MATH 3050 – Predictive Analytics

UNC CHARLOTTE

Topic 1: The R Programming Language – Part 2

- ❑ If…Then…Else
- ❑ Loops
- ❑ User Defined Functions
- ❑ Tables

UNC CHARLOTTE 1

1

Objectives of this Lesson:

By the end of this lesson you should be able to:

- Create if...then...else statements
- Use loops to iterate through functions
- Write custom functions
- Create data tables

UNC CHARLOTTE 2

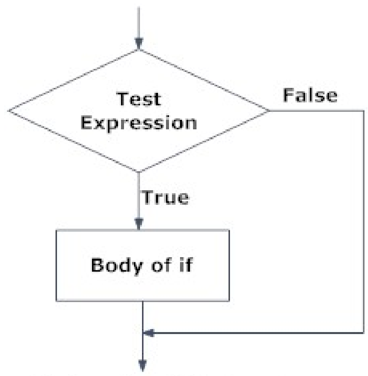
2

If...then...else Statements:

The if() statement
The syntax of if statement is:

```
if (test_expression)
  {
  statements
  }
```

If the test_expression is TRUE, the statement gets executed. But if it's FALSE, nothing happens.



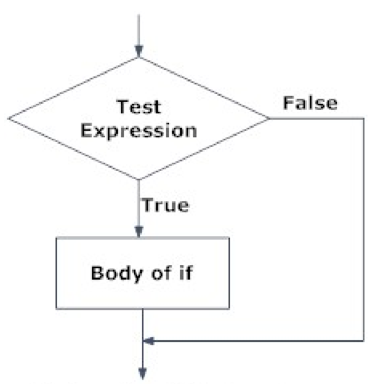
If...then...else Statements:

Example:

```
x <- 5
if(x > 0)
{
print("Positive number")
}
```

Output

```
[1] "Positive number"
```



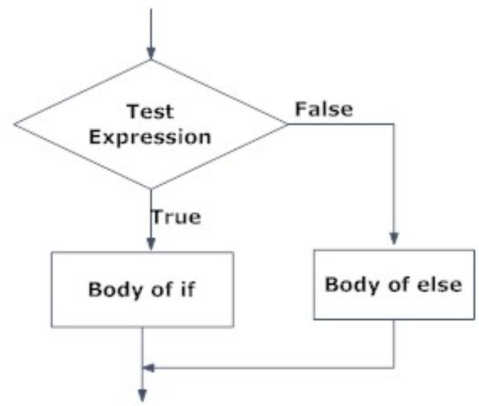
If…then…else Statements:

if…else () statement

The syntax of if…else statement is:

```
if (test_expression)
  {
  statements
  } else
  {
  statements
  }
```

else MUST appear after the brace or you will get an error.



The else part is optional and is only evaluated if test_expression is FALSE.

It is important to note that else must be in the same line as the closing braces of the if statement.

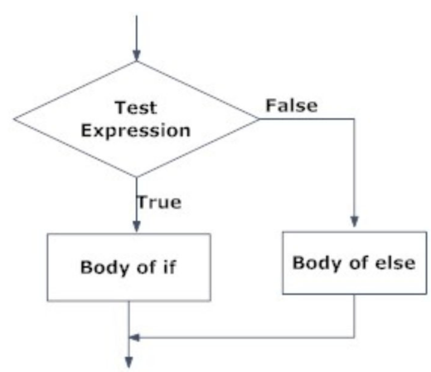
If…then…else Statements:

Example of if…else statement

```
x <- -5
if(x > 0)
 {
 print("Non-negative number")
 } else
 {
 print("Negative number")
 }
```

Output

```
[1] "Negative number"
```



#The above conditional can also be written in a single line as follows.

if(x > 0) print("Non-negative number") else print("Negative number")

If…then…else Statements:

The if…else ladder (if…else…if) statement allows you execute a block of code among more than 2 alternatives

The syntax of if…else statement is:

```
if ( test_expression1)
  {
  statements
  } else if ( test_expression2)
  {
  statements
  } else if ( test_expression3)
  {
  statements
  } else
  {
  statements
  }
```

Notice how the last alternative is preceded by an “else” and not an “else if”. This is how we terminate the ladder if.

If…then…else Statements:

Example of nested if…else

```
x <- 0
if (x < 0)
  {
  print("Negative number")
  } else if (x > 0)
  {
  print("Positive number")
  } else print("Zero")
```

Output

```
[1] "Zero"
```

If…then…else Statements:

R ifelse() Function

This is a shorthand function to the traditional if…else statement.
Works like the if() statement in excel.

Syntax of ifelse() function

ifelse(test_expression, x, y)

- test_expression must be a logical vector (or an object that can be coerced to a logical). The return value is a vector with the same length as test_expression.
- x corresponds the TRUE value of the test_expression
- y corresponds the FALSE value of the test_expression
- Test_expression, x and y can be vectors

If…then…else Statements:

R ifelse() Function

The ifelse() function is something called a "**vectorized function**." This means it can only return vector values and not the construction of values.

The "cat()" function constructs a sentence and the ifelse() does not like that. You can use the paste() function because it just copies and pastes. Although the result of the two functions is the same, the way these functions are defined in R is different.

That is how they wrote the language.

If…then…else Statements:

R ifelse() vs. if() and if..else()

An Important Difference:

if() and if..else() **should not** be applied when the Condition being evaluated is a vector. It is best used only when meeting a single element condition. In most applications the condition is an element not related to the data object being manipulated. The ifelse() works with vectors

Thus it can be applied to a column of data within a data object.

If…then…else Statements:

R ifelse() vs. {if() and if..else()}

Example:

if() and **if..else()** don't work with vectors, only ifelse() does!

```
month <- c(1, 4, 10, 8, 7)
sp<-c(3,4,5)
su<-c(6,7,8)
f<-c(9,10,11)

season <- ifelse(month %in% sp, "Spring", ifelse(month %in% su, "Summer", ifelse(month %in% f, "Fall","Winter" ) ) )

season = if (month %in% sp){"Spring"}else if (month %in% su){"Summer"}else if (month %in% f){"Fall"} else {"Winter"}
season
```

```
Warning messages:
1: In if (month %in% sp) { :
  the condition has length > 1 and only the first element will be used
2: In if (month %in% su) { :
  the condition has length > 1 and only the first element will be used
3: In if (month %in% f) { :
  the condition has length > 1 and only the first element will be used
```

While this is a warning message and not technically an error, you get the wrong answer.

If...then...else Statements:

Example: ifelse() function

```
> a = c(5,7,2,9)

> ifelse(a %% 2 == 0,"even","odd")

[1] "odd"  "odd"  "even" "odd"
```

In the above example, the test_expression is a %% 2 == 0 which will result into the vector (FALSE,FALSE,TRUE ,FALSE).

If...then...else Statements:

Example: ifelse() function

```
v1 <- c(1,2,3,4,5,6)
v2 <- c("a","b","c","d","e","f")

ifelse(c(TRUE,FALSE,TRUE,FALSE,TRUE,FALSE), v1, v2)
```

Output

```
[1] "1" "b" "3" "d" "5" "f"
```

- This statement draws from v1 if condition TRUE is encountered and v2 if condition FALSE is encountered.
- v1 and v2 can also be functions (i.e., subroutines).

Exampes: If…then…else Statements:

```
x <- c("what","is","truth")

if("Truth" %in% x) {
  print("Truth is found the first time")
} else if ("truth" %in% x) {
  print("truth is found the second time")
} else {
  print("No truth found")
}
```

Notice this code is less structured than in the previous examples. You will all have to make sure the braces {} match up or you will get a syntax error.

Output

[1] "truth is found the second time"

Exampes: If…then…else Statements:

```
category <- 'A'
price <- 10

if (category =='A'){ cat('A vat rate of 8% is applied.','The total price is',price *1.08)
} else if (category =='B'){cat('A vat rate of 10% is applied.','The total price is',price *1.10)
} else { cat('A vat rate of 20% is applied.','The total price is',price *1.20)}
```

Output

[1] A vat rate of 8% is applied. The total price is 10.8

Observations:
1. Notice the structure of the code. It is easy to match up the braces{}.
2. You have a new function: **cat()**. It concatenates and prints. This is an advantage over print.

Concatenate and Print Function

Description
Outputs the objects, concatenating the representations. “cat” performs much less conversion than “print”.

Usage
cat(... , file = "", sep = " ", fill = FALSE, labels = NULL, append = FALSE)

Concatenate and Print Function

Arguments

`...`	R objects (see ‘Details’ for the types of objects allowed).
`file`	A connection, or a character string naming the file to print to. If `""` (the default), `cat` prints to the standard output connection, the console unless redirected by `sink`.
`sep`	a character vector of strings to append after each element.
`fill`	a logical or (positive) numeric controlling how the output is broken into successive lines. If `FALSE` (default), only newlines created explicitly by `"\n"` are printed. Otherwise, the output is broken into lines with print width equal to the option `width` if `fill` is `TRUE`, or the value of `fill` if this is numeric. Non-positive `fill` values are ignored, with a warning.
`labels`	character vector of labels for the lines printed. Ignored if `fill` is `FALSE`.
`append`	logical. Only used if the argument `file` is the name of file (and not a connection or `"|cmd"`). If `TRUE` output will be appended to `file`; otherwise, it will overwrite the contents of `file`.

Concatenate and Print Function

Examples

```
iter <- rpois(1, lambda = 10)  #Give 1 random number from a Poisson
                                distribution with lambda =10

## print an informative message
cat("iteration = ", iter <- iter + 1, "\n")

## 'fill' and label lines:
cat(paste(letters, 100* 1:26), fill = TRUE, labels = paste0("{", 1:10, "}:"))
```


Another Important Concatenation

How to Combine Vectors in R

```
> baskets.of.Granny <- c(12, 4, 4, 6, 9, 3)
> baskets.of.Geraldine <- c(5, 3, 2, 2, 12, 9)
> all.baskets <-c(baskets.of.Granny, baskets.of.Geraldine)

> all.baskets
 [1] 12 4 4 6 9 3 5 3 2 2 12 9
```

OR

```
>baskets.of.Granny <- c(12, 4, 4, 6, 9, 3)
>baskets.of.Geraldine <- c(5, 3, 2, 2, 12, 9)
>baskets.of.Granny <-c(baskets.of.Granny, baskets.of.Geraldine)

>baskets.of.Granny
 [1] 12 4 4 6 9 3 5 3 2 2 12 9
```

Both of these examples are useful for building vectors, especially if you have to build them in a loop.

Another Important Concatenation

How to Combine Matrices in R using rbind()

```
x = matrix(1:12, ncol=3)
y = matrix(13:24, ncol=3)

print("Matrix-x")
print(x)

print("Matrix-y")
print(y)

z <-rbind(x,y)
print("Matrix-z")
print(z)
```

"**rbind()**" means to combine the two matrices row wise.

```
#We could also write:
x <-rbind(x,y)
print("Matrix-x")
print(x)
```

Output

```
> print("Matrix-x")
[1] "Matrix-x"
> print(x)
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
>
> print("Matrix-y")
[1] "Matrix-y"
> print(y)
     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
>
> z <-rbind(x,y)
> print("Matrix-z")
[1] "Matrix-z"
> print(z)
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
[5,]   13   17   21
[6,]   14   18   22
[7,]   15   19   23
[8,]   16   20   24
```


Another Important Concatenation

How to Combine Matrices in R using rbind()

```
x = matrix(1:12, ncol=3)
y = matrix(13:24, ncol=3)

print("Matrix-x")
print(x)

print("Matrix-y")
print(y)

z <-cbind(x,y)
print("Matrix-z")
print(z)
```

"**cbind()**" means to combine the two matrices column wise.

```
#We could also write:
x <-cbind(x,y)
print("Matrix-x")
print(x)
```

Output

```
> print("Matrix-x")
[1] "Matrix-x"
> print(x)
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
>
> print("Matrix-y")
[1] "Matrix-y"
> print(y)
     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
>
> z <-cbind(x,y)
> print("Matrix-z")
[1] "Matrix-z"
> print(z)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    2    6   10   14   18   22
[3,]    3    7   11   15   19   23
[4,]    4    8   12   16   20   24
```

Homework

Create a script to do the following using nested if…else statements:

1. Prompts the user to enter their age and store the result in the variable my.age. Hint use the line of code: `my.age <- as.integer(readline(prompt="Please Enter your Age: "))`

2. Evaluate the user input as follows:
   a. If the user is < 18, print the following:
      - You are Not a Major.
      - You are Not Eligible to Work.
   b. If the user is at least 18 and not older than 60, print:
      - You are Eligible to Work.
      - Please fill the Application Form and Email to us.
   c. If the user is older than 60, print:
      - As per the Government Rules, You are too Old to Work.
      - Please Collect your pension!
3. Print the final comment: "This Message is from Outside the Nested IF Else Statement" outside the if…else as the last statement.

Homework

Create a script to do the following using an ifelse() statement:

1. Store the following vector of prices in the variable apple.
   - c(109.49,109.90,109.11,109.95,111.03,112.12)

2. Create an ifelse() statement that tests each of the prices against the value 110.
3. If the price is less than 110, print the result "buy the apple stock".
4. Otherwise print "don't buy the apple stock".

R For Loop

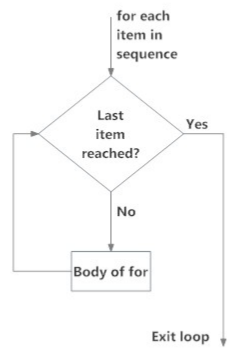
Loops are used in programming to repeat a specific block of code.

Syntax of For Loop

```
for (val in sequence)
  {
  Statements
  }
```

This is the index of the loop. The values to iterate through during the execution of the loop.

Flowchart of for loop



25

R For Loop

Example: For Loop

Below is an example to count the number of even numbers in a vector.

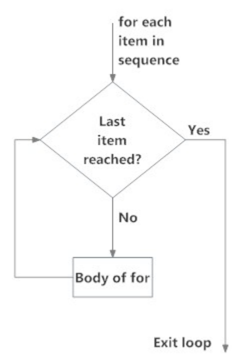
```
x <- c(2,5,3,9,8,11,6)
count <- 0

for (val in x) #R will run through each value in x.  The loop will execute 7 times.
  {
  if(val %% 2 == 0)  count = count+1
  }
print(count)
```

Output

[1] 3   Note: The output implies 3 values in x are evenly divisible by 2.

Flowchart of for loop



26

R While Loop

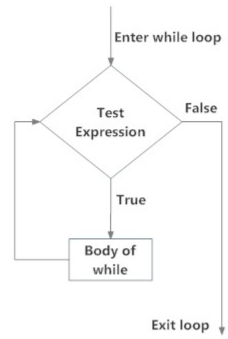
In R programming, while loops are used to loop until a specific condition is met.

Syntax of while loop

```
while (test_expression)
  {
  statement
  }
```

Notes:
- The body of the loop is entered if test_expression is TRUE.
- The statements inside the loop are executed.
- Flow returns to evaluate the test_expression again.
- This is repeated until test_expression evaluates to FALSE.
- Then control exits the loop.

Flowchart of while Loop



27

R While Loop

Example of while Loop

```
i <- 1
while (i < 6)
   {
   print(i)
   i = i+1
   }
```

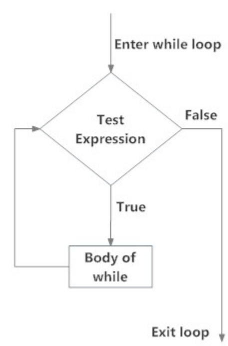
Any kind of object can be executed within the TRUE portion of loop.

Output

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

When the counter hits 6, control exits the loop.

Flowchart of while Loop



28

R break and next Statement

In R programming, a normal looping sequence can be altered using the break or the next statement. This means we can break out of a loop when a condition is met.

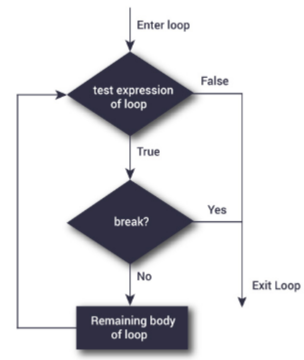
break statement
A break statement is used inside a loop to stop the iterations and flow the control outside of the loop.

In a nested looping situation, where there is a loop inside another loop, this statement exits from the innermost loop that is being evaluated.

The syntax of break statement is:

```
if (test_expression)
  {
  break
  }
```

Flowchart of break statement



R break and next Statement

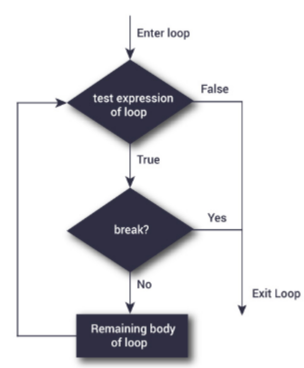
Example: break statement

```
x <- 1:5
for (val in x)
 {
if (val == 3)
  {
  break
  }
print(val)
 }
```

Output

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

Flowchart of break statement



R break and next Statement

next statement
A next statement is useful when we want to skip the current iteration of a loop without terminating it.

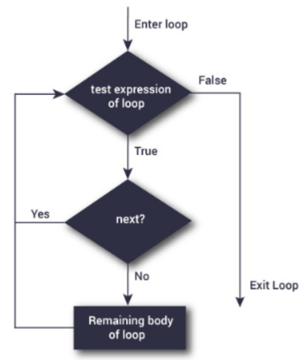
On encountering next, the R parser skips further evaluation and starts next iteration of the loop.

The syntax of next statement is:

```
if (test_condition)
  {
  next
  }
```

Remember "next i" from other programming languages?

Flowchart of next statement



31

Topic 1: The R Programming Language – Part 2

The Next Statement

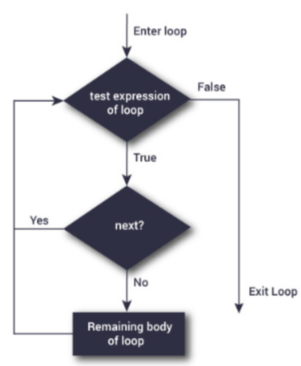
Example: Using length a a control index.

```
x <- 1:5
for (val in x) {
if (val == 3){
next
}
print(val)
}
```

Output

```
[1] 1
[1] 2
[1] 4
[1] 5
```

Flowchart of next statement



32

Loops in R

Homework

An Armstrong number, also known as narcissistic number, is a number that is equal to the sum its own digits where each digit is raised to a power equal to the length of the number.

For example, 370 is an Armstrong number since 370 = 3*3*3 + 7*7*7 + 0*0*0.
1634 is an Armstrong number since 1634 = 1*1*1*1 + 6*6*6*6 + 3*3*3*3 + 4*4*4*4

Write a script to
1. Prompt the user to input a number and store the number in num.
   Hint: `num = as.integer(readline(prompt="Enter a number: "))`
2. Prompt the user to enter the length of the number and store the result in len. You can also determine the length through code.
3. Use a loop to determine if the number is an Armstrong number. You will have to pick off each digit.
4. Print "[num] is an Armstrong number." if num is an Armstrong number and print "num is NOT an Armstrong number." otherwise.
5. Substitute the actual number for [num] in the printout.

The following are Armstrong numbers for your testing: 1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407, 1634, 8208, 9474, 54748.

Loops in R

Homework

Armstrong Numbers

Number	Digits					Exponent	Digits^Exponent					Sum
1	1					1	1	0	0	0	0	1
2	2					1	2	0	0	0	0	2
3	3					1	3	0	0	0	0	3
4	4					1	4	0	0	0	0	4
5	5					1	5	0	0	0	0	5
6	6					1	6	0	0	0	0	6
7	7					1	7	0	0	0	0	7
8	8					1	8	0	0	0	0	8
9	9					1	9	0	0	0	0	9
153	1	5	3			3	1	125	27	0	0	153
370	3	7	0			3	27	343	0	0	0	370
371	3	7	1			3	27	343	1	0	0	371
407	4	0	7			3	64	0	343	0	0	407
1,634	1	6	3	4		4	1	1,296	81	256	0	1,634
8,208	8	2	0	8		4	4,096	16	0	4,096	0	8,208
9,474	9	4	7	4		4	6,561	256	2,401	256	0	9,474
54,748	5	4	7	4	8	5	3,125	1,024	16,807	1,024	32,768	54,748

Loops in R

Homework

The Fibonacci Sequence is a sequence of numbers where a sequence number is generated from the sum of the prior two numbers. The first two numbers are defined as 1, 1. Then the second number generated is 2, the third is 3, the fourth is 5, etc.

Write a script to

1. Generate the first 15 numbers of the Fibonacci sequence.
2. Store the results in the vector Fibonacci
3. Print the vector

Custom Functions in R

R Functions

Functions are used to logically break our code into simpler parts which become easy to maintain and understand.

It's pretty straightforward to create your own function in R programming.

Syntax for Writing Functions in R

```
func_name <- function (argument(s))
  {
  A Bunch of Statements
  }
```

The reserved word **function** is used to declare a function in R.

Custom Functions in R

R Functions

Example of a Function

```
pow <- function(x, y)
   {
   # function to print x raised to the power y
   result <- x^y
   print(paste(x, "raised to the power", y, "is", result))
   }
```

It takes two arguments. The first argument is the base number, and the second argument is the power. It then prints the result in the indicated format.

We have used the built-in function **paste()** to concatenate strings.

Custom Functions in R

R Functions

How to call a function?
We can call the above function as follows.

```
>pow(8, 2)
[1] "8 raised to the power 2 is 64"

> pow(2, 8)
[1] "2 raised to the power 8 is 256"
```

Custom Functions in R

Named Arguments

In the above function calls, the argument matching the formal argument to the actual arguments takes place in positional order.

This means that, in the call pow(8,2), the formal arguments x and y are assigned 8 and 2 respectively.

We can also call the function using named arguments.

When calling a function in this way, the order of the actual arguments doesn't matter. For example, all of the function calls given below are equivalent.

```
> pow(8, 2)
[1] "8 raised to the power 2 is 64"

> pow(x = 8, y = 2)
[1] "8 raised to the power 2 is 64"

> pow(y = 2, x = 8)
[1] "8 raised to the power 2 is 64"
```

We can use named and unnamed arguments in a single call.

```
>pow(x=8, 2)
[1] "8 raised to the power 2 is 64"

> pow(2, x=8)
[1] "8 raised to the power 2 is 64"
```


Topic 1: The R Programming Language – Part 2

Custom Functions in R

Default Values for Arguments

We can assign default values to arguments in a function in R.

This is done by providing an appropriate value to the formal argument in the function declaration.

Here is the above function with a default value for y.

```
pow <- function(x, y = 2)
  {
  # function to print x raised to the power y
  result <- x^y
  print(paste(x,"raised to the power", y, "is", result))
  }
```

The use of a default value as an argument makes it optional when calling the function. y is optional and will take the value 2 when not provided.

```
> pow(3)
[1] "3 raised to the power 2 is 9"
> pow(3,1)
[1] "3 raised to the power 1 is 3"
```

Custom Functions in R

R Return Value from Function

Many times, we require our functions to do some processing and return the result.
This is accomplished with the return() function in R.

Syntax of return()

```
return(expression)
```

The value returned from a function can be any valid object.

Custom Functions in R

R Return Value from Function

Example: return()
Let's look at an example which will return whether a given number is positive, negative or zero.

```
check <- function(x)
  {
  if (x > 0) {
  result <- "Positive"
  }else if (x < 0)
  {
  result <- "Negative"
  }else
  {
  result <- "Zero"
  }
return(result)
}
```

Here are some sample runs:

```
> check(1)
[1] "Positive"

> check(-10)
[1] "Negative"

> check(0)
[1] "Zero"
```

Custom Functions in R

Functions without return()
If there are no explicit returns from a function, the value of the last evaluated expression is returned automatically in R.

For example, the following is equivalent to the above function.

```
check <- function(x) {
if (x > 0) {
result <- "Positive"
} else if (x < 0) {
result <- "Negative"
} else {
result <- "Zero"
}
result
}
```


Custom Functions in R

We generally use explicit return() functions to return a value immediately from a function.

If it is not the last statement of the function, it will prematurely end the function bringing the control to the place from which it was called.

```
check <- function(x)
{
  if (x>0) {return("Positive")
  }else if (x<0) {return("Negative")
  }else { return("Zero")}
}
```

In the above example, if x > 0, the function immediately returns "Positive" without evaluating rest of the body.

Custom Functions in R

Multiple Returns

The return() function can return only a single object. If we want to return multiple values in R, we can use a list (or other objects) and return it.

Following is an example.

```
multi_return <- function()
  {
  my_list <- list("color" = "red", "size" = 20, "shape" = "round")
  return(my_list)
  }
```

Here, we create a list my_list with multiple elements and return this single list.

```
> a <- multi_return()
> a$color
[1] "red"
> a$size
[1] 20
> a$shape
[1] "round"
```


Custom Functions in R

You can return multiple values by saving the results in a vector (or a list) and returning it.

Example:

```
math <- function(x, y) {
  add <- x + y
  sub <- x - y
  mul <- x * y
  div <- x / y
  c(addition = add, subtraction = sub, multiplication = mul, division = div)
  }

math(6, 3)
```

There is no explicit return function. Returned the last implicit assignment.

```
addition    subtraction multiplication       division
       9              3             18              2
```

Custom Functions in R

R Function with No Arguments

```
# R function with no arguments

sayHello = function(){
   print("Hello !")
}

sayHello()
```


47

Topic 1: The R Programming Language – Part 2

Custom Functions in R

R Function with Arguments

```
# R function with arguments
addition = function(a,b,c){
   print(a+b+c)
}

addition(4,15,6)
25
```


48

Custom Functions in R

R Function with Arguments and Return Value

```
# R function with arguments and return value

addition = function(a,b,c){
  return (a+b+c)
}

d = addition(4,15,6)

print(d)
25
```

Notes:

1. Functions can only return one value at a time, but the value can be a vector or matrix.
2. You can assign the return value to a variable and then perform operations on the return variable.

Custom Functions in R

R Function – Uses a Loop to Create a Count

```
# R function with arguments and return value

GrandCount <- function(c)
{
  count <- 0
  for (i in 1:length(c)) { count <- count + i}
  count

}

d <-  c(1:1000)

GrandCount (d)
```

Notes:

1. Functions can only return one value at a time, but the value can be a vector or matrix.
2. You can assign the return value to a variable and then perform operations on the return variable.

Topic 1: The R Programming Language – Part 2

Custom Functions in R

R Function – Uses a Loop to Create a Sum

```
# R function with arguments and return value

GrandCount <- function(c)
{
  count <- 0
  for (i in 1:length(c)) { count <- count + i}
  count

}

d <-  c(1:1000)

GrandCount (d)
```

Notes:

1. Functions can only return one value at a time, but the value can be a vector or matrix.
2. You can assign the return value to a variable and then perform operations on the return variable.

UNC CHARLOTTE 51

51

Topic 1: The R Programming Language – Part 2

Custom Functions in R

R Function – Uses a Loop to Create a Sum

```
# R function with arguments and return value

GrandCount <- function(c)
{
  count <- 0
  for (i in 1:length(c)) { count <- count + i}
  count

}

d <-  c(1:1000)

GrandCount <- (d)
```

Note the matching braces {}

UNC CHARLOTTE 52

52

Topic 1: The R Programming Language – Part 2

Custom Functions in R

R Function – Uses a Loop to Create a Sum

```
# R function with arguments and return value

GrandCount <-function(c)
{
   count <- 0
   for (i in 1:length(c)) { count <- count + i}
   count

}

d <-  c(1:1000)

GrandCount(d)
```

{All of your function code goes between the matching braces}

Topic 1: The R Programming Language – Part 2

Custom Functions in R

R Function – Uses a Loop to Create a Sum

```
# R function with arguments and return value

GrandCount <- function(c)
{
   count <- 0
   for (i in 1:length(c)) { count <- count + i}
   count

}

d <-  c(1:1000)

GrandCount (d)
```

Don’t forget to indicate your calculated quantity. This is the value that is returned by your function.

Custom Functions in R

R Function – Uses a Loop to Create a Sum

```
# R function with arguments and return value

GrandTotal <- function(c)
{
   Sum <- 0
   for (i in 1:length(c)) { Sum <- Sum + c[i]}
   Sum

}

d <-  c(1:10)

GrandTotal (d)
```

Don't forget to indicate your calculated quantity. This is the value that is returned by your function.

What value is returned?_____

Functions in R

Homework

Write a script to:
1. Prompt the user to input an integer and store the integer in the variable num.
2. Create a function to determine the number of digits in a number.
3. Use the function to determine how many digits are in the number and store the result in len.
4. Print "The [num] has [len] digits."

Hint: Try function: `nDigits <- function(x) nchar(trunc( abs(x) ) )`

Functions in R

Homework

Create a function called "Times" that given a vector and an integer will return how many times the integer appears inside the vector.

Example: vec<-c(1,2,3,3,4,4,5,5,2,6,4,6,3,8,9,7,7,7,7). Int = 6. The function will return "The number 6 appears 2 times in the vector."

Hint: Use the following lines to enter the vector.

```
n <- readline(prompt="How many numbers do you want to enter: ")
 n <- as.integer(n)
# if (is.na(n)){n <- readnumber()}

 Numbers<-c()
 for (i in 1:n){
   num <- readline(prompt="Enter an integer: ")
   Numbers[i]<-as.numeric(num)
 }
```


Creating Tables in R

The table() function in R performs categorical tabulations of data with the variable and its frequency. Table() function is also helpful in creating frequency tables with condition and cross tabulations.

Let's use iris data set to create a frequency table for types of species of the iris flower. The iris data set is part of base R.

```
>## Frequency table with table() function in R
>head(iris)
>summary(iris)
>table(iris$Species)
```

Output

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
> summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width   
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100  
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300  
 Median :5.800   Median :3.000   Median :4.350   Median :1.300  
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199  
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800  
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500  
       Species  
 setosa    :50  
 versicolor:50  
 virginica :50  
```

The table shows there are 50 of each of the three types which is also verified by the summary() function.

```
> table(iris$Species)

    setosa versicolor  virginica 
        50         50         50 
```

Creating Tables in R

Frequency table with condition:
We can also create a frequency table with predefined condition using R table() function.

For example let's say we need to get how many observations have Sepal.Length>5.0 in iris table.

```
>## Frequency table with condition using table function in R
>table(iris$Sepal.Length>5.0)

FALSE   TRUE
   32    118
```

Notice the use of the "$" to access the field name Sepal.Length. This statement can be shortened to following is we use the **attach()** function.

```
>attach(iris)
>table(Sepal.Length>5.0)
```

The attach() function saves a lot unnecessary typing.

attach(), detach(), rm()

attach(filename**):** Will give you access to the fieldnames in the attached file without explicitly using the filename when accessing variables with the "$" operator. For example, you can write "mpg" instead of "mtcars$mpg".

detach(filename**):** Eliminate ability to access fieldnames without explicitly using the filename when accessing variable with the "$" operator. For example, you can only write "mtcars$mpg" instead of "mpg".

rm(filename**):** Removes file or other object from environment.

Creating Tables in R

Two-Way Cross Table in R:
Table function is also helpful in creating two-way cross tables in R. For example, let's create a cross tabulation of gears and carb using the mtcars data set.

```
>## cross tabulation gear * carb
>attach(mtcars)
>table(gear, carb)
```

```
      carb
gear 1 2 3 4 6 8
   3 3 4 3 5 0 0
   4 4 4 0 4 0 0
   5 0 2 0 1 1 1
```

Gears has values of 3, 4, and 5
Carb has the values of 1, 2, 3, 4, 6, 8
The interior of the table shows the counts.

Creating Tables in R

Three-Way Cross Table in R:
Similar to a two-way cross table, we can create a three-way cross table in R with the help of table function.

```
>## Three- way cross tabulation gear * carb* cyl with table function in R
>attach(mtcars)
>table(gear,carb,cyl)
```

Output:

A two-way table is created for each level of cyl. The table() function can host more than three levels.

```
, , cyl = 4

    carb
gear 1 2 3 4 6 8
   3 1 0 0 0 0 0
   4 4 4 0 0 0 0
   5 0 2 0 0 0 0

, , cyl = 6

    carb
gear 1 2 3 4 6 8
   3 2 0 0 0 0 0
   4 0 0 0 4 0 0
   5 0 0 0 0 1 0

, , cyl = 8

    carb
gear 1 2 3 4 6 8
   3 0 4 3 5 0 0
   4 0 0 0 0 0 0
   5 0 0 0 1 0 1
```

Creating Tables in R

Summary Tables Using tapply() (aka Table Apply)

The most important function in R for generating summary tables is the somewhat obscurely named **tapply** function. It is called **tapply** because it applies a named function (such as mean or variance) across specified margins (factor levels) to create a table.

It works like the PivotTable function in Excel

```
data<-read.table("c:\\temp\\Daphnia.txt",header=T)
attach(data)
names(data)

tapply(Growth.rate,Detergent,mean)

  BrandA   BrandB   BrandC   BrandD
3.884832 4.010044 3.954512 3.558231
```

This function calculates the mean growth rate (response variable) of the observations in each detergent (explanatory variable) group.

Creating Tables in R

Two-dimensional summary tables are created by replacing the single explanatory variable (the second argument in the function call) by a list of explanatory variables using the **list()** function. The **list()** function is used to indicate which variables are rows, columns, and indices of the summary table.

```
tapply(Growth.rate,list(Daphnia,Detergent),mean)

         BrandA   BrandB   BrandC   BrandD
Clone1 2.732227 2.929140 3.071335 2.626797
Clone2 3.919002 4.402931 4.772805 5.213745
Clone3 5.003268 4.698062 4.019397 2.834151

tapply(Growth.rate,list(Water,Daphnia),median)

         BrandA   BrandB   BrandC   BrandD
Clone1 2.705995 3.012495 3.073964 2.503468
Clone2 3.924411 4.282181 4.612801 5.416785
Clone3 5.057594 4.627812 4.040108 2.573003
```

It is important to note that any number of built-in or custom functions can be used as the last argument.

More than two explanatory variables can be used in the **list()** function,

Creating Tables in R

Build a table of the standard errors of the means

```
tapply(Growth.rate,list(Daphnia,Detergent), function(x) sqrt(var(x)/length(x)))

          BrandA    BrandB    BrandC    BrandD
Clone1 0.2163448 0.2319320 0.3055929 0.1905771
Clone2 0.4702855 0.3639819 0.5773096 0.5520220
Clone3 0.2688604 0.2683660 0.5395750 0.4260212
```


Creating Tables in R

Using `tapply()` to produce a three-dimensional table.

```
tapply(Growth.rate,list(Daphnia,Detergent,Water),mean)

, , Tyne

         BrandA   BrandB   BrandC   BrandD
Clone1 2.811265 2.775903 3.287529 2.597192
Clone2 3.307634 4.191188 3.620532 4.105651
Clone3 4.866524 4.766258 4.534902 3.365766

, , Wear

         BrandA   BrandB   BrandC   BrandD
Clone1 2.653189 3.082377 2.855142 2.656403
Clone2 4.530371 4.614673 5.925078 6.321838
Clone3 5.140011 4.629867 3.503892 2.302537
```

Creating Tables in R

The function **ftable()** (which stands for 'flat table') often produces more pleasing output:

```
               Tyne      Wear

Clone1 BrandA  2.811265 2.653189
       BrandB  2.775903 3.082377
       BrandC  3.287529 2.855142
       BrandD  2.597192 2.656403
Clone2 BrandA  3.307634 4.530371
       BrandB  4.191188 4.614673
       BrandC  3.620532 5.925078
       BrandD  4.105651 6.321838
Clone3 BrandA  4.866524 5.140011
       BrandB  4.766258 4.629867
       BrandC  4.534902 3.503892
       BrandD  3.365766 2.302537
```

Notice that the order of the rows, columns or tables is determined by the alphabetical sequence of the factor levels (e.g. Tyne comes before Wear in the alphabet).

Creating Tables in R

If you want to override this, you must specify that the factor levels are ordered in a non-standard way:

```
>water<-factor(Water,levels=c("Wear","Tyne"))
>ftable(tapply(Growth.rate,list(Daphnia,Detergent,water),mean))
```

```
               Wear      Tyne

Clone1 BrandA  2.653189 2.811265
       BrandB  3.082377 2.775903
       BrandC  2.855142 3.287529
       BrandD  2.656403 2.597192
Clone2 BrandA  4.530371 3.307634
       BrandB  4.614673 4.191188
       BrandC  5.925078 3.620532
       BrandD  6.321838 4.105651
Clone3 BrandA  5.140011 4.866524
       BrandB  4.629867 4.766258
       BrandC  3.503892 4.534902
       BrandD  2.302537 3.365766
```

Creating Tables in R

Homework:

Create a script that does the following:

1. Create a vector of 1000 random numbers from a Poisson distribution with lambda = 10 and then create a frequency table of the results.

2. Build tables to do the followjng with the iris data set: (Remember the "iris" data set is in base R)
    1. Calculate the average Sepal.Length by Species
    2. Calculate the average Sepal.Width by Species
    3. Calculate the average Pedal.Length by Species
    4. Calculate the average Pedal.Width by Species

3. Use the mtcars dataset to create the following tables: (Remember the "mtcars" data set is in base R)
    1. Median mpg by cyl and vs (V or straight engine).
    2. Median mpg by cyl, vs (V or straight engine), and gear.
    3. Median disp by cyl, vs (V or straight engine), gear, carb in a flat table.