Week 3: Logic & Data Cleaning

Elena & Willa

8/25/2020

Today's agenda

- Check-in + any questions about the program?
- Warm-up: Review data practice from last week
- Logic operators + if/else
- Subsetting vectors
- Subsetting data frames
- Data cleaning

Warm-up: Review data practice from last week

Getting started with logic

Let's imagine we are at the candy store.

I want to buy some delicious treats for all the quack participants.

Without naming the exact candy, describe two different kinds of candy you might want.

Example: *I want something that has chocolate and caramel, or I want something that is candy coated and colorful.*

Type into the chat: What would you want?

Getting started with logic

I tell the clerk, “I want something that has chocolate and caramel, or I want something that is candy coated and colorful. If you don’t have either of those, I’ll have something sour.”

He returns with a bag of twizzlers.

I tell him, “No, that is not what I want.”

Then he returns with a bag of skittles.

I say, “Yes, that is what I want!”

Questions:

1) If he didn’t have Twix or Skittles (or anything that met my first two conditions), and then he brought me sour patch kids, would I purchase them?

2) What are some of the keywords that are important in this story and to getting me the candy I want?

Key words

And, or

Is (equal to), is not (equal to)

If, if/else

Yes/no

Can also imagine: More/greater than, less than

While (e.g., “While something isn’t what I want, keep bringing me another one”)

Equal and not equal

Fun fact: The exclamation point is called a 'bang'! So `!=` can also be read as 'bang equals'.

Yes or no: Does 4 + 5 = 9? Does 4 + 5 = 10? Does 4 + 5 not equal 10?

In R, we use `==` to express 'equals', and `!=` to express 'not equals'.

Let's try it out in R:

1. Type `4 + 5 == 9` into your R console and see what happens!
   - 'Yes' is the same as TRUE, a boolean
2. What do you think will happen when you type `4 + 5 == 10`? Now try it out.
3. What if you type `4 + 5 = 9`? What happens?
4. Guess what will happen when you type `4 + 5 != 10`. Now try it out.
5. Guess what will happen when you type `4 + 5 != 9`. Now try it out.

You can also test equality of strings

Try it out:

1. Test if “hello” is equal to “hello”
2. Test if “hello” is equal to “hella”
3. Test whether “hello” is not equal to “world”

Greater than or less than

> Greater than >= Greater than or equal to

< Less than <= Less than or equal to

On your own, predict what will happen then try it out in R:

1. 15 > 5
2. 15 >= 5
3. 15 >=15
4. 15 >= 15
5. 2 <= 4
6. 2 < 2
7. 2 < 3
8. 2 <= 2
9. “a” > “b”
10. “hello” < “world”
11. “hello” < “hella”
12. “Hello” <= “hello”
13. “hello” < “hello”
14. Is “aa” > or < “ab”? Try it out.

And, or

Think back: I only want my candy if it has chocolate AND caramel

Alternatively, Willa might say she would eat a candy that has chocolate OR caramel.

Would she be happy with a candy that has both? **Most likely, yes!**

And, or

In R, we use `&` to express ‘and’ and `|` to express ‘or’

Our candy example in pseudocode:

- Elena: candyBase == chocolate & candyContains == caramel
- Willa: candyBase == chocolate | candyContains == caramel

Right-hand side

Left-hand side	And	TRUE	FALSE
	TRUE	TRUE	FALSE
	FALSE	FALSE	FALSE

Right-hand side

Left-hand side	Or	TRUE	FALSE
	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE

And, or in R

Predict what will happen then try it out in R:

x = 1, y = 2

1. x + y == 3 | x * y == 3
2. x + y == 3 & x * y == 3

Now, x = 2, y = 2

3. x + y == 4 | x * y == 4
4. x + y == 4 & x * y == 4

%in% operator

This operator let's us know whether an element is in a vector, which is very useful! There are many uses for it, here is one.

What if I want to check whether a child's `grade` is 3, 5 or 7. For whatever reason, I only want these grades, not 4 or 6, so I can't just say grade > 2.

How could I write this?

```
grade == 3 | grade == 5 | grade == 7
```

Try it in R! Do it both ways (| and %in%), first with `grade <- 3`. Then try with `grade <- 4`. What if `grade <- 5`?

R gives us a nicer way of doing this: `grade %in% c(3, 5, 7)`

%in% operator

This operator also gives us more flexibility to use a variable in case what we want changes.

```
grade == 3 | grade == 5 | grade == 7
```

Compared to:

```
possibleGrades <- c(3, 5, 7)

grade %in% possibleGrades
```

If/else

We use conditionals every day. Examples:

- Deciding what to buy at the grocery store - “Get me those TJ’s cookies I love if they have them. If not, just get me some bananas.”
- Planning what we do and when - “If I finish debugging this script I can have a beer tonight”

And we use them in our code, too. Let’s write this in pseudocode: If the store has a chocolate AND caramel candy, I’ll buy it! If not, I won’t buy anything.

if(candyBase == chocolate & candyContains == caramel): buy candy
else: don’t buy candy

If/else

Syantax for if/else in R:

```
if(condition) {

...do something…

} else {

... do something different...

}
```

```
# You can add as many conditions as you want.

if(condition1) {

...do something…

} else if (condition2) {

... do something different...

} else {

... do something different...

}
```

If/else

Predict what will print, then try it out in R:

```
donutsSold <-  25

if (donutsSold > 20) {

    print('You sold a lot!')

} else {

    print('Not enough for today')

}
```

What if `donutsSold <- 20`?

What if `donutsSold <- 5`?

If/else

```
possibleGrades <- c(1, 3, 5, 7, 9, 11)

if (grade %in% possibleGrades) {

    gradeType <- “odd”

} else {

    gradeType <- “even”

}

print(gradeType)
```

1. Assign grade a value that will make this code snippet print “even” at the end.
2. Now assign a value that will make it print “odd”

Remember, R is vectorized!

Vectorized means that operations occur in parallel in certain R objects. This allows you to write code that is efficient, concise, and easier to read than in non-vectorized languages.

Think back to the first week... what will `t` equal? And `z` ?

```
x <- 1:4
y <- 6:9

t <- x + 5
z <- x + y
```

That's right, R operates on each element. It knows that when you use the operator '+' you are saying add that number to each element in the vector. In the case of `z`, it adds pairwise.

So what if we use the operators on vectors instead of individual numbers and characters?

```
x <- 1:4 # --> c(1, 2, 3, 4)
y <- 6:9 # --> c(6, 7, 8, 9)
```

```
x > 2
FALSE   FALSE   TRUE    TRUE
```

```
x >= 2
FALSE   TRUE    TRUE    TRUE
```

```
y == 8
FALSE   FALSE   TRUE    FALSE
```

Important: These calls below are returning vectors of logicals! Try saving one!

```
x == 8
FALSE   FALSE   FALSE   FALSE
```

```
x > 1 & x < 4
FALSE   TRUE    TRUE    FALSE
```

```
x != 3 | x <= 2
TRUE    TRUE    FALSE   TRUE
```

Logical Indexing: “Subsetting” vectors

So we have the vector `y <- 6:9`. What if I only want the numbers in `x` that are greater than 7?

Remember, we can index using bracket notation: `y[1]` gives the first element

We are about to do something super cool… drumroll please…

We can use our logic statements *inside* our bracket notation to get out (keep) the elements that we want!!!

Try this out in R: `y[y > 7]`

Let's break this down to make sure we understand

First, it is important to note: `y[1]` is referring to the *index*, while `y[y > 7]` is referring to the *elements inside of s* that meet a criterium.

```
elementsToKeep <- y > 7 # Returns a logical vector
newY <- y[elementsToKeep]
```

Index	1	2	3	4	
y <- c(	6	7	8	9	)

Let's break this down to make sure we understand

```
elementsToKeep <- y > 7 # Returns a logical vector
newY <- y[elementsToKeep]
```

Notice that this returns ONLY the elements at the indexes that are TRUE!
In this way, it is works "element-wise" or "vectorized"

(Note: Don't worry about understanding this too deeply! We are looking under the hood. But should help us!)

Index	1	2	3	4	
`y <- c(`	6	7	8	9	`)`
`elementsToKeep <- y > 7`	FALSE	FALSE	TRUE	TRUE	

Another useful resource: https://bookdown.org/ndphillips/YaRrr/logical-indexing.html

A practical example: checking for missing values

`is.na()` is a vectorized function that checks whether it is NA, returning TRUE or FALSE; can take value or vector

```
is.na(5)

is.na(NA)

is.na(x)

is.na(c(5, 7, NA, 9, 6))

!is.na(c(5, 7, NA, 9, 6))
```

A practical example: checking for missing values

Here is a vector of kids' ages, one of the columns in our data frame. Our goal is to get rid of the NAs, because we can't do our analyses with missing data.

```
age <- c(5, 10, NA, 6, 9, NA, 8)

is.na(age)

!is.na(age) # We want to keep the values that are NOT NA

age[!is.na(age)]
```

Now, let's do this with a whole data frame.

subjid	age	gender	bilingual
1	5	m	0
2	10	m	1
3	NA	f	0
4	6	m	1
5	9	other	0
6	NA	f	1
7	8	NA	0

Remember, indexes of the columns are the row numbers!

	Index (Row)	1	2	3	4	5	6	7
Vector names (i.e., cols)	`subjid <-`	1	2	3	4	5	6	7
	`age <-`	5	10	NA	6	9	NA	8
	`gender <-`	m	m	f	m	other	f	N
	`bilingual <-`	0	1	0	1	0	1	0

Remember, indexes of the columns are the row numbers!

	Index (Row)	1	2	3	4	5	6	7
Vector names (i.e., cols)	subjid <-	1	2	3	4	5	6	7
	age <-	5	10	NA	6	9	NA	8
	gender <-	m	m	f	m	other	f	Na
	bilingual <-	0	1	0	1	0	1	0
	!is.na(df$age)	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE

```
df[!is.na(df$age), ] # Keeps all indices (i.e., rows) that are TRUE!
```