Programming in R

- We will learn how to program using scripts and user-defined functions.
- **Scripts** are simply a collection of commands saved in a text file.
  - Make sure you use a non-formatted text editor, not a word processor!
  - Save the script in your working directory
  - Run the script using the `source()` command
- **Functions** are objects in R's workspace (just like variables)
  - We can define functions to do specific tasks.
  - Note: we need to get functions into R's workspace, by `source`-ing them once (or entering at the command line).

Getting started

An R program is a list of instructions that change the program state (i.e. the values of variables)

- If we perform a useful series of commands at the command line, and think we'll want to perform them again, then we can collect them into a script.

```
xx <- seq(0,100,by=0.1)
yy <- 3*xx^2 - xx + 2
plot(xx,yy, type='l', col='red', main='My plot')
```

Flow control: conditional execution

- Sometimes we only want to execute part of our program under certain conditions.

- Use an **if statement**
  - express your condition as a logical expression

```
xx > 10                    xx == 2
xx >= yy                   length(xx) != 0
xx < 2 | xx > 20
```

  - then you can set a block of code to be executed only **if** that condition is fulfilled

Flow control: conditional execution

Syntax in R

```
if (logical_expression) {
      expression_1
      ...
}
```

or for two options

```
if (logical_expression) {
      expression_1
      ...
} else {
      expression_2
      ...
}
```

Flow control: conditional execution

Example

```
# a script to say how a geometric growth model will act.

RR <- 1.2          # value of annual growth rate

if (RR > 1) {
      ss <- 'My population will grow.'
} else {
      ss <- 'My population will shrink or stay the same'
}

ss
```

Flow control: looping

- Often there will be tasks that you want to repeat many times.
- Can accomplish this using a **for loop** or a **while loop**.

- **for loops**
  - Use to loop a fixed number of times.
  - Define an index variable and a set of values it should take.
  - Go through the loop once for each value.

- **while loops** *(can mostly ignore these for now…)*
  - Use to loop until some condition isn't met anymore.
  - Define the logical condition that needs to be satisfied.
  - Go through the loop until the condition isn't satisfied any more.

Flow control: looping

Syntax in R -- for loops

```
for (xx in 1:100) {
      expression_1
      ...
}
```

or more generally

```
for (xx in someVector) {
      expression_1
      ...
}
```

Syntax in R -- while loops

```
while (logical_expression) {
      expression_1
      ...
}
```

Flow control: looping

Example

- Two ways of doing an operation 10 times

With a **for loop**

```
nn <- 10
RR <- 1.1

for (tt in 1:10) {
      nn[tt+1] <- RR*nn[tt]
}
```

With a **while loop**

```
nn <- 10
RR <- 1.1

tt <- 1

while (tt <= 10) {
      nn[tt+1] <- RR*nn[tt]
      tt <- tt+1
}
```

Programming style

- Be clear rather than clever.
  - You'll be glad when you have to come back to a program you wrote two years ago…. to analyze new data or modify your analysis.

- Use meaningful variable names
  - birthRate and probSurvival instead of b and p

- When using simple variable names (x, y, i, j, etc), it's better to use double letters (xx, yy) since they're easier to find (and replace) using automated text search functions.

Programming style

- Comments! `# you can never have too many`

- Put a line or two of comments at the top of your program to say what it does.
  - Use 'sub-headings' to mark sections of code as well

- White space
  - leave blanks lines to separate blocks of code
  - use indentation consistently to indicate lines of code within loops or if statements
  - R doesn't insist on this like some languages, but it will help you keep your thinking straight, keep your parentheses matched up, and avoid bugs.

Pseudo-code

- **Pseudo-code** is an informal way to plan out the structure and logic of your programs.

- Pseudo-code doesn't worry about the syntax of the programming language you're using, but it does pay attention to the variables and flow control structures you will use.

- Because the basic structures of many high-level programming languages are the same, this means you can write down a plan for your program in a way that would translate easily to R, Matlab, python, or a range of other languages.

Pseudo-code example

Goal: Determine how many students pass the bootcamp.

```
# set the number of passes to zero
# set the number of fails to zero

# loop over the number of students in the class
  # if the student completed their exercises and committed them
     # add one to the number of passes
  # or else if they didn't complete and commit their exercises
     # add one to the number of fails
     # print "Come on, this is the first thing we've asked you do in grad
                    school!"

# show the result
```

Pseudo-code exercise

Calculate the sum of all numbers from 1 to 20

Pseudo-code exercise

The geometric growth model is the simplest model for population growth in discrete time. It assumes that every year the size of the population changes by the same factor, R.

$$N(t+1) = R \times N(t)$$

Exercise: write pseudo-code for a program that simulates the growth of a population for 10 years, starting with N=100 animals and assuming R = 1.05, and prints the final population size.

Bonus: modify the pseudo-code so the program will make a plot of N versus t.

General layout of modeling scripts

1. Setup statements, if needed (e.g. loading packages)

2. Input data, set parameter values, and/or set initial conditions

3. Perform the calculations

4. Display the results by plotting, saving, or showing on-screen.

Obviously there will be many deviations from this pattern, but the broad framework is useful to keep in mind.

One useful habit is to lay out the program in pseudocode, then use the pseudocode as commented headings in your real code.

A program for the geometric model

```
# geometricGrowthScript.R
# a script to simulate and plot the discrete logistic model

# Setup
# none needed, since the program is so simple

# Set initial conditions and parameter values
N0 <- 100
RR <- 1.05
ttMax <- 10

# initialize variable to a vector of NA values
NN <- matrix(NA, nrow=1, ncol=ttMax+1)
NN[1] <- N0   # set first value to initial condition

# use a loop to iterate the model the desired number of times
for (tt in 1:ttMax) {
  NN[tt+1] <- RR*NN[tt]
}

# plot the results
plot(1:(ttMax+1), NN, xlab="time", ylab="N", col='blue')
```

Programming your own functions

Writing your own functions is the most powerful way to make R work flexibly and efficiently for you.

→ User-defined functions enable you to reduce long stretches of complicated programming to a single command.

A function takes in arguments, performs some operations, and returns a value.

To define a function, you need to

- choose a name for the function (avoid reserved words)
- define the arguments that need to be passed to the function
- define the operations
- set what value the function returns

Programming your own functions

R syntax

```
name <- function(argument_1, argument_2, ...) {
      expression_1
      expression_2
      ...
      return(output)
}
```

Example: compute the mean grade in a class, with possible inflation

```
myMeanGrade <- function(rawGrades, inflate) {
      sumGrades <- sum(rawGrades)
      nStudents <- length(rawGrades)
      trueMeanGrade <- sumGrades/nStudents
      return(trueMeanGrade * inflate)
}
```

Inputs: default values for arguments

You can assign default values to function arguments by giving them a value in the function definition.

```
myMeanGrade <- function(rawGrades, inflate = 1) {
      sumGrades <- sum(rawGrades)
      nStudents <- length(rawGrades)
      trueMeanGrade <- sumGrades/nStudents
      return(trueMeanGrade * inflate)
}

> myMeanGrade(c(60,70,80,90))
[1] 75
> myMeanGrade(c(60,70,80,90), 1.1)
[1] 82.5
```

Inputs: optional arguments

Any argument for which you've given a default value is optional when you call the function.

If you pass fewer values than the function has arguments, then R will assign them from left to right:

```
> test3 <- function(x = 1, y = 1, z = 1) {
+     return(x * 100 + y * 10 + z)
+ }
> test3(2, 2)
[1] 221
```

Unless you name the arguments as you pass values:

```
> test3(y = 2, z = 2)
[1] 122
```

Outputs: the value that is returned

The function will continue evaluating until:

- it hits the first `return()` statement, after which it returns the requested value and quits.
  - note that there can be multiple `return()` statements in a function, e.g. in a branching program structure

- it runs out of commands to run, in which case it returns the value of the last unassigned expression and quits.
  - sometimes the returned value isn't the point of the function, e.g. in a function whose purpose is to make a plot.

The function workspace and variable scope

- When a function is executed, R sets aside memory for a separate workspace for the function to operate.

- The function can define new variables and conduct all its operations in complete isolation from the main workspace.

- When the operations are complete, the function returns the appropriate quantity and all other information in the function workspace is lost.

Example: converting a model script to a function

By converting a script to a function, you can make it quick and easy to run the model with different parameter values or initial conditions.

The conversion process is easy – for example:

```
# Simple model script
# Define parameter values and ICs
RR <- 1.05
N0 <- 100
ttMax <- 10

# Initialize vector to hold output
NN <- rep(NA,ttMax+1)
NN[1] <- N0

# Use a for loop to step forward
for (tt in 1:ttMax){
    NN[tt+1] <- RR*NN[tt]
}

plot(1:(ttMax+1),NN,lty=2,type='l',...
    xlab='t', ylab='Population size')
```

Basic change: just moved parameters, ICs from body of script to function arguments

```
# Function version

geomFun <- function(RR, N0, ttmax){
    # Initialize vector to hold output
    NN <- rep(NA,ttMax+1)
    NN[1] <- N0

    # Use a for loop to step forward
    for (tt in 1:ttMax){
        NN[tt+1] <- RR*NN[tt]
    }

    plot(1:(ttMax+1),NN,lty=2,type='l',
        ... xlab='t', ylab='Population
        size')
}
```

Example: converting a model script to a function

Now you can run the full model, for any parameter values and initial conditions, using a single statement (by *calling* the function).

```
geomFun(RR=1.01, n0=200, ttmax=20)
```

This enables you to do fancier things, such as running the model many times to analyze the sensitivity to different parameter values.

Pseudocode for a sensitivity analysis

Define parameter values; use a vector to hold a range of values for the parameter(s) you wish to vary.

Initialize a matrix to collect all outputs

Use a **for loop** to repeatedly run the model and collect output

    # NOTE: this for loop is not over the timesteps of the model, it is over the set of
    # different parameter values for which you want to run the model.
    # You can use your function to run the model in a single line within the loop.

analyze results

Packages

Assembled collections of R functions, made available by other users

e.g. deSolve, emdbook, geiger

1. Choose CRAN mirror ('USA (CA 2)' is UCLA)
2. Install package(s)
3. In R session, need to enter

`> library(`*`package_name`*`)`

Then can use the functions in that package.

In subsequent uses, just do step 3.

Building an ODE model

- install package 'deSolve'
- load the package into memory using `library(deSolve)`

We will use a function called lsoda that is a versatile and robust numerical integrator for ordinary differential equations.

- powerful tool but you need to interact with it in a very specific way -- be careful with syntax!

Syntax for `lsoda`

Generic syntax for `lsoda` is:

```
output <- lsoda(init, tseq, ODEfunction, pars)
```

where:

`init` is the initial value of the state variable

`tseq` is a vector of the time points where the model will be evaluated

`ODEfunction` is a place-holder for the name of the function holding the model equations

`pars` is a vector containing any parameters used in the model

`output` is the variable where `lsoda` will return its results.

Syntax for `lsoda`

The function holding the model equations must have syntax:

```
ODEfunction <- function(tt, yy, pars) {
   derivs <- [insert model equations]
   return(list(derivs))
}
```

where:

`tt` is a variable used by R to keep track of the timestep

`yy` is the state variable (or vector of state variables, for multi-variate models)

`pars` is your vector of model parameters

`derivs` is an internal variable that records the time series of results

`lsoda` example: exponential growth

$$\frac{dN}{dt} = rN$$

```
expGrowthODE <- function(tt, NN, pars) {
   derivs <- pars['rr'] * NN
   return(list(derivs))
}
init <- 1
tseq <- seq(0, 20, by=0.01)
pars <- c(rr = 0.1)
```

Note new way of indexing a vector.
Access with `pars['rr']`
Don't need to use this method, but it's useful when you have lots of parameters.

Then call with:

```
expOutput <- lsoda( init, tseq, expGrowthODE, pars)
```

Output from `lsoda`

The output from our command:

```
expOutput <- lsoda( init, tseq, expGrowthODE, pars)
```

will be a matrix made up of two column vectors.

The first column will be the time points where the state of the system is recorded, and the second column will be the corresponding values of the state variable.

```
> expOutput
      time        1
 [1,] 0.00 1.000000
 [2,] 0.01 1.001001
 [3,] 0.02 1.002002
 [4,] 0.03 1.003005
...
```

So we can plot the dynamics with:

```
plot(expOutput[,1], expOutput[,2], col='blue', type='l')
```

`lsoda` example: logistic growth

$$\frac{dN}{dt} = rN\left(1 - \frac{N}{K}\right)$$

```
logisticGrowthODE <- function(tt, NN, pars) {
   derivs <- pars['rr'] * NN * (1 - NN/pars['KK'])
   return(list(derivs))
}
init <- 1
tseq <- seq(0, 20, by=0.01)
pars <- c(rr = 0.1, KK = 100)
```

Then call with:

```
logisticOutput <- lsoda( init, tseq, expGrowthODE, pars)
```