Lecture 6: Writing Functions in R

STAT UN2102 *Applied Statistical Computing*

Gabriel Young
Columbia University

March 6, 2018

Last Time

- **Character Data in** `R`. Commands like `nchar()`, `paste()`, `strsplit()`, `substr()`, `grep()`.
- **Regular Expressions**. A grammar (lots of rules) to works with patterns of characters.
- **Web Scraping**. Data from the internet.

Functions

Functions in R

Why Functions?

- Data structures tie related values into one object.
- Functions tie related commands into one object.
- Both cases: easier to understand, work with, and to build into larger structures.

Functions in R

Basic Structure

```
function_name <- function(arg1, arg2, ... ) {
  statements
  return(object)
}
```

Functions in R

Basic Structure

```
function_name <- function(arg1, arg2, ... ) {
  statements
  return(object)
}
```

- A **function** is a group of instructions that takes inputs, uses them to compute other values, and returns a result.
- We can write and add our own functions in R.
- Functions:
  1. Have names.
  2. *Usually* take in arguments.
  3. Include body of code that does something.
  4. *Usually* return an object at the end.

Example Function

A Function to Check for Significance at $\alpha = 0.05$

```
> # Input x should be a single p-value in [0,1]
> significant <- function(x) {
+   if (x <= 0.05) { return(TRUE) }
+   else { return(FALSE) }
+ }
```

Example Function

A Function to Check for Significance at $\alpha = 0.05$

```
> # Input x should be a single p-value in [0,1]
> significant <- function(x) {
+   if (x <= 0.05) { return(TRUE) }
+   else { return(FALSE) }
+ }
```

- First, tell `R` to define a function named `significant`.
- Brackets `{` and `}` mark the start and close of the body.
- `R` tells you you're in the body of the function by using $+$ as a prompt (instead of $>$).
- At the end, use the `return()` command.

Example Function

A Piecewise Function ('Robust' Loss Function in Regression)

```
> # Inputs:  A vector of numbers (x)
> # Outputs: A loss vector with x^2 for small elements,
> #          and 2|x|-1 for large ones
>
> res_loss <- function(x) {
+   loss_vec <- ifelse(x^2 > 1, 2*abs(x) - 1, x^2)
+   return(loss_vec)
+ }
```

Example Function

A Piecewise Function ('Robust' Loss Function in Regression)

```
> # Inputs:  A vector of numbers (x)
> # Outputs: A loss vector with x^2 for small elements,
> #          and 2|x|-1 for large ones
>
> res_loss <- function(x) {
+   loss_vec <- ifelse(x^2 > 1, 2*abs(x) - 1, x^2)
+   return(loss_vec)
+ }
```

Let's try it:

```
> vec <- c(-0.5, 0.9, -3, 4)
> res_loss(vec)
```

```
[1] 0.25 0.81 5.00 7.00
```

Example Function

A Piecewise Function ('Robust' Loss Function in Regression)

```
> res_loss <- function(x) {
+   loss_vec <- ifelse(x^2 > 1, 2*abs(x) - 1, x^2)
+   return(loss_vec)
+ }
```

Break apart the function

What are the...

- Inputs?
- Outputs?
- Body Statements?

Example Function

A Piecewise Function ('Robust' Loss Function in Regression)

```
> res_loss <- function(x) {
+   loss_vec <- ifelse(x^2 > 1, 2*abs(x) - 1, x^2)
+   return(loss_vec)
+ }
```

Break apart the function

What are the...

- Inputs? `x`
- Outputs? `loss_vec`
- Body Statements?

```
loss_vec <- ifelse(x^2 > 1, 2*abs(x) - 1, x^2)
return(loss_vec)
```

When Should We Make a Function?

- Things you will rerun.
- Chunks of code that are small parts of bigger analyses.

Check Yourself

Task

Write a function called `FiveTimesSum` that takes as input a vector of numerical values and returns 5 times the sum of those values. Test it on the vector `1:3`. Your output should be 30.

Check Yourself

Task

Write a function called `FiveTimesSum` that takes as input a vector of numerical values and returns 5 times the sum of those values. Test it on the vector `1:3`. Your output should be 30.

Solution

```
> FiveTimesSum <- function(vec){
+   return(5*sum(vec))
+ }
> FiveTimesSum(1:3)
```

```
[1] 30
```

Named and Default Arguments

A Piecewise Function ('Robust' Loss Function in Regression)

```
> # Inputs: A vector of numbers (x),
> #         crossover location (c > 0)
> # Outputs: A loss vector with x^2 for small elements,
> #          and 2c|x|-c for large ones
>
> res_loss2 <- function(x, c = 1) {
+   loss_vec <- ifelse(x^2 > c, 2*sqrt(c)*abs(x) - c, x^2)
+   return(loss_vec)
+ }
```

Named and Default Arguments

A Piecewise Function ('Robust' Loss Function in Regression)

```
> # Inputs: A vector of numbers (x),
> #         crossover location (c > 0)
> # Outputs: A loss vector with x^2 for small elements,
> #          and 2c|x|-c for large ones
>
> res_loss2 <- function(x, c = 1) {
+   loss_vec <- ifelse(x^2 > c, 2*sqrt(c)*abs(x) - c, x^2)
+   return(loss_vec)
+ }
```

Let's try it:

```
> identical(res_loss(vec), res_loss2(vec, c=1))
```

```
[1] TRUE
```

Named and Default Arguments

A Piecewise Function ('Robust' Loss Function in Regression)

```
> # Inputs: A vector of numbers (x),
> #         crossover location (c > 0)
> # Outputs: A loss vector with x^2 for small elements,
> #          and 2c|x|-c for large ones
>
> res_loss2 <- function(x, c = 1) {
+   loss_vec <- ifelse(x^2 > c^2, 2*sqrt(c)*abs(x) - c, x^2)
+   return(loss_vec)
+ }
```

Let's try it:

```
> identical(res_loss(vec), res_loss2(vec, c=2))
```

```
[1] FALSE
```

Named and Default Arguments

Default values get used if names are missing:

```
> identical(res_loss2(vec, c=1), res_loss2(vec))
```

```
[1] TRUE
```

Named and Default Arguments

Default values get used if names are missing:

```
> identical(res_loss2(vec, c=1), res_loss2(vec))
```

```
[1] TRUE
```

Named argument can go in any order when they are explicitly tagged:

```
> identical(res_loss2(x=vec, c=2), res_loss2(c=2, x=vec))
```

```
[1] TRUE
```

Checking Arguments

Funny things can happen when arguments aren't as we expect:

```
> vec <- c(-0.5, 0.9, -3, 4)
> res_loss2(vec, c = c(1,1,1,5))
```

```
[1]  0.25  0.81  5.00 16.00
```

Checking Arguments

Funny things can happen when arguments aren't as we expect:

```
> vec <- c(-0.5, 0.9, -3, 4)
> res_loss2(vec, c = c(1,1,1,5))
```

```
[1]  0.25  0.81  5.00 16.00
```

```
> res_loss2(vec, c = -1)
```

```
[1] 0.25 0.81  NaN  NaN
```

Checking Arguments

Solution: Add some checks to your function.

A 'Robust' Loss Function (for Outlier-Resistant Regression)

```
> res_loss2 <- function(x, c = 1) {
+   # Scale should be a single positive number
+   stopifnot(length(c) == 1, c > 0)
+   loss_vec <- ifelse(x^2 > c^2, 2*sqrt(c)*abs(x) - c, x^2)
+   return(loss_vec)
+ }
```

Checking Arguments

Solution: Add some checks to your function.

A 'Robust' Loss Function (for Outlier-Resistant Regression)

```
> res_loss2 <- function(x, c = 1) {
+   # Scale should be a single positive number
+   stopifnot(length(c) == 1, c > 0)
+   loss_vec <- ifelse(x^2 > c^2, 2*sqrt(c)*abs(x) - c, x^2)
+   return(loss_vec)
+ }
```

`stopifnot()`

- Arguments are a series of expressions which should all be `TRUE`.
- Execution stops with error at first `FALSE`.

Checking Arguments

Solution: Add some checks to your function.

A 'Robust' Loss Function (for Outlier-Resistant Regression)

```
> res_loss2 <- function(x, c = 1) {
+   # Scale should be a single positive number
+   stopifnot(length(c) == 1, c > 0)
+   loss_vec <- ifelse(x^2 > c^2, 2*sqrt(c)*abs(x) - c, x^2)
+   return(loss_vec)
+ }
```

Test it:

```
> # res_loss2(vec, c = c(1,1,1,5))
> # res_loss2(vec, c = -1)
```

Check Yourself

Task

Write a function called `KTimesSum` that takes as input a vector of numerical values and a scalar value K (with a default value of 5). The function should return the sum of those values multiplied times the value K. Test it with the following: `KTimesSum(1:3)` and `KTimesSum(1:3, K = 10)`.

Check Yourself

Task

Write a function called `KTimesSum` that takes as input a vector of numerical values and a scalar value K (with a default value of 5). The function should return the sum of those values multiplied times the value K. Test it with the following: `KTimesSum(1:3)` and `KTimesSum(1:3, K = 10)`.

Solution

```
> KTimesSum <- function(vec, K = 5){
+   return(K*sum(vec))
+ }
> KTimesSum(1:3); KTimesSum(1:3, K = 10)
```

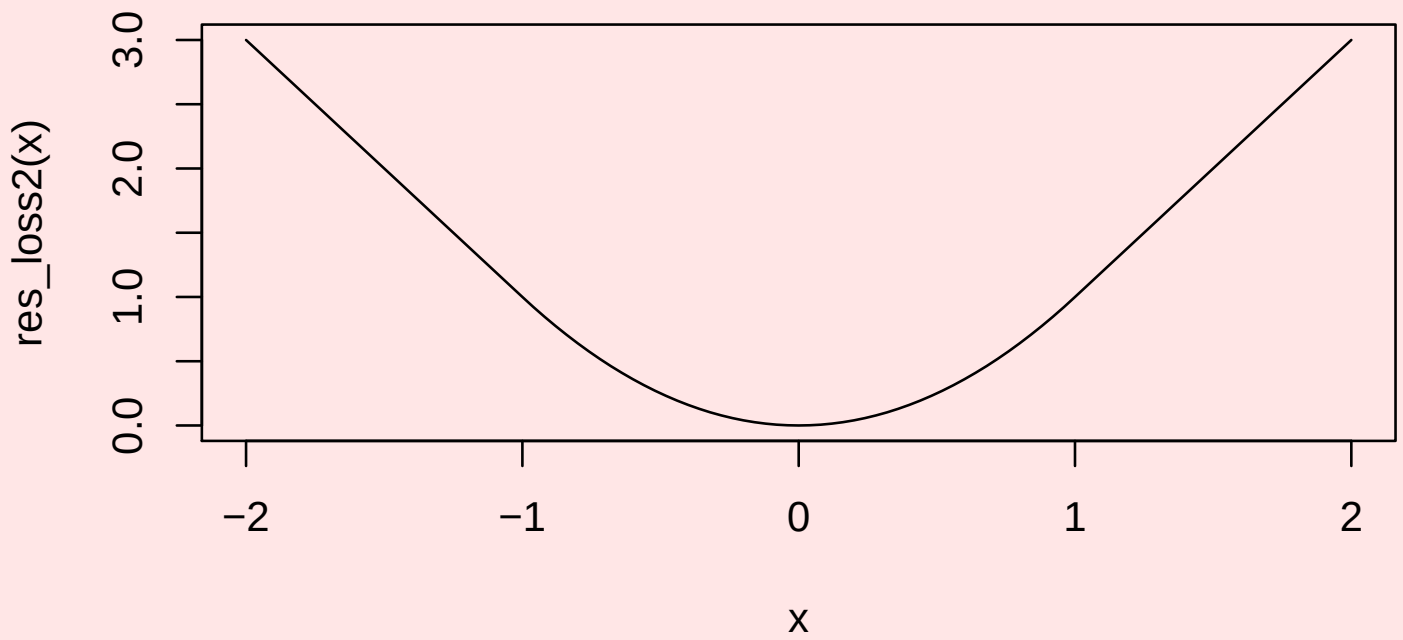
```
[1] 30
```

```
[1] 60
```

Use Your Functions in Other Functions

- Use your own function in built-in `R` functions like `apply()`.
- Ex: `curve(expression, from = , to = )` plots a curve.

```
> curve(res_loss2, from = -2, to = 2)
```



The R Environment

- The **global environment** (or the workspace) in R consists of the collection of your named objects.
- When you start an R session, a new environment is initialized (unless you load a saved environment).
- When a function is called, a new **local environment** is created within the body of the function.

The R Environment

Code Example.

The Function Environment

- Each function has its own (internal) **environment**.
- Names in the function environment override names from the **global environment**.
- Assignments in the internal environment don't change the global environment.
- Functions search for named variables (undefined in the function itself) in the environment in which the function was created (in our case, the global environment).

The Function Environment

```
> x <- 7
> y <- c("dog", "cat")
> addition <- function(y) {x <- x + y; return(x)}
> addition(1)
```

```
[1] 8
```

The Function Environment

```
> x <- 7
> y <- c("dog", "cat")
> addition <- function(y) {x <- x + y; return(x)}
> addition(1)
```

```
[1] 8
```

```
> x
```

```
[1] 7
```

```
> y
```

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

The Function Environment

```
> circle.area <- function(r) {return(pi*r^2)}
> circle.area(1:3)
```

```
[1]  3.141593 12.566371 28.274334
```

The Function Environment

```
> circle.area <- function(r) {return(pi*r^2)}
> circle.area(1:3)
```

```
[1]  3.141593 12.566371 28.274334
```

```
> true.pi <- pi # Save the real value
> pi      <- 3 # Assign a new value
> circle.area(1:3)
```

```
[1]  3 12 27
```

The Function Environment

```
> circle.area <- function(r) {return(pi*r^2)}
> circle.area(1:3)
```

```
[1]  3.141593 12.566371 28.274334
```

```
> true.pi <- pi # Save the real value
> pi      <- 3 # Assign a new value
> circle.area(1:3)
```

```
[1]  3 12 27
```

```
> pi <- true.pi # Restore the real value
> circle.area(1:3)
```

```
[1]  3.141593 12.566371 28.274334
```

Use Your Function Interfaces

The function **interfaces** are the places where the function interacts with the global environment: at the **inputs** and the **outputs**.

Use Your Function Interfaces

The function **interfaces** are the places where the function interacts with the global environment: at the **inputs** and the **outputs**.

- Interact with the rest of the system only at the interfaces:
  - Arguments should give your function all the information it needs
  - Reduces the risk of bugs
  - Exception would be universal constants like `pi`.
- Output should be only through `return()`.

Capstone Example: Two-Sample T-test

Capstone Example (My T-test Function)

The goal of this example is to write a function (named `my.t.test`) that runs the basic two-sample t-test procedure.

Requirements of `my.t.test`.

- Inputs
  - Variable X_1.
  - Variable X_2.
  - Hypothesized value (with default set at 0).
  - Direction ("Upper","Lower","Two") with default set at "Two".
  - Significance level as decimal. Must be strictly between 0 and 1 with default at $\alpha = .05$.
- The output should include the relevant statistical results, e.g.,
  - Two Sample t-test:
  - Hypothesized value
  - Test statistic with df
  - Direction and p-value
  - Reject rule at significance level

Recall the Two-Sample t-test

Consider testing the null hypothesis

$$H_0 : \mu_1 - \mu_2 = \Delta_0.$$

Assumptions of the two sample t-test

1. $X_1, X_2, \ldots, X_m$ is a random sample from a normal distribution with mean μ_1 and variance σ_1^2.
2. $Y_1, Y_2, \ldots, Y_n$ is a random sample from a normal distribution with mean μ_2 and variance σ_2^2.
3. The X and Y samples are independent of one another.

Classic setting

Does resting heart rate of 35 year old males statistically differ between the control group and the dosage group? Here X is the control group and Y is the dosage group.

Recall the Two-Sample t-test

Consider testing the null hypothesis $H_0 : \mu_1 - \mu_2 = \Delta_0$.

Test Statistic

- The test statistic is

$$t_{calc} = \frac{\bar{x} - \bar{y} - \Delta_0}{\sqrt{\frac{s_1^2}{m} + \frac{s_2^2}{n}}},$$

  where $\bar{x}, \bar{y}, s_1^2, s_2^2$ are the respective sample means and sample variances of datasets X and Y.

- The approximate degrees of freedom is given by

$$df = \frac{\left(\frac{s_1^2}{m} + \frac{s_2^2}{n}\right)^2}{\frac{(s_1^2/m)^2}{m-1} + \frac{(s_2^2/n)^2}{n-1}}$$

Note: Use `pt()` with t_{calc} and df to compute the relevant p-values.

Capstone Example (My T-test Function)

Alternative Hypothesis	P-value calculation
$H_A : \mu_1 - \mu_2 > \Delta_0$ (upper-tailed)	$P(t_{calc} > T)$
$H_A : \mu_1 - \mu_2 < \Delta_0$ (lower-tailed)	$P(t_{calc} < T)$
$H_A : \mu_1 - \mu_2 \neq \Delta_0$ (two-tailed)	$2 * P(\lvert t_{calc} \rvert > T)$

Reject H_0 when:

$$Pvalue \leq \alpha$$

Capstone Example (My T-test Function)

The goal of this example is to write a function (named `my.t.test`) that runs the basic two-sample t-test procedure.

Requirements of `my.t.test`.

- Inputs
  - Variable X_1.
  - Variable X_2.
  - Hypothesized value (with default set at 0).
  - Direction ("Upper","Lower","Two") with default set at "Two".
  - Significance level as decimal. Must be strictly between 0 and 1 with default at $\alpha = .05$.
- The output should include the relevant statistical results, e.g.,
  - Two Sample t-test:
  - Hypothesized value
  - Test statistic with df
  - Direction and p-value
  - Reject rule at significance level

Capstone Example (My T-test Function)

Use the `paste()` function to create organized output.

Example Output 1:

Two Sample t-test:
Hypothesized value = 0.
Test statistic -2.5134 with df = 62.9976.
Lower-tailed p-value = 0.00726.
Reject H0 at 5% significance.

Example Output 2:

Two Sample t-test:
Hypothesized value = -1.
Test statistic -1.4574 with df = 62.9976.
Two-tailed p-value = 0.15.
Fail to reject H0 at 5% significance.

Capstone Example (My T-test Function)

The goal of this example is to write a function (named `my.t.test`) that runs the basic two-sample t-test procedure.

Main function

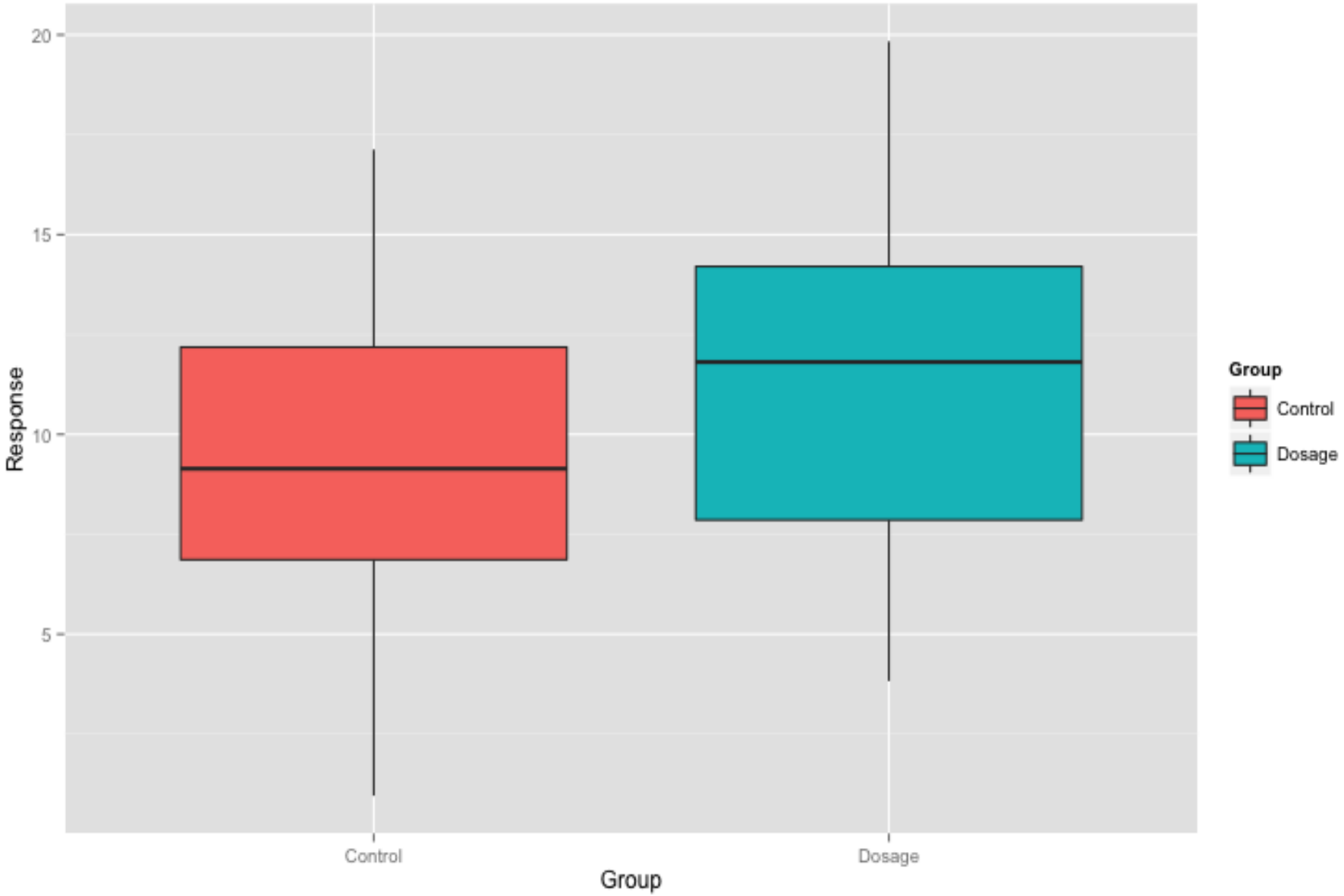
```
> my.t.test <- function(X1,
+                       X2,
+                       level=.05,
+                       direction="Two",
+                       value=0) {
+
+ ## ----------  Body of function -------------
+
+ return(output)
+ }
>
```

Capstone Example (My T-test Function)

Simulate and plot a dataset

```
> # Sim data
> set.seed(8)
> Control <- rnorm(30,mean=10,sd=3)
> Dosage <- rnorm(35,mean=12,sd=4.3)
> # Stack data
> data.stacked <- data.frame(Response=c(Control,Dosage),
+                            Group=c(rep("Control",30),
+                                    rep("Dosage",35)
+                                    )
+                            )
> # plot  data
> library(ggplot2)
> ggplot(data=data.stacked)+
+   geom_boxplot(mapping=aes(y=Response,x=Group,fill=Group))
```

Capstone Example (My T-test Function)



Capstone Example (My T-test Function)

Start with defining a function for degrees of freedom

$$df = \frac{\left(\frac{s_1^2}{m} + \frac{s_2^2}{n}\right)^2}{\frac{(s_1^2/m)^2}{m-1} + \frac{(s_2^2/n)^2}{n-1}}$$

```
> df.function <- function(X,Y) {
+
+   m <- length(X)
+   n <- length(Y)
+   numerator <- (var(X)/m+var(Y)/n)^2
+   denominator <- (var(X)/m)^2/(m-1)+(var(Y)/n)^2/(n-1)
+   return(numerator/denominator)
+
+ }
```

Capstone Example (My T-test Function)

Start with defining a function for degrees of freedom

$$df = \frac{\left(\frac{s_1^2}{m} + \frac{s_2^2}{n}\right)^2}{\frac{(s_1^2/m)^2}{m-1} + \frac{(s^2/n)^2}{n-1}}$$

```
> # Test function
> df.function(X=Control,Y=Dosage)
```

```
[1] 62.97384
```

Capstone Example (My T-test Function)

Consider testing the null hypothesis $H_0 : \mu_1 - \mu_2 = 0$.

Next calculate the test statistic

$$t_{calc} = \frac{\bar{x} - \bar{y} - \Delta_0}{\sqrt{\frac{s_1^2}{m} + \frac{s_2^2}{n}}},$$

```
> X <- Control
> Y <- Dosage
> # Test Statistic
> m <- length(X)
> n <- length(Y)
> t.stat <- ((mean(X)-mean(Y))-0)/sqrt(var(X)/m+var(Y)/n)
> t.stat
```

```
[1] -2.34238
```

Capstone Example (My T-test Function)

Next calculate the p-values

```
> df.est <- df.function(X,Y)  # compute df
> 2*(1-pt(abs(t.stat),df=df.est))  # two tailed
```

```
[1] 0.02233566
```

```
> (1-pt(t.stat,df=df.est))  # upper tailed
```

```
[1] 0.9888322
```

```
> pt(t.stat,df=df.est)  # lower tailed
```

```
[1] 0.01116783
```

Capstone Example (My T-test Function)

Conclusion for two-tailed 5% level test

```
> level <- .05
> p.value <- 2*(1-pt(abs(t.stat),df=df.est))
> # Paste results
> con <- paste("Two Sample t-test:",
+              "\nHypothesisized value = 0.",
+             "\nTest statistic ",round(t.stat,4),
+             " with df = ",round(df.est,4),
+             ". \n","Two-tailed p-value = ",
+             signif(p.value,3),".\n","Reject H0 at ",
+             round(level*100,1),"% significance.",sep="")
```

Capstone Example (My T-test Function)

Conclusion for two-tailed 5% level test

```
> cat(con)
```

```
Two Sample t-test:
Hypothesisized value = 0.
Test statistic -2.3424 with df = 62.9738.
Two-tailed p-value = 0.0223.
Reject H0 at 5% significance.
```

Capstone Example (My T-test Function)

Conclusion for two-tailed 5% level test

```
> cat(con)
```

```
Two Sample t-test:
Hypothesisized value = 0.
Test statistic -2.3424 with df = 62.9738.
Two-tailed p-value = 0.0223.
Reject H0 at 5% significance.
```

How do we put all of this together in one function? We also want the output to change per dataset.

Capstone Example (My T-test Function)

The goal of this example is to write a function (named `my.t.test`) that runs the basic two-sample t-test procedure.

Main function

```
> my.t.test <- function(X1,
+                       X2,
+                       level=.05,
+                       direction="Two",
+                       value=0) {
+
+ ## ----------  Body of function -------------
+
+ output <- cat(con)
+ return(output)
+ }
>
```

Capstone Example (My T-test Function)

See R full script for complete function

Full function

```
> my.t.test <- function(X,
+                       Y=NULL,
+                       level=.05,
+                       direction="Two",
+                       value=0) {
+
+     m <- length(X)
+     n <- length(Y)
+     df.est <- df.function(X,Y)
+     t.stat <- ((mean(X)-mean(Y))-value)/sqrt(var(X)/m+var(Y
+     if (direction=="Two") {p.value <- 2*(1-pt(abs(t.stat),d
+     if (direction=="Upper") {p.value <- (1-pt(t.stat,df=df.
+     if (direction=="Lower") {p.value <- pt(t.stat,df=df.est
+     sig <- ifelse(p.value<=level,"Reject H0","Fail to rejec
```

Capstone Example (My T-test Function)

Test the function `my.t.test()`

```
> my.t.test(X=Control,
+           Y=Dosage,
+           level=.05,
+           direction="Two",
+           value=0)
```

```
Two Sample t-test:
Hypothesized value = 0.
Test statistic -2.3424 with df = 62.9738.
Two-tailed p-value = 0.0223.
Reject H0 at 5% significance.NULL
```

Capstone Example (My T-test Function)

Compare with the R t.test() function

```
> t.test(Control,Dosage)
```

```
Welch Two Sample t-test

data:  Control and Dosage
t = -2.3424, df = 62.974, p-value = 0.02234
alternative hypothesis: true difference in means is not equal
95 percent confidence interval:
 -3.8396132 -0.3043072
sample estimates:
mean of x mean of y
 9.698627 11.770587
```

Check Yourself

Task

Run `my.t.test()` function with different inputs and different simulated datasets.

```
> # Task 1
> # Try:  direction="Lower"
> #
> # Task 2
> set.seed(8)
> Control <- rnorm(10,mean=10,sd=3)
> Dosage <- rnorm(8,mean=12,sd=4.3)
> #
> # Task 3
> set.seed(8)
> Control <- rnorm(30,mean=10,sd=3)
> Dosage <- rnorm(35,mean=11,sd=4.3)
```

Check Yourself

Task

- Generalize `my.t.test()` to also include the **one-sample t-test**.
- The updated function should automatically detect if the input is a one-sample or two-sample procedure.
- To accomplish this task, use `if()` statements.
- Recall

$$t_{calc} = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}, \qquad df = n - 1$$

Solution: See R full Script for complete function