we need to add a three here. So when we had the two, that would reference the 375. If we add three, we have cat, 375, and mouse all within that third list element, which was that embedded list.

Then we can add a whole other element to our multi_list. We'll just add a string called new and then if we print multi_list, you can see we have our Weekend here, we have our matrix, we have our updated third element, and then we have a fourth element just called new. If you want to remove the last element, you can use null and let's put that again and it's gone away.

If you need to convert a list to a vector, you can use `unlist()`.

```
list_a <- list(2,4,6,8)
list_b <- list(1,2,3,4)

vec_a <- unlist(list_a)
vec_b <- unlist(list_b)

vec_a + vec_b
```

```
[1]  3  6  9 12
```

The last thing we'll talk about is if you need to convert a list to a vector, you can use unlist. This really only works if your list has numeric values in it. So we have list_a and list_b are showing up as lists over here. If we unlist them, we will see they show up as vec_a and vec_b, and those are stored as numeric vectors and we could go ahead and add those together if we needed to. That may come in handy as well. That was a lot of information. We'll be talking about data frames coming up soon.

13 Logical vectors

13.1 Introduction

In this chapter, you'll learn tools for working with logical vectors. Logical vectors are the simplest type of vector because each element can only be one of three possible values: `TRUE`, `FALSE`, and `NA`. It's relatively rare to find logical vectors in your raw data, but you'll create and manipulate them in the course of almost every analysis.

We'll begin by discussing the most common way of creating logical vectors: with numeric comparisons. Then you'll learn about how you can use Boolean algebra to combine different logical vectors, as well as some useful summaries. We'll finish off with `if_else()` and `case_when()`, two useful functions for making conditional changes powered by logical vectors.

13.1.1 Prerequisites

Most of the functions you'll learn about in this chapter are provided by base R, so we don't need the tidyverse, but we'll still load it so we can use `mutate()`, `filter()`, and friends to work with data frames. We'll also continue to draw examples from the `nycflights13::flights` dataset.

```
library(tidyverse)
library(nycflights13)
```

However, as we start to cover more tools, there won't always be a perfect real example. So we'll start making up some dummy data with `c()`:

```
x <- c(1, 2, 3, 5, 7, 11, 13)
x * 2
#> [1]  2  4  6 10 14 22 26
```

This makes it easier to explain individual functions at the cost of making it harder to see how it might apply to your data problems. Just remember that any manipulation we do to a free-floating vector, you can do to a variable inside a data frame with `mutate()` and friends.

```
df <- tibble(x)
df |>
  mutate(y = x * 2)
#> # A tibble: 7 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1     2
#> 2     2     4
#> 3     3     6
#> 4     5    10
#> 5     7    14
```

```
#> 6    11    22
#> # ℹ 1 more row
```

13.2 Comparisons

A very common way to create a logical vector is via a numeric comparison with `<`, `<=`, `>`, `>=`, `!=`, and `==`. So far, we've mostly created logical variables transiently within `filter()` — they are computed, used, and then thrown away. For example, the following filter finds all daytime departures that arrive roughly on time:

```
flights |>
  filter(dep_time > 600 & dep_time < 2000 & abs(arr_delay) < 20)
#> # A tibble: 172,286 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      601            600         1      844            850
#> 2  2013     1     1      602            610        -8      812            820
#> 3  2013     1     1      602            605        -3      821            805
#> 4  2013     1     1      606            610        -4      858            910
#> 5  2013     1     1      606            610        -4      837            845
#> 6  2013     1     1      607            607         0      858            915
#> # ℹ 172,280 more rows
#> # ℹ 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>, …
```

It's useful to know that this is a shortcut and you can explicitly create the underlying logical variables with `mutate()`:

```
flights |>
  mutate(
    daytime = dep_time > 600 & dep_time < 2000,
    approx_ontime = abs(arr_delay) < 20,
    .keep = "used"
  )
#> # A tibble: 336,776 × 4
#>   dep_time arr_delay daytime approx_ontime
#>      <int>     <dbl> <lgl>   <lgl>
#> 1      517        11 FALSE   TRUE
#> 2      533        20 FALSE   FALSE
#> 3      542        33 FALSE   FALSE
#> 4      544       -18 FALSE   TRUE
#> 5      554       -25 FALSE   FALSE
#> 6      554        12 FALSE   TRUE
#> # ℹ 336,770 more rows
```

This is particularly useful for more complicated logic because naming the intermediate steps makes it easier to both read your code and check that each step has been computed correctly.

All up, the initial filter is equivalent to:

```
flights |>
  mutate(
    daytime = dep_time > 600 & dep_time < 2000,
    approx_ontime = abs(arr_delay) < 20,
  ) |>
  filter(daytime & approx_ontime)
```

13.2.1 Floating point comparison

Beware of using `==` with numbers. For example, it looks like this vector contains the numbers 1 and 2:

```
x <- c(1 / 49 * 49, sqrt(2) ^ 2)
x
#> [1] 1 2
```

But if you test them for equality, you get `FALSE`:

```
x == c(1, 2)
#> [1] FALSE FALSE
```

What's going on? Computers store numbers with a fixed number of decimal places so there's no way to exactly represent 1/49 or `sqrt(2)` and subsequent computations will be very slightly off. We can see the exact values by calling `print()` with the `digits`[1] argument:

```
print(x, digits = 16)
#> [1] 0.9999999999999999 2.0000000000000004
```

You can see why R defaults to rounding these numbers; they really are very close to what you expect.

Now that you've seen why `==` is failing, what can you do about it? One option is to use `dplyr::near()` which ignores small differences:

```
near(x, c(1, 2))
#> [1] TRUE TRUE
```

13.2.2 Missing values

Missing values represent the unknown so they are "contagious": almost any operation involving an unknown value will also be unknown:

```
NA > 5
#> [1] NA
10 == NA
#> [1] NA
```

The most confusing result is this one:

```
NA == NA
#> [1] NA
```

It's easiest to understand why this is true if we artificially supply a little more context:

```
# We don't know how old Mary is
age_mary <- NA

# We don't know how old John is
age_john <- NA

# Are Mary and John the same age?
age_mary == age_john
#> [1] NA
# We don't know!
```

So if you want to find all flights where `dep_time` is missing, the following code doesn't work because `dep_time == NA` will yield `NA` for every single row, and `filter()` automatically drops missing values:

```
flights |>
  filter(dep_time == NA)
#> # A tibble: 0 × 19
#> # ℹ 19 variables: year <int>, month <int>, day <int>, dep_time <int>,
#> #   sched_dep_time <int>, dep_delay <dbl>, arr_time <int>, …
```

Instead we'll need a new tool: `is.na()`.

13.2.3 `is.na()`

`is.na(x)` works with any type of vector and returns `TRUE` for missing values and `FALSE` for everything else:

```
is.na(c(TRUE, NA, FALSE))
#> [1] FALSE  TRUE FALSE
is.na(c(1, NA, 3))
#> [1] FALSE  TRUE FALSE
is.na(c("a", NA, "b"))
#> [1] FALSE  TRUE FALSE
```

We can use `is.na()` to find all the rows with a missing `dep_time`:

```
flights |>
  filter(is.na(dep_time))
#> # A tibble: 8,255 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1       NA           1630        NA       NA           1815
#> 2  2013     1     1       NA           1935        NA       NA           2240
```

```
#> 3  2013     1     1       NA           1500        NA       NA           1825
#> 4  2013     1     1       NA            600        NA       NA            901
#> 5  2013     1     2       NA           1540        NA       NA           1747
#> 6  2013     1     2       NA           1620        NA       NA           1746
#> # ℹ 8,249 more rows
#> # ℹ 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>, …
```

`is.na()` can also be useful in `arrange()`. `arrange()` usually puts all the missing values at the end but you can override this default by first sorting by `is.na()`:

```
flights |>
  filter(month == 1, day == 1) |>
  arrange(dep_time)
#> # A tibble: 842 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      533            529         4      850            830
#> 3  2013     1     1      542            540         2      923            850
#> 4  2013     1     1      544            545        -1     1004           1022
#> 5  2013     1     1      554            600        -6      812            837
#> 6  2013     1     1      554            558        -4      740            728
#> # ℹ 836 more rows
#> # ℹ 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>, …

flights |>
  filter(month == 1, day == 1) |>
  arrange(desc(is.na(dep_time)), dep_time)
#> # A tibble: 842 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1       NA           1630        NA       NA           1815
#> 2  2013     1     1       NA           1935        NA       NA           2240
#> 3  2013     1     1       NA           1500        NA       NA           1825
#> 4  2013     1     1       NA            600        NA       NA            901
#> 5  2013     1     1      517            515         2      830            819
#> 6  2013     1     1      533            529         4      850            830
#> # ℹ 836 more rows
#> # ℹ 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>, …
```

We'll come back to cover missing values in more depth in Chapter 19.

13.2.4 Exercises

1. How does `dplyr::near()` work? Type `near` to see the source code. Is `sqrt(2)^2` near 2?
2. Use `mutate()`, `is.na()`, and `count()` together to describe how the missing values in `dep_time`, `sched_dep_time` and `dep_delay` are connected.

13.3 Boolean algebra

Once you have multiple logical vectors, you can combine them together using Boolean algebra. In R, `&` is "and", `|` is "or", `!` is "not", and `xor()` is exclusive or[2]. For example, `df |> filter(!is.na(x))` finds all rows where `x` is not missing and `df |> filter(x < -10 | x > 0)` finds all rows where `x` is smaller than -10 or bigger than 0. Figure 13.1 shows the complete set of Boolean operations and how they work.

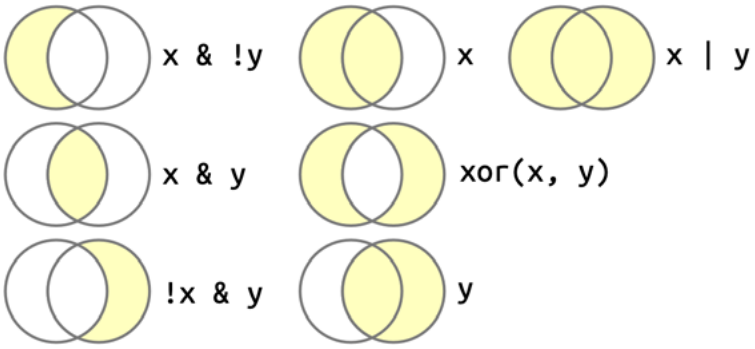


Figure 13.1: The complete set of Boolean operations. `x` is the left-hand circle, `y` is the right-hand circle, and the shaded region show which parts each operator selects.

As well as `&` and `|`, R also has `&&` and `||`. Don't use them in dplyr functions! These are called short-circuiting operators and only ever return a single `TRUE` or `FALSE`. They're important for programming, not data science.

13.3.1 Missing values

The rules for missing values in Boolean algebra are a little tricky to explain because they seem inconsistent at first glance:

```
df <- tibble(x = c(TRUE, FALSE, NA))

df |>
  mutate(
    and = x & NA,
    or = x | NA
  )
#> # A tibble: 3 × 3
#>   x     and   or
#>   <lgl> <lgl> <lgl>
#> 1 TRUE  NA    TRUE
#> 2 FALSE FALSE NA
#> 3 NA    NA    NA
```

To understand what's going on, think about `NA | TRUE` (`NA` or `TRUE`). A missing value in a logical vector means that the value could either be `TRUE` or `FALSE`. `TRUE | TRUE` and `FALSE | TRUE` are both `TRUE` because at least one of them is `TRUE`. `NA | TRUE` must also be `TRUE` because `NA` can either be `TRUE` or `FALSE`. However, `NA | FALSE` is `NA` because we don't know if `NA` is `TRUE` or `FALSE`. Similar reasoning applies with `NA & FALSE`.

13.3.2 Order of operations

Note that the order of operations doesn't work like English. Take the following code that finds all flights that departed in November or December:

```
flights |>
   filter(month == 11 | month == 12)
```

You might be tempted to write it like you'd say in English: "Find all flights that departed in November or December.":

```
flights |>
   filter(month == 11 | 12)
#> # A tibble: 336,776 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      533            529         4      850            830
#> 3  2013     1     1      542            540         2      923            850
#> 4  2013     1     1      544            545        -1     1004           1022
#> 5  2013     1     1      554            600        -6      812            837
#> 6  2013     1     1      554            558        -4      740            728
#> # ℹ 336,770 more rows
#> # ℹ 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>, …
```

This code doesn't error but it also doesn't seem to have worked. What's going on? Here, R first evaluates `month == 11` creating a logical vector, which we call `nov`. It computes `nov | 12`. When you use a number with a logical operator it converts everything apart from 0 to `TRUE`, so this is equivalent to `nov | TRUE` which will always be `TRUE`, so every row will be selected:

```
flights |>
  mutate(
    nov = month == 11,
    final = nov | 12,
    .keep = "used"
  )
#> # A tibble: 336,776 × 3
#>   month nov   final
#>   <int> <lgl> <lgl>
#> 1     1 FALSE TRUE
#> 2     1 FALSE TRUE
#> 3     1 FALSE TRUE
#> 4     1 FALSE TRUE
#> 5     1 FALSE TRUE
#> 6     1 FALSE TRUE
#> # ℹ 336,770 more rows
```

13.3.3 `%in%`

An easy way to avoid the problem of getting your `==`s and `|`s in the right order is to use `%in%`. `x %in% y` returns a logical vector the same length as `x` that is `TRUE` whenever a value in `x` is anywhere in `y`.

```
1:12 %in% c(1, 5, 11)
#>  [1]  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
letters[1:10] %in% c("a", "e", "i", "o", "u")
#>  [1]  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE
```

So to find all flights in November and December we could write:

```
flights |>
  filter(month %in% c(11, 12))
```

Note that `%in%` obeys different rules for `NA` to `==`, as `NA %in% NA` is `TRUE`.

```
c(1, 2, NA) == NA
#> [1] NA NA NA
c(1, 2, NA) %in% NA
#> [1] FALSE FALSE  TRUE
```

This can make for a useful shortcut:

```
flights |>
  filter(dep_time %in% c(NA, 0800))
#> # A tibble: 8,803 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      800            800         0     1022           1014
#> 2  2013     1     1      800            810       -10      949            955
#> 3  2013     1     1       NA           1630        NA       NA           1815
#> 4  2013     1     1       NA           1935        NA       NA           2240
#> 5  2013     1     1       NA           1500        NA       NA           1825
#> 6  2013     1     1       NA            600        NA       NA            901
#> # ℹ 8,797 more rows
#> # ℹ 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>, …
```

13.3.4 Exercises

1. Find all flights where `arr_delay` is missing but `dep_delay` is not. Find all flights where neither `arr_time` nor `sched_arr_time` are missing, but `arr_delay` is.
2. How many flights have a missing `dep_time`? What other variables are missing in these rows? What might these rows represent?
3. Assuming that a missing `dep_time` implies that a flight is cancelled, look at the number of cancelled flights per day. Is there a pattern? Is there a connection between the proportion of cancelled flights and the average delay of non-cancelled flights?

13.4 Summaries

The following sections describe some useful techniques for summarizing logical vectors. As well as functions that only work specifically with logical vectors, you can also use functions that work with numeric vectors.

13.4.1 Logical summaries

There are two main logical summaries: `any()` and `all()`. `any(x)` is the equivalent of `|`; it'll return `TRUE` if there are any `TRUE`'s in `x`. `all(x)` is equivalent of `&`; it'll return `TRUE` only if all values of `x` are `TRUE`'s. Like all summary functions, they'll return `NA` if there are any missing values present, and as usual you can make the missing values go away with `na.rm = TRUE`.

For example, we could use `all()` and `any()` to find out if every flight was delayed on departure by at most an hour or if any flights were delayed on arrival by five hours or more. And using `group_by()` allows us to do that by day:

```
flights |>
  group_by(year, month, day) |>
  summarize(
    all_delayed = all(dep_delay <= 60, na.rm = TRUE),
    any_long_delay = any(arr_delay >= 300, na.rm = TRUE),
    .groups = "drop"
  )
#> # A tibble: 365 × 5
#>    year month   day all_delayed any_long_delay
#>   <int> <int> <int> <lgl>       <lgl>
#> 1  2013     1     1 FALSE       TRUE
#> 2  2013     1     2 FALSE       TRUE
#> 3  2013     1     3 FALSE       FALSE
#> 4  2013     1     4 FALSE       FALSE
#> 5  2013     1     5 FALSE       TRUE
#> 6  2013     1     6 FALSE       FALSE
#> # ℹ 359 more rows
```

In most cases, however, `any()` and `all()` are a little too crude, and it would be nice to be able to get a little more detail about how many values are `TRUE` or `FALSE`. That leads us to the numeric summaries.

13.4.2 Numeric summaries of logical vectors

When you use a logical vector in a numeric context, `TRUE` becomes 1 and `FALSE` becomes 0. This makes `sum()` and `mean()` very useful with logical vectors because `sum(x)` gives the number of `TRUE`s and `mean(x)` gives the proportion of `TRUE`s (because `mean()` is just `sum()` divided by `length()`).

That, for example, allows us to see the proportion of flights that were delayed on departure by at most an hour and the number of flights that were delayed on arrival by five hours or more:

```
flights |>
  group_by(year, month, day) |>
  summarize(
    all_delayed = mean(dep_delay <= 60, na.rm = TRUE),
```

```
    any_long_delay = sum(arr_delay >= 300, na.rm = TRUE),
    .groups = "drop"
  )
#> # A tibble: 365 × 5
#>    year month   day all_delayed any_long_delay
#>   <int> <int> <int>       <dbl>          <int>
#> 1  2013     1     1       0.939              3
#> 2  2013     1     2       0.914              3
#> 3  2013     1     3       0.941              0
#> 4  2013     1     4       0.953              0
#> 5  2013     1     5       0.964              1
#> 6  2013     1     6       0.959              0
#> # ℹ 359 more rows
```

13.4.3 Logical subsetting

There's one final use for logical vectors in summaries: you can use a logical vector to filter a single variable to a subset of interest. This makes use of the base `[` (pronounced subset) operator, which you'll learn more about in Section 28.2.

Imagine we wanted to look at the average delay just for flights that were actually delayed. One way to do so would be to first filter the flights and then calculate the average delay:

```
flights |>
  filter(arr_delay > 0) |>
  group_by(year, month, day) |>
  summarize(
    behind = mean(arr_delay),
    n = n(),
    .groups = "drop"
  )
#> # A tibble: 365 × 5
#>    year month   day behind     n
#>   <int> <int> <int>  <dbl> <int>
#> 1  2013     1     1   32.5   461
#> 2  2013     1     2   32.0   535
#> 3  2013     1     3   27.7   460
#> 4  2013     1     4   28.3   297
#> 5  2013     1     5   22.6   238
#> 6  2013     1     6   24.4   381
#> # ℹ 359 more rows
```

This works, but what if we wanted to also compute the average delay for flights that arrived early? We'd need to perform a separate filter step, and then figure out how to combine the two data frames together[3]. Instead you could use `[` to perform an inline filtering: `arr_delay[arr_delay > 0]` will yield only the positive arrival delays.

This leads to:

```
flights |>
  group_by(year, month, day) |>
  summarize(
    behind = mean(arr_delay[arr_delay > 0], na.rm = TRUE),
    ahead = mean(arr_delay[arr_delay < 0], na.rm = TRUE),
    n = n(),
    .groups = "drop"
  )
#> # A tibble: 365 × 6
#>    year month   day behind ahead     n
#>   <int> <int> <int>  <dbl> <dbl> <int>
#> 1  2013     1     1   32.5 -12.5   842
#> 2  2013     1     2   32.0 -14.3   943
#> 3  2013     1     3   27.7 -18.2   914
#> 4  2013     1     4   28.3 -17.0   915
#> 5  2013     1     5   22.6 -14.0   720
#> 6  2013     1     6   24.4 -13.6   832
#> # ℹ 359 more rows
```

Also note the difference in the group size: in the first chunk `n()` gives the number of delayed flights per day; in the second, `n()` gives the total number of flights.

13.4.4 Exercises

1. What will `sum(is.na(x))` tell you? How about `mean(is.na(x))`?
2. What does `prod()` return when applied to a logical vector? What logical summary function is it equivalent to? What does `min()` return when applied to a logical vector? What logical summary function is it equivalent to? Read the documentation and perform a few experiments.

13.5 Conditional transformations

One of the most powerful features of logical vectors are their use for conditional transformations, i.e. doing one thing for condition x, and something different for condition y. There are two important tools for this: `if_else()` and `case_when()`.

13.5.1 `if_else()`

If you want to use one value when a condition is `TRUE` and another value when it's `FALSE`, you can use `dplyr::if_else()`[4]. You'll always use the first three argument of `if_else()`. The first argument, `condition`, is a logical vector, the second, `true`, gives the output when the condition is true, and the third, `false`, gives the output if the condition is false.

Let's begin with a simple example of labeling a numeric vector as either "+ve" (positive) or "-ve" (negative):

```
x <- c(-3:3, NA)
if_else(x > 0, "+ve", "-ve")
```

```
#> [1] "-ve" "-ve" "-ve" "-ve" "+ve" "+ve" "+ve" NA
```

There's an optional fourth argument, `missing` which will be used if the input is `NA`:

```
if_else(x > 0, "+ve", "-ve", "???")
#> [1] "-ve" "-ve" "-ve" "-ve" "+ve" "+ve" "+ve" "???"
```

You can also use vectors for the the `true` and `false` arguments. For example, this allows us to create a minimal implementation of `abs()`:

```
if_else(x < 0, -x, x)
#> [1]  3  2  1  0  1  2  3 NA
```

So far all the arguments have used the same vectors, but you can of course mix and match. For example, you could implement a simple version of `coalesce()` like this:

```
x1 <- c(NA, 1, 2, NA)
y1 <- c(3, NA, 4, 6)
if_else(is.na(x1), y1, x1)
#> [1] 3 1 2 6
```

You might have noticed a small infelicity in our labeling example above: zero is neither positive nor negative. We could resolve this by adding an additional `if_else()`:

```
if_else(x == 0, "0", if_else(x < 0, "-ve", "+ve"), "???")
#> [1] "-ve" "-ve" "-ve" "0"   "+ve" "+ve" "+ve" "???"
```

This is already a little hard to read, and you can imagine it would only get harder if you have more conditions. Instead, you can switch to `dplyr::case_when()`.

13.5.2 case_when()

dplyr's `case_when()` is inspired by SQL's `CASE` statement and provides a flexible way of performing different computations for different conditions. It has a special syntax that unfortunately looks like nothing else you'll use in the tidyverse. It takes pairs that look like `condition ~ output`. `condition` must be a logical vector; when it's `TRUE`, `output` will be used.

This means we could recreate our previous nested `if_else()` as follows:

```
x <- c(-3:3, NA)
case_when(
  x == 0   ~ "0",
  x < 0    ~ "-ve",
  x > 0    ~ "+ve",
  is.na(x) ~ "???"
)
#> [1] "-ve" "-ve" "-ve" "0"   "+ve" "+ve" "+ve" "???"
```

This is more code, but it's also more explicit.

To explain how `case_when()` works, let's explore some simpler cases. If none of the cases match, the output gets an `NA`:

```
case_when(
  x < 0 ~ "-ve",
  x > 0 ~ "+ve"
)
#> [1] "-ve" "-ve" "-ve" NA    "+ve" "+ve" "+ve" NA
```

Use `.default` if you want to create a "default"/catch all value:

```
case_when(
  x < 0 ~ "-ve",
  x > 0 ~ "+ve",
  .default = "???"
)
#> [1] "-ve" "-ve" "-ve" "???" "+ve" "+ve" "+ve" "???"
```

And note that if multiple conditions match, only the first will be used:

```
case_when(
  x > 0 ~ "+ve",
  x > 2 ~ "big"
)
#> [1] NA    NA    NA    NA    "+ve" "+ve" "+ve" NA
```

Just like with `if_else()` you can use variables on both sides of the `~` and you can mix and match variables as needed for your problem. For example, we could use `case_when()` to provide some human readable labels for the arrival delay:

```
flights |>
  mutate(
    status = case_when(
      is.na(arr_delay)      ~ "cancelled",
      arr_delay < -30       ~ "very early",
      arr_delay < -15       ~ "early",
      abs(arr_delay) <= 15  ~ "on time",
      arr_delay < 60        ~ "late",
      arr_delay < Inf       ~ "very late",
    ),
    .keep = "used"
  )
#> # A tibble: 336,776 × 2
#>   arr_delay status
#>       <dbl> <chr>
#> 1        11 on time
```

```
#> 2        20 late
#> 3        33 late
#> 4       -18 early
#> 5       -25 early
#> 6        12 on time
#> # ℹ 336,770 more rows
```

Be wary when writing this sort of complex `case_when()` statement; my first two attempts used a mix of `<` and `>` and I kept accidentally creating overlapping conditions.

13.5.3 Compatible types

Note that both `if_else()` and `case_when()` require **compatible** types in the output. If they're not compatible, you'll see errors like this:

```
if_else(TRUE, "a", 1)
#> Error in `if_else()`:
#> ! Can't combine `true` <character> and `false` <double>.

case_when(
  x < -1 ~ TRUE,
  x > 0  ~ now()
)
#> Error in `case_when()`:
#> ! Can't combine `..1 (right)` <logical> and `..2 (right)` <datetime<local>>.
```

Overall, relatively few types are compatible, because automatically converting one type of vector to another is a common source of errors. Here are the most important cases that are compatible:

- Numeric and logical vectors are compatible, as we discussed in Section 13.4.2.
- Strings and factors (Chapter 17) are compatible, because you can think of a factor as a string with a restricted set of values.
- Dates and date-times, which we'll discuss in Chapter 18, are compatible because you can think of a date as a special case of date-time.
- `NA`, which is technically a logical vector, is compatible with everything because every vector has some way of representing a missing value.

We don't expect you to memorize these rules, but they should become second nature over time because they are applied consistently throughout the tidyverse.

13.5.4 Exercises

1. A number is even if it's divisible by two, which in R you can find out with `x %% 2 == 0`. Use this fact and `if_else()` to determine whether each number between 0 and 20 is even or odd.

2. Given a vector of days like `x <- c("Monday", "Saturday", "Wednesday")`, use an `ifelse()` statement to label them as weekends or weekdays.

3. Use `ifelse()` to compute the absolute value of a numeric vector called `x`.

4. Write a `case_when()` statement that uses the `month` and `day` columns from `flights` to label a selection of important US holidays (e.g., New Years Day, 4th of July, Thanksgiving, and Christmas). First create a logical column that is either `TRUE` or `FALSE`, and then create a character column that either gives the name of the holiday or is `NA`.

13.6 Summary

The definition of a logical vector is simple because each value must be either `TRUE`, `FALSE`, or `NA`. But logical vectors provide a huge amount of power. In this chapter, you learned how to create logical vectors with `>`, `<`, `<=`, `=>`, `==`, `!=`, and `is.na()`, how to combine them with `!`, `&`, and `|`, and how to summarize them with `any()`, `all()`, `sum()`, and `mean()`. You also learned the powerful `if_else()` and `case_when()` functions that allow you to return values depending on the value of a logical vector.

We'll see logical vectors again and again in the following chapters. For example in Chapter 15 you'll learn about `str_detect(x, pattern)` which returns a logical vector that's `TRUE` for the elements of `x` that match the `pattern`, and in Chapter 18 you'll create logical vectors from the comparison of dates and times. But for now, we're going to move onto the next most important type of vector: numeric vectors.

1. R normally calls print for you (i.e. `x` is a shortcut for `print(x)`), but calling it explicitly is useful if you want to provide other arguments.↩

2. That is, `xor(x, y)` is true if x is true, or y is true, but not both. This is how we usually use "or" In English. "Both" is not usually an acceptable answer to the question "would you like ice cream or cake?".↩

3. We'll cover this in Chapter 20.↩

4. dplyr's `if_else()` is very similar to base R's `ifelse()`. There are two main advantages of `if_else()` over `ifelse()`: you can choose what should happen to missing values, and `if_else()` is much more likely to give you a meaningful error if your variables have incompatible types.↩

R for Data Science (2e) was written by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund.

This book was built with Quarto.