Session 2

Programming in R

Matt Denwood

2023-05-31

Recap

Material

Everything is on the public GitHub repo:

1. Use GitHub Desktop (https://desktop.github.com)
2. Choose “File” and “Clone Repository” and the “GitHub.com” tab
3. Enter ku-awdc/R6ModellingCourse2023 and click on Clone
4. Remember to pull changes before each day, as things will be added
5. Remember to copy/paste code from the HTML version, NOT the PDF version

Material

Everything is on the public GitHub repo:

1. Use GitHub Desktop (https://desktop.github.com)
2. Choose “File” and “Clone Repository” and the “GitHub.com” tab
3. Enter ku-awdc/R6ModellingCourse2023 and click on Clone
4. Remember to pull changes before each day, as things will be added
5. Remember to copy/paste code from the HTML version, NOT the PDF version

You only have write access, so you can’t push changes. It is best not to modify anything in the course files (notes etc) in case you get conflicts pulling changes later...

R packages

You should create an R package for (almost) every project.

1. Create the repository on github.com (with a README)
2. Use the neatpkg package:

```
install.packages("neatpkg",
↪  repos=c("https://cran.rstudio.com/","https://ku-awdc.github.io/drat/"))

neatpkg::pkg_new("name")
```

3. Push changes regularly

Variable names

- Use snake_case for functions and variables
  - Function arguments count as variables
  - R has first-class functions, so functions (and methods) are also variables...
  - Don't re-use the name of the package as a function name

Variable names

- Use snake_case for functions and variables
  - Function arguments count as variables
  - R has first-class functions, so functions (and methods) are also variables...
  - Don't re-use the name of the package as a function name
- Use PascalCase for:
  - Class names (S3, R6, and Rcpp)
  - Columns within data frames (to avoid clashes with variables)
  - Probably also element names within lists (because data frames are lists, and we like consistency)

Variable names

- Use snake_case for functions and variables
  - Function arguments count as variables
  - R has first-class functions, so functions (and methods) are also variables...
  - Don't re-use the name of the package as a function name
- Use PascalCase for:
  - Class names (S3, R6, and Rcpp)
  - Columns within data frames (to avoid clashes with variables)
  - Probably also element names within lists (because data frames are lists, and we like consistency)
- Use dot.separated names if:
  - You deliberately want to create confusion for S3 method dispatch, for some reason
  - You want to make it clear that I have not contributed in any way to your code base

Debugging an R package

Debugging functions (and R6 methods) in an R package is a little different to debugging a normal script.

1. Be a defensive programmer

- Always use curlies {} around if() and else if() etc
- Use stopifnot() and stop() *liberally*

Debugging an R package

Debugging functions (and R6 methods) in an R package is a little different to debugging a normal script.

1. Be a defensive programmer

- Always use curlies {} around if() and else if() etc
- Use stopifnot() and stop() *liberally*

2. Use a step-through debugger:

- Place a call to browser() in a specific location and re-install
- Use debug(foo) to step through foo (set undebug(foo) afterwards)
- See also ?traceback ?dump.frames ?trace

3. To recover any future error use:

- Interactive sessions:
  - options(error = utils::recover)
- Non-interactive sessions:
  - options(error = quote({dump.frames(to.file = TRUE); q(status = 1)}))

3. To recover any future error use:

- Interactive sessions:
  - options(error = utils::recover)
- Non-interactive sessions:
  - options(error = quote({dump.frames(to.file = TRUE); q(status = 1)}))

See also: https://adv-r.hadley.nz/debugging.html

Current status

You should all have:

- GitHub Desktop installed and logged in to your github.com account
- The neatpkg package re-installed (there have been changes!):

```
install.packages("neatpkg",
↪  repos=c("https://cran.rstudio.com/","https://ku-awdc.github.io/drat/"))
```

- The basic structure for your own R package

Programming paradigms

Imperative / procedural

The oldest programming paradigm, used by machine code, Fortran, BASIC, ALGOL etc

- Imperative:
  - One big script, code run from top to bottom
- Procedural:
  - Code is broken up into functions/procedures/subroutines

Imperative / procedural

The oldest programming paradigm, used by machine code, Fortran, BASIC, ALGOL etc

- Imperative:
  - One big script, code run from top to bottom
- Procedural:
  - Code is broken up into functions/procedures/subroutines

Common in C-like languages such as R.

Imperative (stochastic, discrete-time) SIR model:

```
iters <- 10L; time <- 100L
beta <- 0.01; gamma <- 0.1
start <- list(S=99L, I=1L, R=0L)

output <- as.list(1L:iters)
for(i in seq_along(output)){
  output[[i]] <- as.data.frame(c(list(Iter=i, Time=1L:time), start))

  for(t in 2L:time){
    S <- output[[i]][["S"]][t-1]
    I <- output[[i]][["I"]][t-1]
    R <- output[[i]][["R"]][t-1]

    nI <- rbinom(1L, S, 1.0 - exp(-beta * I))
    nR <- rbinom(1L, I, 1.0 - exp(-gamma))

    output[[i]][t,c("S","I","R")] <- c(S-nI, I+nI-nR, R+nR)
  }
}
```

Procedural version:

```
#' In sir_mod.R within your package, and with documentation!
sir_mod <- function(iter, beta=0.01, gamma=0.1, time=100L,
                    start = list(S=99L, I=1L, R=0L)){
  output <- as.data.frame(c(list(Iter=iter, Time=1L:time), start))
  for(t in 2L:time){
    S <- output[["S"]][t-1]
    I <- output[["I"]][t-1]
    R <- output[["R"]][t-1]

    nI <- rbinom(1L, S, 1.0 - exp(-beta * I))
    nR <- rbinom(1L, I, 1.0 - exp(-gamma))

    output[t,c("S","I","R")] <- c(S-nI, I+nI-nR, R+nR)
  }
  return(output)
}

## In a file somewhere under notebooks:
output <- as.list(1L:10L)
for(i in seq_along(output)){
  output[[i]] <- sir_mod(i)
}
```

Benefits of using functions:

- Encapsulation
  - There is no ‘bleed-out’ of temporary variables
  - The sir_mod function can be documented, and re-used
  - The sir_mod function can be tested

Benefits of using functions:

- Encapsulation
  - There is no ‘bleed-out’ of temporary variables
  - The sir_mod function can be documented, and re-used
  - The sir_mod function can be tested
- Abstraction
  - Separation of implementation from usage makes both clearer
  - Future changes to sir_mod do not break other code

Functional

- A relatively modern paradigm, used heavily by Lisp, Haskell, F# etc

Features: - Functions are ‘first class citizens’ i.e. are treated as variables - Avoid using temporary variables - No for/repeat/while loops (recursion may be used) - Functions are chained together

Functional

- A relatively modern paradigm, used heavily by Lisp, Haskell, F# etc

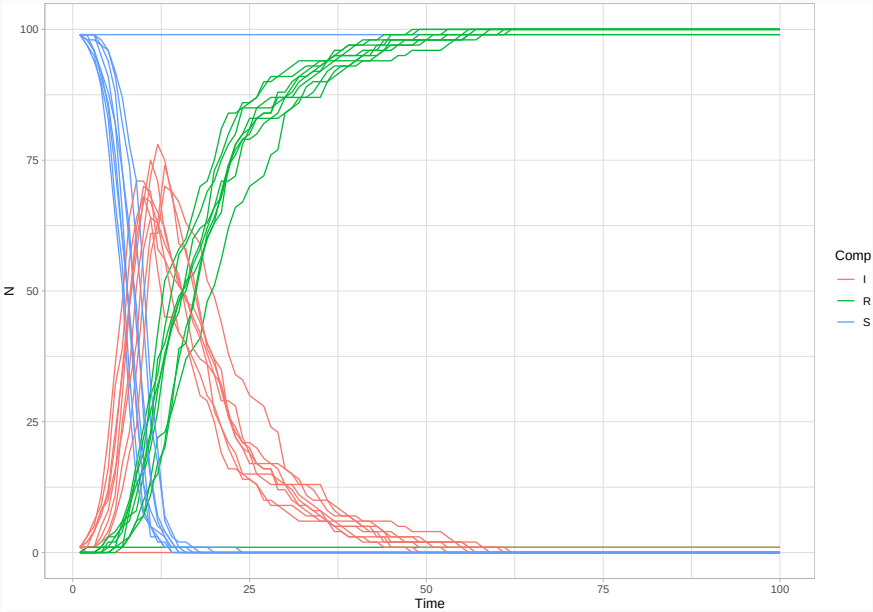
Features: - Functions are ‘first class citizens’ i.e. are treated as variables - Avoid using temporary variables - No for/repeat/while loops (recursion may be used) - Functions are chained together

At it’s core, R is actually a functional programming language. . . !

Most obvious with tidyverse / dplyr / chaining.

A functional version:

```
1L:iters |>
  as.list() |>
  pblapply(sir_mod) |>
  bind_rows() |>
  pivot_longer(cols = c("S","I","R"), names_to="Comp", values_to="N") |>
  ggplot(aes(x=Time, y=N, col=Comp, group=interaction(Comp, Iter))) +
  geom_line() +
  theme_light() ->
  plot_out
```


100
75
50
25
0
N
0
25
50
75
100
Time
Comp
I
R
S

Object-oriented

Originated with languages such as Simula, Smalltalk, C++ etc

1. Functional OO in R: S3, S4, R7
2. Encapsulated OO in R: ReferenceClasses, R6

Object-oriented

Originated with languages such as Simula, Smalltalk, C++ etc

1. Functional OO in R: S3, S4, R7
2. Encapsulated OO in R: ReferenceClasses, R6

Key features of encapsulated OO:

- Classes form the core building blocks of code
- Data and functions (aka methods) are encapsulated within the object
- Structured (and typically strict) separation of implementation and usage
- Inheritance allows derived classes to extend parent classes
  - Run-time and/or compile-time polymorphism

Simple example in R6:

```
library("R6")

SimpleSIR <- R6Class("SimpleSIR",
    public = list(
      S = 99L,
      I = 1L,
      R = 0L,

      update = function(beta=0.01, gamma=0.1){
        nI <- rbinom(1L, self$S, 1.0 - exp(-beta * self$I))
        nR <- rbinom(1L, self$I, 1.0 - exp(-gamma))
        self$S <- self$S-nI
        self$I <- self$I+nI-nR
        self$R <- self$R + nR
      },

      print = function(){
        cat("S:", self$S, ", I:", self$I, ", R:", self$R, "\n", sep="")
      }
    )
)
```

Creating an instance of the class:

```
model_instance <- SimpleSIR$new()
model_instance
## S:99, I:1, R:0
```

Calling methods:

```
model_instance$update()
model_instance
## S:99, I:1, R:0
model_instance$update()
model_instance
## S:98, I:2, R:0
```

Creating an instance of the class:

```
model_instance <- SimpleSIR$new()
model_instance
## S:99, I:1, R:0
```

Calling methods:

```
model_instance$update()
model_instance
## S:99, I:1, R:0
model_instance$update()
model_instance
## S:98, I:2, R:0
```

[More later]

Other programming paradigms

Entity Component Systems (aka ECS) - Similar to OO, except that data and methods are separate - Performance advantages for some applications - Typically implemented in compiled languages like C, C++, Rust

Other programming paradigms

Entity Component Systems (aka ECS) - Similar to OO, except that data and methods are separate - Performance advantages for some applications - Typically implemented in compiled languages like C, C++, Rust

Declarative - Languages that define relationships directly - Statistical models e.g. lme4 syntax, BUGS/JAGS, NIMBLE, Stan - Database languages e.g. SQL - [Pure functional programming is a type of declarative]

Which style to choose?

The important things are that code should be:

- Encapsulated
  - i.e. implementation and usage are separated
- Robust
  - i.e. documented and tested

Which style to choose?

The important things are that code should be:

- Encapsulated
  - i.e. implementation and usage are separated
- Robust
  - i.e. documented and tested

What do I use?

- Data cleaning and exploration -> functional
- Statistical modelling -> declarative
- Complex tasks like mechanistic simulation modelling -> encapsulated OO
- Very simple sub-tasks within these -> procedural

A quick tour of R6

A more complex R6 class

```
library("R6")

SIRmodel <- R6Class("SIRmodel")

SIRmodel$set("private", "time", numeric())
SIRmodel$set("private", "S", numeric())
SIRmodel$set("private", "I", numeric())
SIRmodel$set("private", "R", numeric())

SIRmodel$set("public", "initialize",
  function(S=99L, I=1L, R=0L){
    "Initialize method with optional S/I/R compartment sizes"
    private$S <- S; private$I <- I; private$R <- R
    private$time <- 0L
  }
)
```

```
SIRmodel$set("public", "update",
  function(beta=0.01, gamma=0.1){
    "Update method to increment time and process compartments"
    private$time <- private$time + 1L
    nI <- rbinom(1L, private$S, 1.0 - exp(-beta * private$I))
    nR <- rbinom(1L, private$I, 1.0 - exp(-gamma))
    private$S <- private$S-nI; private$I <- private$I+nI-nR; private$R <- private$R + nR
    invisible(self)
  }
)
```

```
SIRmodel$set("public", "print",
  function(){
    "Simple print method"
    cat("Time:", private$time, ", S:", private$S, ", I:", private$I, ", R:", private$R, "\n",
    ↪  sep="")
  }
)
```

Active bindings are cool:

```
SIRmodel$set("active", "status",
  function(){
    "Read-only property for current status (Time/S/I/R)"
    status <- tibble(Time=private$time, S=private$S, I=private$I, R=private$R)
    return(status)
  }
)
```

Active bindings are cool:

```
SIRmodel$set("active", "status",
  function(){
    "Read-only property for current status (Time/S/I/R)"
    status <- tibble(Time=private$time, S=private$S, I=private$I, R=private$R)
    return(status)
  }
)
```

```
model_instance <- SIRmodel$new()
model_instance$status
## # A tibble: 1 x 4
##    Time     S     I     R
##   <int> <int> <int> <int>
## 1     0    99     1     0
model_instance$update()$status
## # A tibble: 1 x 4
##    Time     S     I     R
##   <int> <int> <int> <int>
## 1     1    99     1     0
```

Assignment semantics

Copying by simple assignment is shallow!

```
model_instance_copy <- model_instance
model_instance$update()$status
## # A tibble: 1 x 4
##    Time     S     I     R
##   <int> <int> <int> <int>
## 1     2    98     2     0
model_instance_copy$status
## # A tibble: 1 x 4
##    Time     S     I     R
##   <int> <int> <int> <int>
## 1     2    98     2     0
```

Use clone to create a deep copy:

```
model_instance_clone <- model_instance$clone()
model_instance$update()$status
## # A tibble: 1 x 4
##    Time     S     I     R
##   <int> <int> <int> <int>
## 1     3    97     2     1
model_instance_clone$status
## # A tibble: 1 x 4
##    Time     S     I     R
##   <int> <int> <int> <int>
## 1     2    98     2     0
```

Helper methods

```
SIRmodel$set("public", "run",
  function(times=100L, ...){
    "Wrapper around the update method returning a tibble as output"
    output <- matrix(NA_integer_, nrow=times, ncol=4L, dimnames=list(NULL,
↪   c("Time","S","I","R")))
    for(t in seq_len(times)){
      self$update(...)
      output[t,] <- c(private$time,private$S,private$I,private$R)
    }
    return(as_tibble(output))
  }
)
```

Helper methods

```
SIRmodel$set("public", "run",
  function(times=100L, ...){
    "Wrapper around the update method returning a tibble as output"
    output <- matrix(NA_integer_, nrow=times, ncol=4L, dimnames=list(NULL,
↪   c("Time","S","I","R")))
    for(t in seq_len(times)){
      self$update(...)
      output[t,] <- c(private$time,private$S,private$I,private$R)
    }
    return(as_tibble(output))
  }
)
```

```
model_instance <- SIRmodel$new(S=99L, I=1L, R=0L)
model_instance$run(3L)
## # A tibble: 3 x 4
##    Time     S     I     R
##   <int> <int> <int> <int>
## 1     1    97     3     0
## 2     2    95     4     1
## 3     3    89     8     3
```

```
model_instance$run(3L)
## # A tibble: 3 x 4
##    Time     S     I     R
##   <int> <int> <int> <int>
## 1     4    83    13     4
## 2     5    73    22     5
## 3     6    63    30     7
model_instance$run(4L)
## # A tibble: 4 x 4
##    Time     S     I     R
##   <int> <int> <int> <int>
## 1     7    47    41    12
## 2     8    32    50    18
## 3     9    17    59    24
## 4    10     6    67    27
```

Combining R6 and S3

It is possible (and beneficial) to use S3 dispatch with R6 classes:

```
as_tibble.SIRmodel <- function(x, ...){
  x$status
}
as_tibble(model_instance)
## # A tibble: 1 x 4
##    Time     S     I     R
##   <int> <int> <int> <int>
## 1    10     6    67    27
```

Combining R6 and S3

It is possible (and beneficial) to use S3 dispatch with R6 classes:

```
as_tibble.SIRmodel <- function(x, ...){
  x$status
}
as_tibble(model_instance)
## # A tibble: 1 x 4
##    Time     S     I     R
##   <int> <int> <int> <int>
## 1    10     6    67    27
```

You can also define the print method this way if you prefer.

Using R6 in an R package

Run:

```
neatpkg::pkg_R6()
```

And look at the example_R6.R file

Using R6 in an R package

Run:

```
neatpkg::pkg_R6()
```

And look at the example_R6.R file

Notes:

- Document R6 methods by providing text on the first line of the method
- You should also document argument for methods BUT they are not checked by R CMD check unfortunately
- It is sometimes easier to treat your R6 class as internal within the package, and export helper functions and/or S3 methods for external use

Exercise

Instructions

1. Add an R6 class to your R package
2. Alter the R6 class so that it implements the methods provided in this presentation
3. Add some argument checking to the methods to ensure that provided arguments are sensible
4. Add some tests under tests/testthat to ensure that your class doesn't produce garbage output
5. Optional: add a suitable autoplot method (see ?ggplot2::autoplot)