# Introduction to R, Part III

## Intro to Programming in R

Tim Frasier

# Loops

# Loops

- Often we use computers to perform the same analyses when we have to do them multiple times, or across multiple "samples"

- Can automate this process with loops
  - "for" loops
  - "while" loops

- Repeat specified commands as long as conditions are met

# Loops

## `for` loops

```
for (i in 1:10) {

    cat(sprintf("i is %d\n", i)

}
```

# Loops

## for loops

```
for (i in 1:10) {
    cat(sprintf("i is %d\n", i)
}
```

Variable that will change throughout the loop (can be anything you want)

What values it will "step" through

# Loops

## for loops

```
for (i in 1:10) {
    cat(sprintf("i is %d\n", i)
}
```

Commands to conduct for each "step" through the loop

# Loops

## `for` loops

```
for (i in 1:10) {
    cat(sprintf("i is %d\n", i)
}
```

A version of the `print` function for use when you don't know what the value will be (used with wildcards)

# Loops

## `for` loops

```
for (i in 1:10) {
    cat(sprintf("i is %d\n", i)
}
```

What value to use for the wildcard

Wildcard

A version of the `print` function for use when you don't know what the value will be (used with wildcards)

# Loops

## `for` loops

| Wildcard | Data Type |
|---|---|
| %d | integer |
| %s | text/character |
| %f | floating-point number |

# Loops

## `for` loops

```
for (i in 1:10) {
    cat(sprintf("i is %d\n", i)
}
```

Needed for `sprintf` function to print properly

Indicates to print a "new line"

# Loops

## `for` loops

```
for (i in 1:10) {

    cat(sprintf("i is %d\n", i)

}
```

What will this function do, and
what will be the output?

# Loops

## `for` loops

```
for (i in 1:10) {

    cat(sprintf("i is %d\n", i)

}
```

```
i is 1
i is 2
i is 3
i is 4
i is 5
i is 6
i is 7
i is 8
i is 9
i is 10
```

# Loops

## `for` loops

```
for (i in 1:10) {

    j = (i * 2) / 3
    cat(sprintf("i is %d, j is %f\n", i, j))

}
```

What will this function do, and
what will be the output?

# Loops

## `for` loops

```
for (i in 1:10) {

    j = (i * 2) / 3
    cat(sprintf("i is %d, j is %f\n", i, j))

}

i is 1, j is 0.666667
i is 2, j is 1.333333
i is 3, j is 2.000000
i is 4, j is 2.666667
i is 5, j is 3.333333
i is 6, j is 4.000000
i is 7, j is 4.666667
i is 8, j is 5.333333
i is 9, j is 6.000000
i is 10, j is 6.666667
```

# Loops

## `while` loops

- `for` loops step through commands a pre-defined number of times

- `while` loops step through commands as long as a defined condition is met

# Loops

## `while` loops

```
i = 1
while (i <= 10) {

    cat(sprintf("i is %d\n", i))
    i = i + 1

}
```

# Loops

## `while` loops

```
i = 1
while (i <= 10) {

    cat(sprintf("i is %d\n", i))
    i = i + 1

}
```

Need something like this, otherwise will loop forever

# Loops

- Can usually accomplish the same thing using either a `for` or `while` loop

- I prefer `for` loops because it is less easy to make them go on indefinitely

# Decisions
# (if statements)

# Decisions

- Can use `if…else` statements to make decisions regarding what to do with certain types of data

# Decisions

- Create a vector from 1 to 10

```
myVec = 1:10
```

- Have it go through list, and call any value <= 5 “small”, and all others “large”

```
for (i in 1:length(myVec)) {

    if (myVec[i] <= 5) {
        print(“small”)
    } else {
        print(“large”)
    }
}
```

# Your Turn!

# Decisions

- Write commands that will loop through the `afdata` data frame and store all of the fat percentage data for males in one vector and those for females in another vector

# Decisions

```
males = 0
females = 0

malecounter = 1
femalecounter = 1

for (i in 1:length(afdata[, 1])) {

    if (afdata[i, 4] == "male") {
        males[malecounter] = afdata[i, 3]
        malecounter = malecounter + 1

    } else {
        females[femalecounter] = afdata[i, 3]
        femalecounter = femalecounter + 1
    }
}
```

# Good Programming Practices

# Good Programming Practices

## 1. Extensive use of comments

- R will ignore lines starting with #
  - Can use to make notes about what is happening
- Code should have enough comments that someone could read them and understand what is going on without any explanation from you
  - Code should speak for itself (and to you)

```r
#-------------------------#
# Function - compareestimators
#-------------------------#
# This function analyses simulated data with all moment-based relatedness
# estimators and then plots the data in way where estimates can be easily
# compared.
# It takes 2 arguments:
# 1. A data frame of genotype data
# 2. The number of individuals to simulate
#-----------------------------------------------------------#

compareestimators <- function(filename, ninds) {
    #-----------------------------------------------#
    # Generate simulated data
    #-----------------------------------------------#
    simdata <- familysim(filename$freqs, ninds)
    output <- coancestry(simdata, lynchli=1, lynchrd=1, quellergt=1, wang=1)
    simrel <- cleanuprvals(output$relatedness, ninds)

    ##############################################################
    # Calculate Correlation Coefficient Between Estimators        #
    ##############################################################

    #-------------------------------------------------------------#
    # Create reference values for comparison
    #-------------------------------------------------------------#
    urval <- rep(0, ninds)
    hsval <- rep(0.25, ninds)
    fsval <- rep(0.5, ninds)
    poval <- rep(0.5, ninds)
    relvals <- c(poval, fsval, hsval, urval)

    #-------------------------------------------------------------#
    # Calculate Correlation Coefficients
    #-------------------------------------------------------------#
    wangcor <- cor(relvals, simrel[, 6])
    lynchlicor <- cor(relvals, simrel[, 7])
    lynchrdcor <- cor(relvals, simrel[, 8])
    quellergtcor <- cor(relvals, simrel[, 10])

    #-------------------------------------------------------------#
    # Print These To The Screen
    #-------------------------------------------------------------#
    cat(sprintf("\nCorrelation Coefficients Between Observed & Expected Values:\n"))
    cat(sprintf("wang\t\t%f\n", wangcor))
    cat(sprintf("lynchli\t\t%f\n", lynchlicor))
    cat(sprintf("lynchrd\t\t%f\n", lynchrdcor))
    cat(sprintf("quellergt\t%f\n", quellergtcor))
```

```r
#---Generate Counter for Rows---#
rowcount <- 1

#---Loop through lines---#
for (i in 2:nlines) {

    #---Check if this is a new individual---#
    if (captures[i, 1] == captures[(i-1), 1]) {

        # If so, record new sighting in same row as above

        # Loop through periods
        for (j in 1:nperiods) {

            # Enter a 1 for periods were individual
            # was seen, ignore others so as to
            # not overwrite previous sightings with 0s
            # (1s are okay)
            if (captures[i, 3] == periods[j]) {
                caphist[rowcount, j] <- 1
            }
            if (captures[i, 6] == periods[j]) {
                caphist[rowcount, j] <- 1
            }
        }
            } else {

                # If not, increase rowcount and record
                # sighting in new row
                rowcount <- rowcount + 1

                # Loop through periods
                for (j in 1:nperiods) {

                    # Enter a 1 for period where individual was
                    # sighted, zero elsewhere
                    ifelse(captures[i, 3] == periods[j], caphist[rowcount, j] <- 1, caphist[rowcount, j] <-
                        0)
                    if (captures[i, 6] == periods[j]) {
                        caphist[rowcount, j] <- 1
                    }
                }
            }
}
```

# Good Programming Practices

## 2. Use of tabs

- Should indent nested processes with a tab
  - Anything wrapped in brackets ({ }) should be indented with a tab at the "next level"
  - Functions
  - `for` and `while` loops
  - `if…else` statements
- Makes it clear what is contained within each process, and when each ends

# All within same function (not shown)

```
#---Generate Counter for Rows---#
rowcount <- 1

#---Loop through lines---#
for (i in 2:nlines) {

    #---Check if this is a new individual---#
    if (captures[i, 1] == captures[(i-1), 1]) {

        # If so, record new sighting in same row as above

        # Loop through periods
        for (j in 1:nperiods) {

            # Enter a 1 for periods were individual
            # was seen, ignore others so as to
            # not overwrite previous sightings with 0s
            # (1s are okay)
            if (captures[i, 3] == periods[j]) {
                caphist[rowcount, j] <- 1
            }
            if (captures[i, 6] == periods[j]) {
                caphist[rowcount, j] <- 1
            }
        }
            } else {

                # If not, increase rowcount and record
                # sighting in new row
                rowcount <- rowcount + 1

                # Loop through periods
                for (j in 1:nperiods) {

                    # Enter a 1 for period where individual was
                    # sighted, zero elsewhere
                    ifelse(captures[i, 3] == periods[j], caphist[rowcount, j] <- 1, caphist[rowcount, j] <-
                        0)
                    if (captures[i, 6] == periods[j]) {
                        caphist[rowcount, j] <- 1
                    }
                }
            }
}
```

# Within this `for` loop

```
#---Generate Counter for Rows---#
rowcount <- 1

#---Loop through lines---#
for (i in 2:nlines) {

    #---Check if this is a new individual---#
    if (captures[i, 1] == captures[(i-1), 1]) {

        # If so, record new sighting in same row as above

        # Loop through periods
        for (j in 1:nperiods) {

            # Enter a 1 for periods were individual
            # was seen, ignore others so as to
            # not overwrite previous sightings with 0s
            # (1s are okay)
            if (captures[i, 3] == periods[j]) {
                caphist[rowcount, j] <- 1
            }
            if (captures[i, 6] == periods[j]) {
                caphist[rowcount, j] <- 1
            }
        }
            } else {

                # If not, increase rowcount and record
                # sighting in new row
                rowcount <- rowcount + 1

                # Loop through periods
                for (j in 1:nperiods) {

                    # Enter a 1 for period where individual was
                    # sighted, zero elsewhere
                    ifelse(captures[i, 3] == periods[j], caphist[rowcount, j] <- 1, caphist[rowcount, j] <-
                        0)
                    if (captures[i, 6] == periods[j]) {
                        caphist[rowcount, j] <- 1
                    }
                }
            }
}
```

# Within this `if` statment…

```
#---Generate Counter for Rows---#
rowcount <- 1

#---Loop through lines---#
for (i in 2:nlines) {

    #---Check if this is a new individual---#
    if (captures[i, 1] == captures[(i-1), 1]) {

        # If so, record new sighting in same row as above

        # Loop through periods
        for (j in 1:nperiods) {

            # Enter a 1 for periods were individual
            # was seen, ignore others so as to
            # not overwrite previous sightings with 0s
            # (1s are okay)
            if (captures[i, 3] == periods[j]) {
                caphist[rowcount, j] <- 1
            }
            if (captures[i, 6] == periods[j]) {
                caphist[rowcount, j] <- 1
            }
        }
            } else {

                # If not, increase rowcount and record
                # sighting in new row
                rowcount <- rowcount + 1

                # Loop through periods
                for (j in 1:nperiods) {

                    # Enter a 1 for period where individual was
                    # sighted, zero elsewhere
                    ifelse(captures[i, 3] == periods[j], caphist[rowcount, j] <- 1, caphist[rowcount, j] <-
                        0)
                    if (captures[i, 6] == periods[j]) {
                        caphist[rowcount, j] <- 1
                    }
                }
            }
}
```

# Good Programming Practices

## 3. Use of horizontal space

- R ignores blank horizontal space
- Use horizontal spacing to make code easy to read, and to make it clear what commands are closely associated and which are not

```
#-----------------------------#
# Specify the model for JAGS   #
#-----------------------------#
modelstring = "
model {
    for (i in 1:M) {

        #Likelihood

        # Omega is an indicator for whether a row in the
        # augmented matrix contains real data or not,
        # and is modelled with a Bernoulli distribution.
        # Omega will be estimated from the data.
        z[i] ~ dbern(omega)

        # Calculate likelihood for each time period
        for (j in 1:T) {
            yaug[i, j] ~ dbern(p.eff[i, j])
            p.eff[i, j] <- z[i] * p     # Can only be detected if z=1
        } #j
    } #i

    # Priors
    omega ~ dunif(0, 1)
    p ~ dunif(0, 1)

    # Derived quantities
    N <- sum(z[])
}
"
writeLines(modelstring,con="model.txt")
```

# Good Programming Practices

## 4. Use of vertical space

- R ignores blank vertical space
- Use vertical spacing to make code easy to read

# Good Programming Practices

## 4. Use of vertical space

- Single space around qualifiers or modifiers…
  - <-
  - =
  - etc.
- But not after parentheses or brackets

# Good Programming Practices

## 4. Use of vertical space

This

```
list2 = sample(list1, size = 20, replace = TRUE)
```

Not this

```
list2=sample(list1, size=20, replace=TRUE)
```

# Good Programming Practices

## 4. Use of vertical space

- Put a space after (but not before) commas

This

```
list2 = sample(list1, size = 20, replace = TRUE)
```

Not this

```
list2 = sample(list1,size = 20,replace = TRUE)
```

# Good Programming Practices

- Clean, clear code is a beautiful thing
- Makes your programs more usable and reproducible for yourself and others
- Good practice, like keeping a neat lab book

# Questions?