Intro to R for Biologists

IBiS Special Topics, Fall 2021
Class 3: Sept 30, 2021

Erik Andersen and Shelby Blythe

```
for.looper = function(loop.range){
  for(i in loop.range){
    if(i == 1){
      cat('I am a for-loop. \n\nThe first time I run, the variable `i` has the value of ')
      cat(i)
      cat('\nI first run the commands with this value\n')
      cat('and then I loop back to the top of the code... \n\n')
      }else{
        cat('but this time `i` has the value of ')
        cat(i)
        cat('\nI run the commands with this new value, and loop back to the top.\n\n')
      }

    if(i == max(loop.range)) cat('And then I am finished.\n')
  }
}
```

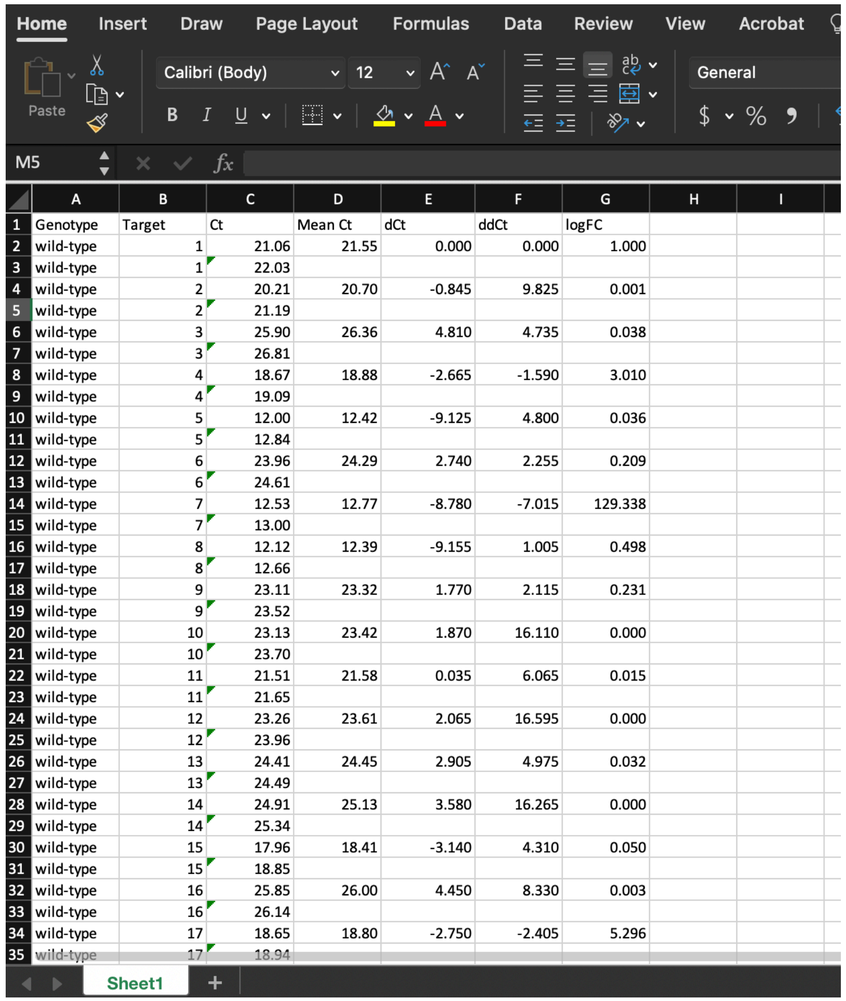
Calculations can be repetitive

- Today's topics of **for-loops, if-else,** and **functions** are the basic tools **R** (and any programming language) provide for *automating* your analyses.

Calculations can be repetitive

Motivation

- Think of a repetitive task in your everyday research (e.g., calculating the mean of dozens of different QPCR replicates, or image analysis, etc.)
- How to do this without excel?
- **First we have to learn to think like a computer.**



	A	B	C	D	E	F	G	H	I
1	Genotype	Target	Ct	Mean Ct	dCt	ddCt	logFC		
2	wild-type	1	21.06	21.55	0.000	0.000	1.000		
3	wild-type	1	22.03						
4	wild-type	2	20.21	20.70	-0.845	9.825	0.001		
5	wild-type	2	21.19						
6	wild-type	3	25.90	26.36	4.810	4.735	0.038		
7	wild-type	3	26.81						
8	wild-type	4	18.67	18.88	-2.665	-1.590	3.010		
9	wild-type	4	19.09						
10	wild-type	5	12.00	12.42	-9.125	4.800	0.036		
11	wild-type	5	12.84						
12	wild-type	6	23.96	24.29	2.740	2.255	0.209		
13	wild-type	6	24.61						
14	wild-type	7	12.53	12.77	-8.780	-7.015	129.338		
15	wild-type	7	13.00						
16	wild-type	8	12.12	12.39	-9.155	1.005	0.498		
17	wild-type	8	12.66						
18	wild-type	9	23.11	23.32	1.770	2.115	0.231		
19	wild-type	9	23.52						
20	wild-type	10	23.13	23.42	1.870	16.110	0.000		
21	wild-type	10	23.70						
22	wild-type	11	21.51	21.58	0.035	6.065	0.015		
23	wild-type	11	21.65						
24	wild-type	12	23.26	23.61	2.065	16.595	0.000		
25	wild-type	12	23.96						
26	wild-type	13	24.41	24.45	2.905	4.975	0.032		
27	wild-type	13	24.49						
28	wild-type	14	24.91	25.13	3.580	16.265	0.000		
29	wild-type	14	25.34						
30	wild-type	15	17.96	18.41	-3.140	4.310	0.050		
31	wild-type	15	18.85						
32	wild-type	16	25.85	26.00	4.450	8.330	0.003		
33	wild-type	16	26.14						
34	wild-type	17	18.65	18.80	-2.750	-2.405	5.296		
35	wild-type	17	18.94						

Repetitive coding of repetitive operations

“Average every pair of values”

- Let’s think through what we do:

```
> set.seed(2021)
> (my.data = data.frame('Ct' = runif(10, min = 15, max = 30)))
         Ct
1  21.76901
2  26.75670
3  25.64523
4  20.72616
5  24.54486
6  25.52019
7  24.60658
8  19.00020
9  27.23132
10 29.74480
```

Repetitive coding of repetitive operations

“Average every pair of values”

- Let’s think through what we do:
- We want to average value
  - 1 & 2
  - 3 & 4
  - 5 & 6
  - 7 & 8
  - 9 & 10
- And we want the values in a new object, in the correct order.

```
> set.seed(2021)
> (my.data = data.frame('Ct' = runif(10, min = 15, max = 30)))
         Ct
1  21.76901
2  26.75670
3  25.64523
4  20.72616
5  24.54486
6  25.52019
7  24.60658
8  19.00020
9  27.23132
10 29.74480
```

Repetitive coding of repetitive operations

“Average every pair of values”

- Let’s think through what we do:
- We want to average value
  - 1 & 2
  - 3 & 4
  - 5 & 6
  - 7 & 8
  - 9 & 10
- And we want the values in a new object, in the correct order.

```
> set.seed(2021)
> (my.data = data.frame('Ct' = runif(10, min = 15, max = 30)))
         Ct
1  21.76901
2  26.75670
3  25.64523
4  20.72616
5  24.54486
6  25.52019
7  24.60658
8  19.00020
9  27.23132
10 29.74480
```

```
>
> my.means = rep(NA, 5) # create a container for the output
>
> my.means[1] = mean(my.data[1:2, "Ct"])
> my.means[2] = mean(my.data[3:4, "Ct"])
> my.means[3] = mean(my.data[5:6, "Ct"])
> my.means[4] = mean(my.data[7:8, "Ct"])
> my.means[5] = mean(my.data[9:10, "Ct"])
>
> my.means
[1] 24.26285 23.18570 25.03252 21.80339 28.48806
```

Perhaps this is where a computer could help.

- You should have the intuition that this is an operation that is very amenable to automation.

```
>
> my.means = rep(NA, 5) # create a container for the output
>
> my.means[1] = mean(my.data[1:2, "Ct"])
> my.means[2] = mean(my.data[3:4, "Ct"])
> my.means[3] = mean(my.data[5:6, "Ct"])
> my.means[4] = mean(my.data[7:8, "Ct"])
> my.means[5] = mean(my.data[9:10, "Ct"])
>
> my.means
[1] 24.26285 23.18570 25.03252 21.80339 28.48806
```

- To automate, we first need to *generalize*:
  - e.g., each line of this calculation takes the form of:

```
my.means[i] = mean(my.data[(2*i-1) : (2*i), “Ct”])
```

Given the general operation, if only there were some way to loop through the relevant values of *i…*

- This is where a **for-loop** can help:
- A **for-loop** will:
  - Take as input a vector of values (numbers, character strings, anything)
  - And perform a series of operations using *each element* of the input vector *in succession*.

For-loops *are computational machines that facilitate repetitive tasks.*

For-Loops

- In R, the general form of a **for-loop** is:

```
for(i in some input vector){
  perform these commands, successively iterating through each value of `i`
}
```

We can apply this readily to the generalized form of our example:

```
my.means[i] = mean(my.data[(2*i-1) : (2*i), “Ct”])
```

For-Loops

- To make a good **for-loop**:
  - Pre-allocate an object to receive the output.
  - Design the generalized form of the operation
  - Decide on the correct set of values to loop through.

```
> set.seed(2021)
> (my.data = data.frame('Ct' = runif(10, min = 15, max = 30)))
         Ct
1  21.76901
2  26.75670
3  25.64523
4  20.72616
5  24.54486
6  25.52019
7  24.60658
8  19.00020
9  27.23132
10 29.74480
```

```
>
> my.means = rep(NA, 5) # create a container for the output
>
> my.means[1] = mean(my.data[1:2, "Ct"])
> my.means[2] = mean(my.data[3:4, "Ct"])
> my.means[3] = mean(my.data[5:6, "Ct"])
> my.means[4] = mean(my.data[7:8, "Ct"])
> my.means[5] = mean(my.data[9:10, "Ct"])
>
> my.means
[1] 24.26285 23.18570 25.03252 21.80339 28.48806
```

For-Loops

- To make a good **for-loop**:
  - Pre-allocate an object to receive the output.
  - Design the generalized form of the operation
  - Decide on the correct set of values to loop through.

```
> set.seed(2021)
> (my.data = data.frame('Ct' = runif(10, min = 15, max = 30)))
         Ct
1  21.76901
2  26.75670
3  25.64523
4  20.72616
5  24.54486
6  25.52019
7  24.60658
8  19.00020
9  27.23132
10 29.74480
```

```
> my.means = rep(NA, 5) # create a container for the output
>
>   for(i in 1 : length(my.means)){
+     my.means[i] = mean(my.data[(2*i-1) : (2*i), "Ct"])
+   }
>
> my.means
[1] 24.26285 23.18570 25.03252 21.80339 28.48806
```

For-Loops

```
> my.means = rep(NA, 5) # create a container for the output
>
>   for(i in 1 : length(my.means)){
+     my.means[i] = mean(my.data[(2*i-1) : (2*i), "Ct"])
+   }
>
> my.means
[1] 24.26285 23.18570 25.03252 21.80339 28.48806
```

```
> my.means = rep(NA, 5) # create a container for the output
>
>   for(i in 1 : length(my.means)){
+     cat(paste0("The value of i is ", i,"\n"))
+     cat(paste0("We are averaging from ", (2*i-1), " to ", (2*i), "\n\n"))
+     #my.means[i] = mean(my.data[(2*i-1) : (2*i), "Ct"])
+   }
The value of i is 1
We are averaging from 1 to 2

The value of i is 2
We are averaging from 3 to 4

The value of i is 3
We are averaging from 5 to 6

The value of i is 4
We are averaging from 7 to 8

The value of i is 5
We are averaging from 9 to 10
```

- You will probably struggle at some point when making these.
- *To troubleshoot **for-loops**: the problem is usually in*
  - The input vector that is looped through (i, in the example), or
  - The generalized form of the function.
  - Assignment of output.

You can always make the code more verbose to see what it is doing.

Sometimes we need options.

IF-THEN-ELSE statements can help

- **IF-THEN-ELSE** statements follow a standard syntax of:
- If the following condition is **TRUE**, then execute this operation.
- If something *different* needs to be done if the condition is **FALSE**, then it can be specified using **ELSE**.

Sometimes we need options.

IF-THEN-ELSE statements can help

- **IF-THEN-ELSE** statements follow a standard syntax of:
- If the following condition is **TRUE**, then execute this operation.
- If something *different* needs to be done if the condition is **FALSE**, then it can be specified using **ELSE**.

```
if(some **logical** argument){
  perform this set of commands
}else{
  perform this set of commands

  (only one or the other set
    of commands is executed.
    Never both.)
}
```

Sometimes we need options.

IF-THEN-ELSE statements can help

```
>
> today = as.Date("2021-09-30")
> today
[1] "2021-09-30"
>
> weekdays(today)
[1] "Thursday"
>
```

- **IF-THEN-ELSE** statements follow a standard syntax of:
- If the following condition is **TRUE**, then execute this operation.
- If something *different* needs to be done if the condition is **FALSE**, then it can be specified using **ELSE**.

```
```{r}

  if(!weekdays(today) %in% c("Monday","Thursday")){
    print("Today, I do labwork.")
  }else{
    print("Today, I learn R.")
  }

```
```

```
[1] "Today, I learn R."
```

Functions

- We use functions all the time in R. We too can write them to make quick work of repetitive tasks. It is easy.

- Let's start by looking at the anatomy of a built-in function, `median`.

First off, how would *you* calculate a median?

- Here's ten numbers. You have to calculate the median. What would you do?

```
27 38 35 25 33 35 33 21 39 44

(Median = 34)
```

median.default

- This is the exact code that R uses to calculate a median.

```
> median.default
function (x, na.rm = FALSE, ...)
{
    if (is.factor(x) || is.data.frame(x))
        stop("need numeric data")
    if (length(names(x)))
        names(x) <- NULL
    if (na.rm)
        x <- x[!is.na(x)]
    else if (any(is.na(x)))
        return(x[FALSE][NA])

    n <- length(x)

    if (n == 0L)
        return(x[FALSE][NA])

    half <- (n + 1L)%/%2L

    if (n%%2L == 1L)
        sort(x, partial = half)[half]
    else mean(sort(x, partial = half + 0L:1L)[half + 0L:1L])
}
```

median.default

- This is the exact code that R uses to calculate a median.
- The function `function`, is itself a function that defines a new function.
- The parenthesis: [`function(x, na.rm = FALSE, …)`] contains the input arguments and options.
- Everything between the curly braces is evaluated.

```
> median.default
function (x, na.rm = FALSE, ...)
{
    if (is.factor(x) || is.data.frame(x))
        stop("need numeric data")
    if (length(names(x)))
        names(x) <- NULL
    if (na.rm)
        x <- x[!is.na(x)]
    else if (any(is.na(x)))
        return(x[FALSE][NA])

    n <- length(x)

    if (n == 0L)
        return(x[FALSE][NA])

    half <- (n + 1L)%/%2L

    if (n%%2L == 1L)
        sort(x, partial = half)[half]
    else mean(sort(x, partial = half + 0L:1L)[half + 0L:1L])
}
```

median.default

- Within the braces, there are three stages.
  - Parsing the input arguments
  - Checking that input arguments are OK
  - Performing a calculation

```
> median.default
function (x, na.rm = FALSE, ...)
{
    if (is.factor(x) || is.data.frame(x))
        stop("need numeric data")
    if (length(names(x)))
        names(x) <- NULL
    if (na.rm)
        x <- x[!is.na(x)]
    else if (any(is.na(x)))
        return(x[FALSE][NA])

    n <- length(x)

    if (n == 0L)
        return(x[FALSE][NA])

    half <- (n + 1L)%/%2L

    if (n%%2L == 1L)
        sort(x, partial = half)[half]
    else mean(sort(x, partial = half + 0L:1L)[half + 0L:1L])
}
```

median.default

- The highlighted arguments both parse the input arguments as well as check whether there are any problems (e.g.: non-numeric arguments were provided, the vector has a length of zero).

```
> median.default
function (x, na.rm = FALSE, ...)
{
    if (is.factor(x) || is.data.frame(x))
        stop("need numeric data")
    if (length(names(x)))
        names(x) <- NULL
    if (na.rm)
        x <- x[!is.na(x)]
    else if (any(is.na(x)))
        return(x[FALSE][NA])

    n <- length(x)

    if (n == 0L)
        return(x[FALSE][NA])

    half <- (n + 1L)%/%2L

    if (n%%2L == 1L)
        sort(x, partial = half)[half]
    else mean(sort(x, partial = half + 0L:1L)[half + 0L:1L])
}
```

median.default

- The highlighted regions here actually calculate the median of the input vector.

Can you make sense of it?

- (Note, the “L” following a digit is R’s way of saying that the digit is an *integer*).

In the real world, 1L = 1.

- Also “%/%” stands for integer division

(11 %/% 2 = 5).

- “%%” stands for modulus

(11 %% 2 = 1)

```
> median.default
function (x, na.rm = FALSE, ...)
{
    if (is.factor(x) || is.data.frame(x))
        stop("need numeric data")
    if (length(names(x)))
        names(x) <- NULL
    if (na.rm)
        x <- x[!is.na(x)]
    else if (any(is.na(x)))
        return(x[FALSE][NA])

    n <- length(x)

    if (n == 0L)
        return(x[FALSE][NA])

    half <- (n + 1L)%/%2L

    if (n%%2L == 1L)
        sort(x, partial = half)[half]
    else mean(sort(x, partial = half + 0L:1L)[half + 0L:1L])
}
```

Making your own functions:

- You can put *anything* in a function.
- The call to **function** includes variables (here **my.vector**, and **tell.me.odds**).
- These can have *default values*, or can be user specified.
- The code in between {curly braces} is evaluated *with those variables*.
- What happens in the function stays in the function.
- To have output, you must **return** it.

```
# this function will take an input variable (my.variable) and report either
the even- or odd-indexed values.

my.nice.function = function(my.vector, tell.me.odds = FALSE){

  if(tell.me.odds){
    output = my.vector[seq(1, length(my.vector), by = 2)]
  }else{
    output = my.vector[seq(2, length(my.vector), by = 2)]
  }
  return(output)
}
```

Making your own functions:

- The *assignment* (**my.nice.function**) places the **function** into the workspace.

```
# this function will take an input variable (my.variable) and report either
the even- or odd-indexed values.

my.nice.function = function(my.vector, tell.me.odds = FALSE){

  if(tell.me.odds){
    output = my.vector[seq(1, length(my.vector), by = 2)]
  }else{
    output = my.vector[seq(2, length(my.vector), by = 2)]
  }
  return(output)
}
```

Making your own functions:

- The *assignment* (**my.nice.function**) places the **function** into the workspace.
- You can then run the function from the command line, or in a script.
- **Tips:**
  - A function is another generalization. Avoid having it depend on objects in your workspace. Also avoid using object names you tend to use in your workspace!
  - Build in error-checking steps to avoid problems.
  - If you don't specify **return()**, the function will return the output of the last operation.

```
# this function will take an input variable (my.variable) and report either
the even- or odd-indexed values.

my.nice.function = function(my.vector, tell.me.odds = FALSE){

  if(tell.me.odds){
    output = my.vector[seq(1, length(my.vector), by = 2)]
  }else{
    output = my.vector[seq(2, length(my.vector), by = 2)]
  }
  return(output)
}
```

```
>
>
> my.nice.function( LETTERS[1:10], tell.me.odds = TRUE)
[1] "A" "C" "E" "G" "I"
>
>
```

In-class activity time.

- Exploration of **for-loops**.

- **Bonus:** Initial digression on loading a package, and how to figure out what's in it.