R

Control Structures, Functions, and Objects

If Statements

- The if statement in R is very C like in it's syntax

```
if (condition) {
} else {
}
```

- Addition else if clauses have the syntax of

```
else if (condition) {
  }
```

In [ ]:

```
## This won't work, else needs to be on same line as end of if
if( 4 > 5) {
    print("Bad Math")
}
else {
    print("Seems right to me")
}
```

In [ ]:

```
if( 4 > 5) {
    print("Bad Math")
} else {
    print("Seems right to me")
}
```

In [ ]:

```r
a <- 4
b <- 2
if ( a %% b == 0)
{
    print(paste(a,"is even"))
} else {
     print(paste(a,"is odd"))
}
```

In [ ]:

```r
## Produces warning only, probably not the intended test condition
vec1 <- c(2,4,6,8)
if( vec1 %% 2 == 0){
    print("All elements are even")
} else{
    print("Not all elements are even")
}
```

In [ ]:

```r
vec1 <- c(2,4,6,8)
if( all(vec1 %% 2 == 0)){
    print("All elements are even")
} else{
    print("Not all elements are even")
}
```

If Tricks

- `if` on the right hand side of an assignment

```
var <- if (condition){
       value
       }
  else{
       value
  }
```

- The `ifelse` function to apply to vectors

```
ifelse(expression_generating_boolean_vector, value_if_true, value_if_fa
lse)
```

In [ ]:

```
num1 <- 100
num2 <- 1000
largest_num <- if(num1 > num2){
    num1
} else {
    num2
}
print(largest_num)
```

In [ ]:

```
float <- 100.004
truncated <- if (float %% 1 != 0){
    float %/% 1
} else{
    float
}
print(truncated)
```

In [ ]:

```
vec3 <- 1:10
print(ifelse(vec3 %% 3 == 0,
             "Is Divisible by 3",
             "Isn't Divisibly by 3" ))
```

In [ ]:

```
vec4 <- -5:5
print(ifelse(vec4 < 0,-1,1))
```

Switch Statement

- R doesn't have a switch statement, only a `switch` function

```
switch(expression, value1, value2, value3...)
switch(expression, key1 = value1, key2 = value2, default)
```

- The switch function takes an expression, followed by a list of things to return if matched
  - With out any parameter keywords, the expression needs to be an integer
  - When using keywords, a parameter with out a keyword is assumed to be a default value

In [ ]:

```
word <- switch(3,"one","two","three","four","five","six","seven")
print(word)

translation <- switch(word, one="uno",two="dos",
                      three="tres",four="quatro",
                      "un numero")
print(translation)

print(switch("seven", one="uno",two="dos",
             three="tres",four="quatro",
             "un numero"))
```

For Loops

- For loops in R look like for-each loops, but are still numeric
- The function `seq_along(X)` produces the sequence of indices for a given object to loop through

```
for(var in integer_vector){
  }
```

- Many libraries exist that attempt to produce better, faster for loops

In [ ]:

```
for(i in 1:5){
    print(i ^ 2)
}
```

In [ ]:
```
print(mtcars)
```

In [ ]:

```
## How can I make this print thet names of the column?
for (feature in seq_along(mtcars)){
    print(paste("The median is",
                median(mtcars[[feature]])))
}
```

Logic Controlled Loops

- R offers only one truly controlled logic loop, the standard `while` loop

```
while(condition){
  }
```

- R also provides a repeat loop, which repeats forever, and must be broken out of explicitly

```
repeat{
       if(condition) break
  }
```

In [ ]:

```
haystack <- c(1,34,5,5,1,4,6,0)
i <- 1
while(haystack[i] != 6){
        i <- i + 1
}
print(paste("I found 6 at position",i))
```

In [ ]:

```
end <- 1
repeat{
    print("This is the song that never ends, yes it goes on and on my friends,
           some people started singing it not knowing what it was, and they will
 keep on
           singing it forever just because...")
    if (end == 10) break
    end <- end + 1
}
```

Lapply

- Often times we are just looping over a data structure in R to apply a function to every member
  - The R function `lapply` does this without writing out the entire loop
- This is the first of many functional programming style statements we will encounter in R
  - Entire libraries have been created to further this style of programming

```
lapply(data,function)
```

In [ ]:

```r
## What is the return type do you think?
results_l <- lapply(mtcars,median)
print(results_l)
```

In [ ]:

```r
## What is the return type do you think?
results_s <- sapply(mtcars,median)
print(results_s)
```

Functions

- A function in R is declared using the syntax

```
function(parameter list){
  function body
  }
```

- The result of this is assigned to a variable which is then used as the function name

In [ ]:

```r
my_first_function <- function(){
    print("Hello!")
}

my_first_function()
```

In [ ]:

```
my_second_function <- function(a,b,c){
    print(a * b + c)
}
my_second_function(1,2,3)
```

Returning From Functions

- To explicitly return from an R function, use the `return` function
  - Note that this is a function, not a statement, and requires parentheses

```
return (x)
```

- If no return function is called, an R function will return the value of the last expression of the function by default

In [ ]:

```
explicit_return <- function(a,b)
    {
    return (a %/% b + a %% b)
}
print(explicit_return(20,3))
```

In [ ]:

```
implicit_return <- function(a,b)
{
a %/% b + a %% b
}
print(implicit_return(20,3))
```

Function Practice

- Use lapply and a function to return the squares of all numbers from 1 to 25

Function Parameters and Arguments

- R provides a wide variety of parameter options
  - Keyword parameters
  - Default parameters
  - Positional parameters
- R also allows a list to provide the arguments to a function, using the `do.call` function

```
do.call(function_name, list_of_arguments)
```