DEMO - While and if

ER Deyle

Fall 2023; Marine Semester Block 3

We've been talking about derivatives, equilibria, and finding zeros, so the examples in this demo will use those.

In other programming languages, *if* statements are sometimes called *if-then* statements, since the syntax explicitly includes "then". There's also the opportunity to include an alternative piece of code for the cases where the conditional statement is "FALSE". This would constistute and *if-else* statement.

Conditionals

There are a lot of relational operators in R, including `>`, `<`, `==`, `!=`, `>=`, and `<=`. Try these out.

`>`: "Is LHS greater than RHS?"

```
1 > 2
```

```
## [1] FALSE
```

`<`: "Is LHS greater than RHS?"

```
1 < 2
```

```
## [1] TRUE
```

`==`: "Is LHS equal to RHS?"

```
1 == 2
```

```
## [1] FALSE
```

`!=`: "Is LHS not equal to RHS?"

```
1 != 2
```

```
## [1] TRUE
```

`>=`: "Is LHS greater than or equal to RHS?"

```
1 >= 2
```

```
## [1] FALSE
```

`<=`: "Is LHS less than or equal to RHS?"

```
1 <= 2
```

```
## [1] TRUE
```

There are a lot of other functions in R that return a logical (i.e. a `TRUE` or `FALSE` value). Here are some useful ones to check a single variable, vector, or list: `is.empty()`, `is.null()`, `is.na()`, `is.finite()`. There are also a lot of useful set-operations for conditionals, like `%in%`.

```
3 %in% 1:10
```

```
## [1] TRUE
```

```r
13 %in% 1:10
```

```
## [1] FALSE
```

```r
"d" %in% list("a","b","c","d","e")
```

```
## [1] TRUE
```

If

The If construction in R bears some structural similarity to `for` loops. The general syntax is:

```r
if(CONDITION){
  CODE
}
```

The enclosed "CODE" will only evaluate if `CONDITION` is `TRUE`. So, if we simply put the value `TRUE` in the arguement of `if()`, the code will evaluate:

```r
if(TRUE){
  print("Yes!")
}
```

```
## [1] "Yes!"
```

The code will also evaluate if we put an expression in that evaluates to `TRUE`.

```r
if(1 < 2){
  print("Yes!")
}
```

```
## [1] "Yes!"
```

This statement can include variables, but of course they need to have defined values already.

```r
a <- 1
b <- 2

if(a < b){
  print("Yes!")
}
```

```
## [1] "Yes!"
```

When the `CONDITION` is `FALSE`, then R will not evalute the code. For example, if we simply put the value `FALSE` in the arguement of `if()`:

```r
if(FALSE){
  print("Yes!")
}
```

Nothing happened. And as before, the code will also be skipped if we put in an expression that evaluates to `FALSE`.

```r
if(1 > 2){
  print("Yes!")
}
```

`Else`

As I mentioned above, it's possible to present an alternative to the code enclosed in the `{}` using `else`. The general syntax is as follows:

```
if(CONDITION){
  CODE_1
}else{
  CODE_2
}
```

When the `CONDITION` is `TRUE`, the code in the first set of braces (`CODE_1`) is evaluated. Alternatively, when the `CONDITION` is `FALSE`, the code in the second set of braces (`CODE_2`) is evaluated.

```
a <- 1
b <- 2

if(a < b){
  print("Yes!")
}else{
  print("No!")
}
```

```
## [1] "Yes!"
```

```
a <- 3
b <- 2

if(a < b){
  print("Yes!")
}else{
  print("No!")
}
```

```
## [1] "No!"
```

EXERCISE Write an `if` statement to check if a variable is an integer. Use the function `round()` (type `?round`).

```
# YOUR CODE HERE
```

`while`.

In some sense, `while` loops are a blend of `for` and `if`. The basic construction is as follows:

```
while(CONDITION){
  CODE
}
```

R will loop through the `CODE` enclosed in by the `{}` so long as `CONDITION` is `TRUE`. This immediately brings up an important point: if nothing ever changes the `CONDITION`, then it's possible a `while` loop will never finish on its own. When we started with `if` above, we first just set the `CONDITION` to be `TRUE`. Now when we do that, we will have to force R to suspend calculation or it will just keep looping.

```
while(TRUE){
  print("1 second has passed")
  Sys.sleep(1)
}
```

Let's do something constrained instead, like counting. We will start at 1, print the value, then add 1, and tell R to stop at 10.

```
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
```

Notice that as soon as counter got to 10, R stopped executing the code. If we wanted R to do the 10th iteration, we could do a few things like changing the conditional slightly:

```
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
```

We could also be clever about when we check the value of `counter` versus when we print the value of `counter`. If we want to print the value first,

```
counter <- 0

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

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
```

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

We can go back to one of our population models for a fish population and use a `while` loop instead!

```r
N_fish_t <- .5
dt <- 0.05

counter <- 0
while(counter < 40){
  counter <- counter + 1
  N_fish_t <- N_fish_t + (N_fish_t * (1 - N_fish_t))*dt
  print(N_fish_t)
}
```

```
## [1] 0.5125
## [1] 0.5249922
## [1] 0.537461
## [1] 0.5498908
## [1] 0.5622663
## [1] 0.5745725
## [1] 0.5867944
## [1] 0.5989178
## [1] 0.6109285
## [1] 0.6228133
## [1] 0.6345591
## [1] 0.6461538
## [1] 0.6575858
## [1] 0.6688441
## [1] 0.6799187
## [1] 0.6908001
## [1] 0.7014799
## [1] 0.7119502
## [1] 0.7222041
## [1] 0.7322353
## [1] 0.7420387
## [1] 0.7516095
## [1] 0.7609442
## [1] 0.7700396
## [1] 0.7788935
## [1] 0.7875044
## [1] 0.7958715
## [1] 0.8039945
## [1] 0.8118739
## [1] 0.8195106
## [1] 0.8269062
## [1] 0.8340629
## [1] 0.840983
## [1] 0.8476695
## [1] 0.8541258
## [1] 0.8603555
## [1] 0.8663627
## [1] 0.8721516
## [1] 0.8777268
```

```
## [1] 0.8830929
```

Why would we want to? Well there's at least one convenient thing here- we can keep track of the actual time value in the model instead of just the number of steps, so our `while` condition will become

```r
N_fish_t <- .5
dt <- 0.05

t_model <- 0
while(t_model < 2){
  t_model <- t_model + dt
  N_fish_t <- N_fish_t + (N_fish_t * (1 - N_fish_t))*dt
  print(c(t_model,N_fish_t))
}
```

```
## [1] 0.0500 0.5125
## [1] 0.1000000 0.5249922
## [1] 0.150000 0.537461
## [1] 0.2000000 0.5498908
## [1] 0.2500000 0.5622663
## [1] 0.3000000 0.5745725
## [1] 0.3500000 0.5867944
## [1] 0.4000000 0.5989178
## [1] 0.4500000 0.6109285
## [1] 0.5000000 0.6228133
## [1] 0.5500000 0.6345591
## [1] 0.6000000 0.6461538
## [1] 0.6500000 0.6575858
## [1] 0.7000000 0.6688441
## [1] 0.7500000 0.6799187
## [1] 0.8000000 0.6908001
## [1] 0.8500000 0.7014799
## [1] 0.9000000 0.7119502
## [1] 0.9500000 0.7222041
## [1] 1.0000000 0.7322353
## [1] 1.0500000 0.7420387
## [1] 1.1000000 0.7516095
## [1] 1.1500000 0.7609442
## [1] 1.2000000 0.7700396
## [1] 1.2500000 0.7788935
## [1] 1.3000000 0.7875044
## [1] 1.3500000 0.7958715
## [1] 1.4000000 0.8039945
## [1] 1.4500000 0.8118739
## [1] 1.5000000 0.8195106
## [1] 1.5500000 0.8269062
## [1] 1.6000000 0.8340629
## [1] 1.650000 0.840983
## [1] 1.7000000 0.8476695
## [1] 1.7500000 0.8541258
## [1] 1.8000000 0.8603555
## [1] 1.8500000 0.8663627
## [1] 1.9000000 0.8721516
## [1] 1.9500000 0.8777268
## [1] 2.0000000 0.8830929
```

Using `while`.

The real value of a `while` loop, however, is if you want to repeat a calculation a lot of times but you aren't sure exactly *how many times*. Recall before we discussed approximating the derivative using it's original definition:

$$\frac{df}{dx} = lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

How small do we need make the step size to get our numerical estimate of the derivative within a certain tolerance? There's no generic answer, but we can simulate it! Each step of the way, we reduce the step size by 50%.

```
delta_x <- 1
tolerated_error <- 10^-5

f_cubic <- function(x) x^3 - 2*x + 1
dfdx_cubic <- function(x) 3*x^2 - 2

x_star <- 2

current_error <- ( f_cubic(x_star + delta_x) - f_cubic(x_star) )/delta_x - dfdx_cubic(x_star)

while(current_error > tolerated_error){

  # cut the step size by 2 before starting over
  delta_x <- delta_x/2
  current_error <- ( f_cubic(x_star + delta_x) - f_cubic(x_star) )/delta_x - dfdx_cubic(x_star)
  print(c(delta_x,current_error))

}
```

```
## [1] 0.50 3.25
## [1] 0.2500 1.5625
## [1] 0.125000 0.765625
## [1] 0.0625000 0.3789062
## [1] 0.0312500 0.1884766
## [1] 0.01562500 0.09399414
## [1] 0.00781250 0.04693604
## [1] 0.00390625 0.02345276
## [1] 0.001953125 0.011722565
## [1] 0.0009765625 0.0058603287
## [1] 0.0004882812 0.0029299259
## [1] 0.0002441406 0.0014649034
## [1] 0.0001220703 0.0007324368
## [1] 6.103516e-05 3.662147e-04
## [1] 3.051758e-05 1.831064e-04
## [1] 1.525879e-05 9.155297e-05
## [1] 7.629395e-06 4.577637e-05
## [1] 3.814697e-06 2.288818e-05
## [1] 1.907349e-06 1.144409e-05
## [1] 9.536743e-07 5.722046e-06
```