R cribsheet: CKMR model/design course

Mark Bravington: updated August 2022
thanks: Laura Tremblay-Boyer, Shane Baylis

1. You probably got this document from github repo `https://github.com/markbravington/csiro22`. That's where all the other course stuff lives, *apart* from a few R packages whose installation instructions are given below. We will update the repo as we go.

2. As noted somewhere, you need R version 4.1.x/4.2.x/4.3x. Most of the recent testing has been with R4.2, but the others *used* to work too...

3. You need to install some packages, from my personal CRAN-alike repository. Fingers crossed, all you'll have to do is this:

   ```
   install.packages( c( 'offarray', 'deconvodisc', 'debug'),
     repos=c('https://markbravington.github.io/Rmvb-repo', getOption('repos')))
   ```

   which installs a *small* number of other packages en route: a couple of mine (`mvbutils`, `atease`), and a couple from CRAN. You'll likely have some of those already. If there are no error messages, and you can do `library(offarray)` without trouble, go to step 5.[1]

4. Paranoia time: *you probably not will reach this step, and you do not need to read about it unless you are unlucky*. If you do see error messages during installation, indicating that some of those packages *weren't* installed... oh dear. Do your best to sort it out ;), within reason, bearing in mind item (a) below. Quite likely, the reason will be a version conflict among packages that I didn't write and have no control over (e.g. `TMB` and `Matrix` can be quarrelsome). Those version conflicts are such fun, aren't they :/? Judicious use of `update.packages()` might get you sorted (read its doco, esp. re. `oldPkgs` argument). If you get completely stuck, let us know, and then we can all bumble around in the dark together...

   (a) For The Record: the only ones you absolutely *need* to have installed to run the course stuff are `mvbutils`, `atease`, and `offarray`. If you can `library(offarray)` without problems (since it autoloads the other two), then you can get by. Of the rest: `deconvodisc`

[1]Note that my CRAN-alike repo is *not* the same as the main course repo, and is *not* a "Github repo" in the normal sense; R & github have quite distinct meanings for "repo". You can't get into my CRAN-alike from github despite its address (I can, you can't). The only way you can access and query my CRAN-alike, is with the same functions you would apply to CRAN itself, using the `repos` argument— `available.packages()` and so on. There is no analogue of the CRAN package-list webpage. That is how R does things!

is used in a couple of examples, but it's more important to understand why it's being used, than to see it running. `debug` (Section 7) might hugely improve your R life, but you can elect to suffer through R-life without it; millions do. But then, millions of people vote for evil morons, too...

(b) FTR 2: the CRAN packages that are directly required , are `mgcv`, `Matrix`, `Rcpp`, `RcppEigen`, `TMB`. They *should* be installed automatically, but... And they in turn require others:

(c) FTR 3: the full dependency chain for my packages is as follows

```
"atease" "graphics" "grDevices" "grid" "lattice" "Matrix" "methods"
"mgcv" "mvbutils" "nlme" "Rcpp" "RcppEigen"
"splines" "stats" "tcltk" "TMB" "tools" "utils"
```

and *most* of those are in R-core's "recommended" list so you would have to try hard to *not* have them already (which does *not* mean that they are perfect, nor that they get along 100% with each other). If things do go wrong due to version conflicts, it's gonna be somewhere within that lot.

5. Hooray! Now, one more thing to test before the course scripts will run: you'll need to *turn off R's byte-compiler*, as described below. [Otherwise, you will find that the scripts eventually fail with mysterious error messages involving `'*tmp*'`.]

6. Laura is providing some tutorials, which do a great job of illustrating both the data and the flow of analysis. There are two versions: with tidyverse, and without. If you want the tidyverse version, you'll also need to install `dplyr` from CRAN. Be aware that installing `dplyr` will trigger auto-installation of a big slew of other packages, "tidily" or otherwise, which can lead to package version problems; you *might* decide the risk isn't worth it... (I am a bit of dinosaur here, but I am also a survivor!) The graphics in the tutorials use some base-R and also some `ggplot2` from CRAN (which does look good, as even a dinosaur has to concede); `ggplot2` also triggers some auto-installs, but not as heinously as `dplyr`. The best advice here might be: if you are already a tidyverse fan, then you'll want to use that version of the tutorials, and you'll have the necessaries installed already. But if you don't use tidyverse, then you don't need to learn it for the course, so maybe stick with the base-R version and don't install `dplyr` (though `ggplot2` might be worth a try).

1 Organization of R stuff for the course

The scripts, datasets, and Laura's tutorials for the course— plus this document and other paraphernalia, e.g. list of my favourite nearby coffee places— are/will be available from Github repo in the Intro[2] (still fiddling with it at time-of-writing). You are not really supposed to look at

[2]This one is a normal Github repo; you can go to the page and clone/download stuff. Not a CRAN-alike "R repo".

the scripts and tutorials before the contents have been explained! However, it's worth having a read thru this docu first, to understand a bit about the structure. Sections 5 and 6 describe some less-well-known R features that are crucial to the scripts.

- The course code all sits in a few script-files with extension dot-r, and the data in "dot-rda" files (which can only be loaded into R— e.g. they are not spreadsheetable). The structure is described in section 3 below.
- Most of the work is done by user-written R functions[3] (), not by line-by-line commands in the scripts. When it comes to it, the core job of a CKMR "modeller" is to write *one* function— a log-likelihood. They'll probably need to write some data-organization stuff too, but that's bread-and-butter.
- The script "ckmr_funs.r" *creates* a bunch of functions that are referred to in the other scripts— you can just `source()` that one without trying to read it.
- Each other script pertains to one specific dataset/example, e.g. "fit_delfi_A.r". You're meant to cut'n'paste lines and intersperse your own, especially for diagnostics/output after the actual fitting step— *not* to just source those other scripts holus bolus, which would teach you absolutely nothing. They each prepare & fit the data for one example; see below for more.
- This docu (tries to) refer to R *scripts* and other files via normal text in quotes, e.g. "fit_notog22.r". R *functions* and other R objects get monospaced courieresque font like this: `lglk_notog()` and `env`. I try to always put parentheses after a function name, so you know it's a function— but I may have slipped up here and there. If you don't understand the difference between scripts and objects, and between functions and other objects, you should get onto Google right now!
- "IMOAL" <=> "In My Opinion, At Least"
- Sections 5 onwards are about R itself: stuff you mightn't know but that is used in the course.

2 Turn off the byte-compiler!

There's a script "turn_off_bc.r" on the repo. Be sure to `source('turn_off_bc.r')` early on fin your R session, and in particular *before* you do `library(offarray)`— which also means, before you run any of the main scripts.

(I'm in the process of working around this for 2022... hmmm it's now looking like 2023 though...)

[3]I.E.: I wrote these particular ones, but by the end of the course you're supposed to be able to write them yourself.

You may not even realize that R has (for the last few years) a built-in byte-compiler which does a lot of very complicated and totally ingenious behind-the-scenes stuff to produce frankly marginal speedups IMOAL of your code. The BC is switched on by default, and in theory it should "just" work without your ever noticing. In practice, there's a bug in it (I reckon) that interferes with my `offarray` package, which is ubiquitous in the scripts here. Because I have automatic code in my R startup to turn *off* the byte-compiler, I always forget that it is likely to bite other people...

I don't rate my chances of getting that BC bug fixed anytime soon (and I don't know exactly what causes it— NFI what goes on inside the BC, but my `offarray` code looks legit to me, albeit complicated), and don't fancy the argy bargy, so I just turn the BC off and forget about it, via "turn_off_bc.r". Eventually I will add an instruction to do that to all the "fit_<blah>r" scripts, but for now you'll need to run it manually. It needs to be `source()` ed once per R session— the earlier the better, I suspect, and I put it in my own ".Rprofile".

3 Idiosyncrasies

I haven't set out to write obscure R code, but everybody's R experience is different and there's no one best/clearest way. So I've just written stuff that *I* think is reasonably clear, and efficient enough to run in real-time— and, crucially, stuff that *I* can follow. It's commented, succinctly; IMOAL having too many mid-code comments is counterproductive. If you are expecting to grasp CKMR, something which can get a bit complicated, then you had better be able to fight your way through (what *you* see as...) the idiosyncrasies of my R code!

The only bits you *really* need to follow, I reckon, are the log-likelihood functions (with `lglk` in the name). You should also be to follow the "fit_<blah-species>.r" scripts— they're simpler, I think— and if you are keen you *could* look at the `boring_data_prep_<blah>()` functions. Laura's tutorials will help, especially if you want to get into the nitty and the gritty.

A couple of tips for deciphering function code (tips you probably won't see in Proper Books):

- Don't necessarily start at the top. If you get a bit stuck, just look vaguely for statements in the *middle* that might do something interesting and recognizable, and focus on those. Then work *backwards* to see how they were set up, and *forwards* to see what comes out at the end.

- If you don't understand a line in detail because it uses "unusual" syntax/functions, then just *ignore* it (for now, or for ever!). It's likely operationally crucial, but only in a nuts-n-bolts way. There are a surprisingly large number of components in an electric kettle (as you find out if you take one apart and try to reassemble it) but the main thing to understand is that there's a big stiff wire that heats stuff up. Eventually you can sort out all the grisly details if you really have to, but don't *start* by trying to understand every minor operational nuance.

That said, there are a few things about the course code that might benefit from a few words. This stuff is *R-specific detail*, not interesting stuff about CKMR, so— as per above— you should not start by trying to memorize it all— whizz thru it quickly now, and return to it only during/after the course when you are flummoxed by something.

- Where do the "data" and "useful byproducts" live/get kept, inside those `...lglk...()` functions? They live in its *`environment`*; more on that below.
- Notation: I am not 100% consistent here, and sometimes favour aesthetics or brevity or laziness over convention, but:
  - "Data" and "pop dyn quantities" often start with capital letters; my general housekeeping variables usually don't. I use underscores a lot, and never use camelCaseBecauseIHateIt, except as noted next.
  - for (off)arrays [see below], I usually embed the "names" of the subscriptors/dimensions at the end, eg `n_comp_POP_BYA` will be 3D with dimensions `B`(irth), `Y`(anked year— of sampling), `A`(ge).
  - Probabilities start with `Pr_`, and are (supposed to be...) of the form `Pr_<things>_<givens>`, so that `Pr_AtruB_CDE` is the probability of "Atru" and "B" (both) given values for "C", "D", and "E"; it will be a 5D array.
- `offarray`: just like a regular R array (or matrix or vector), except that dimensions can start at any number, including negative (regular R arrays must start at 1). Character-indexed dimensions are also allowed, e.g. `SEX=cq(F,M)`. I find `offarray` a lot more natural for "real data" and I make less mistakes. Things basically "just work", but there *are* some differences from base-R behaviour (even if your index happens to start at 1)— see `?offarray` in the eponymous package if you need more info.
  - `dimseq(X)` is useful
  - converting to/from tables is easy
  - to drop dimension(s) from an offarray by taking "slices" (i.e. at just one element from a dimension), you have to explicitly do e.g. `myoff[3:17,SLICE='MALE']`. If you left out the word `SLICE`, then you'd get a 2D offarray where the second dimension has extent 1. This is (deliberately) different from R's default behaviour of dropping extent-1 dimensions silently and wilfully.
  - Many standard R operations, eg `matplot`, will misbehave on `offarrays`; in which case, just do eg `matplot( unclass(x),...)`.
- Loops: instead of nested `for`-loops, you'll see heavy use of `autoloop()` instead, like so:

```
thing_XY <- autoloop(
    X=<Xvalues>, Y=<Yvalues>,
  <R expression involving X & Y>
)
# ... is like doing this:
thing_XY <- offarray(<dimensions derived from X/Yvalues>);
for( thisX in <Xvalues>) {
  for( thisY in <Yvalues>) {
    thing_XY[ thisX, thisY] <- <expression evaluated at thisX and thisY>
}}
```

but usually faster and IMOAL clearer. You can put multiple expressions inside the one call. You can also sum over dimensions you don't need in the final output, either within an `autoloop()` call, or outside it. For example:

```
foobah_X <- autoloop(
    X=<values>,
    SUMOVER=list( Z=<Zvalues>), {
  foo_Z[Z] * bah_XZ[X,Z]
})
lower_dimensional_thing_X <- sumover( thing_XZ, 'Z')
```

- Package `atease` lets me write `x@a` instead of `attr(x,"a")` and `x@a<-<something>` works too. This works for any `x`, not "just" S4 objects (which is what `@` is normally reserved for in R).
- `cq( some, words, here)` is exactly like `c( 'some', 'words', 'here')` but looks less quoty.
- `returnList(a,b)` puts `a&b` into a list with names `"a"` and `"b"`. I often use it to simulate the "multi-argument returns" that R used to have.
- You don't generally need to see inside `boring_data_prep_<blah>()`, but if you do you might see some 'mvbutils' operators such as `'%where%'` (that particular one is for subsetting dataframes by rows). They are documented, and/but kinda obvious.
- I make heavy and unapologetic use of features in base-R, some of which may be unfamiliar (but they are documented, so that's on you). These include:
  `environments` and scoping, `local`, `with`, `evalq`, `mget`, `list2env`, subscript-by-matrix (using `MATSUB=...` syntax for `offarrays`),
- I haven't used any tidyverse or data.table stuff. FWIW Eric Anderson did implement a tidyverse version of (one of) the `boring_data_prep_<blah>()` and `generic_lglk_<blah>()` functions, so it can be done that way too. However, note that full-on CKMR (e.g. for fish stock asst) will generally need TMB and will need to look like the (off)array versions. (Not all "real-life" CKMR is complicated enough to require TMB, though; R is sometimes fine.)

4 Structure of code and data

(This overlaps Laura's new tutorials— which you may well prefer to this. That's all good; hopefully we've said the same thing!)

You first need to `source()` the master script file "ckmr_funs.r" to create all the functions that will be used in the various examples.

The core of each example is a log-likelihood function called something like `generic_lglk_<blah>`. That's the one piece of code you really need to understand for the example— and ultimately to be able to write yourself for your own data.

The actual fitting of CKMR model(s) to example `<specific-eg>` is done by the *script* in file "fit_<blah>.r". The steps are:

1. `load()` the data for that example into the R session ;
2. summarize the data into a form suitable for CKMR models (eg adding up the number of kin-pairs and comparisons by "type", ie individual covariate values and nature-of-kinship)— by calling `boring_data_prep_<blah>(...)`. Basically it's calling `table()`. It also sets the ranges of several covariates, such as years and ages. It's got that name `boring...` for a good reason. You can look at the code if you really want, but I suggest not— concentrate on CKMR! The summaries get stored in an environment, called e.g. `env` — so that e.g. `env$n_MOP` has the MOP-counts.
3. Make a copy of the generic lglk that will be specific for this example, called something like `lglk_<specific-eg>()`.
   (a) The reason you have to make a copy is that `generic_lglk_<blah>()` mentions the data by name (eg `n_MOP`) but can't be called as-is, because it hasn't been told what those data actually are for this *specific* example. And you might well be able to re-use the same generic in different examples, so I don't want to tie it to a single dataset from the get-go.
4. "Tell" your copy `lglk_<specific-eg>()` about the specific data, by resetting its environment to `env`. (This is fundamental R stuff, but lots of people don't know about it— google "R lexical scoping".)
   (a) Whenever `lglk_<specific-eg>()` gets called, it will now update various intermediate quantities such as the fitted numbers-at-age, which are also stored in `env`. They will stay put in `env` after the call has finished. That's how you get at estimated quantities-of-interest (such as the array of numbers-at-age-and-year) after actually fitting the model.
   (b) Note that `env` and `environment(lglk_<specific-eg>)` are now *identical*— they refer to exactly the same bit of memory and they're not copies of each other. Changing something in `env` will also change it in `environment(lglk_<specific-eg>)`, and conversely.

5. pick starting values and fit the model using `nlminb()`. As I said, you've now got access to eg estimated numbers-at-age-and-year matrix inside `env`.

6. Tiddle around with the output in various ways: plots, diagnostics, covariances;... All based on stuff in `env`.

4.1 Data

The data for example "<blah>" lives in the file "samp_<blah>_raw.rda", which you need to `load()` (as per the script for that example) to create an object `samp_<blah>` in your R session. It's a list with components `Samps` and the various kin-pairs, e.g `POPs`, `MHSPs`, `PHSPs`. Some non-sample-related "background" data for the example (e.g. catches-at-age, age-at-maturity) is stored in the attribute `public` which, if you have loaded my package `atease`, you can get at via `samp_<blah>@public`. There might also be "secret true stuff" that you're *not* allowed to look at while fitting, in `samp_<blah>@secre`

The `POPs` etc are all two-column matrices whose entries are row-numbers of `Samps`. Order may matter: `MOPs` should contain Mothers in the first column and Offsprings in the second, and in principle I would organize halfsibs as O(rdered)HSPs where the firstborn is in the first column. However, when ages are uncertain you may not be able to tell which is which (and genetics alone won't usually tell you). If I was being really consistent throughout I would call things `OMMOPs` and so on (Offspring-Mother/Mother-Offspring Pair) but consistency can get in the way of comprehensibility.

`$Samps` will include fields `$Me` (a name), `$Sex`, `$Y`, plus any sample-specific age, length, or other data. It also— and this was possibly a design error on my part— has two logicals `$poss_par` and `$poss_off`. The former says whether the sample was used as a potential parent in POP-checks. `$poss_off` says whether it was used as potential offspring in POP-checks and HSP-checks. The reason they are not both usually `TRUE` for each sample, is that I may deliberately only have "done" (or kept the result of) certain types of comparisons, because they're the ones I actually expect to use in the modelling. [Note that `poss_off` and `poss_par` are *decisions of the modeller*, not *basic properties of the data*— that's why they arguably shouldn't be built into `$Samps`. Too late now!] In the boring-data-prep and lglk steps, comparisons are only totted up across samples that were listed as "suitable" via their `$poss` fields.

In CKMR you don't *have* to use every type of comparison— part of the art is deciding which ones aren't worth the trouble of trying to use. (The only rule is: you're not allowed to cheat by first doing a comparison and afterwards deciding whether to retain it based on its outcome!) The `poss_par/poss_off` designation is *not* the most general (we now use something more complicated for SBT, based on *both* animals' covariates), but it's nice & simple for these examples.

4.2 How I obtained the data

Naturally, via an elaborate process of stakeholder engagement, consultation, and respectful outreach, after completing all required e-learning modules. Plus, of course, bribery, flattery, and downright extortion as required: i.e., often.

Or, more prosaically:

I did an IBM simulation of each population with my R package `kinsimmer`, using `simkin` which keeps track of ancestry. Then I sampled from its dead ones and picked the kin-pairs in the sample, using `prep_from_sim2`; and saved the result into a file "samp_<blah>_raw.rda". Package `kinsimmer` is very fast (considering it's all in R, anyway) and reasonably flexible; it's based on Shane Baylis' package `fishSim` but tweaked for my own fell purposes. *If* you're doing your own simulations, then you might prefer to use Shane's package, which is probably better-documented and more general (it doesn't have to be fish!) but slower. *However*— as I'll explain during the course— I generally *don't* recommend simulations for routine CKMR anyway!

5 Attributes

Most R objects can have *attributes* stuck onto them. Some attributes are built-in, like `dim` and `length`; but you can (usually) add your own, of any type and with any name you like. They are handy for storing "metadata". For example, the CK data pertaining to each scenario consists of a `list` with 2+ core CKMR elements: `$Samps`, `$POPs`, and/or `$HSPs`. But there's also case-specific background data, such as age-at-maturity or catch-at-age. Plus there's "secret" parameter values for the simulated dataset, e.g. the true abundances over time. The case-specific stuff is all stored in a list called `public`, which is attached as an attribute the CKMR data list. The secret stuff is also stored in a list called, yes, `secret`, which is attached as another attribute. So, for the *Notogorgius* (fish) case we have

```
> names( samp_notog1)
[1] "Samps" "POPs"  "HSPs"
> names( attributes( samp_notog1))
[1] "secret" "public"
> names( attr( samp_notog1, "public"))
[1] "C_sya" "Amat"  "wt_a" # catch-at-age, age-at-maturity, weight-at-age
```

As you can see, base-R syntax for accessing and showing-names-of attributes is a PITA. So my tiny package `atease` lets you do this instead:

```
> atts( samp_notog1) # instead of names( attributes( samp_notog1))
[1] "secret" "public"
> names( samp_notog1@public)) # refers to attribute by name, more directly
```

```
[1] "C_sya" "Amat"  "wt_a" # catch-at-age, age-at-maturity, weight-at-age
> # Don't do this to your data! But the @-syntax makes it easy...
samp_notog1@public$C_sya <- samp_notog1@public$C_sya * 2
```

It's borrowing the "@" symbol used normally for S4 slot access. If you don't know what the latter is, my advice is: keep it that way :)

6 Environments and "lexical scoping"

The course code uses R environment objects to stash "permanent" data and "just-calculated" intermediate results required and/or calculated by log-likelihood functions. Most people don't know about this stuff— you often don't really need to— but it's a really powerful fundamental feature of R, and I take advantage of it to make my code shorter and IMOAL clearer. Of course, the main point of the course code is just to illustrate how CKMR models should be programmed— *not* to understand the R minutiae. But if you really do want to understand how the code works or even to adapt it for your own purposes, you'll need to understand how it uses environments.

An "environment" object in R is a lot like a "list": it contains several things, each with its own name, and you can extract/set them via e.g. `myenv$toothpaste` or `myenv[['toothpaste']]`. Environments crop up in several places. Importantly, every R function you write (or I write), comes with its own environment— a *permanent* thing (though you can change its contents). When the function is executing, it also has a *temporary* environment, which holds all the variables created while the function runs, plus the function's arguments; that temporary environment vanishes when the function concludes. If, while executing, the function sees a variable-name that's *not* in its temporary environment, then it looks for that variable in the function's permanent environment (`P`, say); if not found there, it looks in the *parent environment* of `P`; if not there, then in the parent of `P`'s parent; and so on. What are these "parents", you ask? Simply: every environment has one "parent", also known as its "enclosing" environment. Eventually, this ancestor-chain (usually) reaches `.GlobalEnv`, which is the "workspace" you see directly from the command-line; that's why functions can usually get away with referring to "global variables" (though the potential for confusion is huge). In fact, if you just define a function at the command-line without taking special measures, then its permanent environment will be `.GlobalEnv` itself— a very short ancestor-chain. And `.GlobalEnv` itself has a ancestor-chain: the "search path" that you get from `search()`, which ends (basically) in the "base environement" `baseenv()` (you have to call that *function* to refer to the "base environment", there's not a reliable built-in variable— Becos R). Functions in packages have slightly different ancestor-chains.

The practical upshot for complicated log-likelihoods and so on, like all the CKMR `lglk...` in this course, is that you can use your `lglk_X`'s permanent environment to:

1. stash all the constants and data that you need to compute the lglk (not the parameters,

obviously; they are set thru the function's arguments when you call it), so you don't need to explicitly pass in all those data-and-constants as arguments every time you call the function;

2. store intermediate quantities computed during the lglk calculation, e.g. the population-dynamics numbers, so they can be retrieved afterwards. (They will be created in the temporary environment, but you can copy them to the permanent one);

3. and if you have several datasets all with similar structure to which you're going to fit the exact same model each time, you can make a separate copy of the "generic" function for each one, and set the permanent environment of each copy to match one particular dataset. (Don't change the environment of the generic— too risky.)

Here's an example of stashing some data:

```
> somefun <- function( x) x+mm^2 # data is 'mm'
> suppressWarnings( rm( mm)) # make sure it's not just lying around somewhere
> somefun( 1) # won't work; can't find 'mm'
Error in somefun() : object 'mm' not found

> environment( somefun) # So, where does somefun() look for stuff, by default
<environment: R_GlobalEnv>

> zoup <- new.env()
> zoup$mm <- 7 # defining 'mm' inside environment 'zoup'
> environment( somefun) <- zoup # make it look here
> somefun( 1) # bing bong
[1] 50
```

And here's an example of storing intermediate quantities for subsequent "posthumous" retrieval:

```
> var_recip_x <- function( xvec){
+   mean_recip <- mean( 1/xvec)
+   mean_recip2 <- mean( (1/xvec)^2)
+
+   # Now gonna put both into the permanent env, for later access
+   env <- environment( sys.function())
+   list2env( mget( c( 'mean_recip', 'mean_recip2')), envir=env) # trust me i
+ return( mean_recip2 - mean_recip^2)
+ }

> zoup <- new.env()
> environment( var_recip_x) <- zoup
> ls( zoup) # nada
```

```
character(0)

> var_recip_x( 1:5)
[1] 0.08418

> ls( zoup) # oooooh!
[1] "mean_recip"  "mean_recip2"

> zoup$mean_recip # ta-daaah
[1] 0.4567
```

6.1 More on `environments`

There are a couple of good web resources on R environments: <Brodie Gaslam> and <that one I fwded to Robin>. And there are a few R functions specifically for environments which you should know about: e.g. for converting to/from lists. R has a help system and manuals for all that! (Try "Scope" in "An Introduction to R". And there is stuff in "R Language Definition" which may help and/or hinder comprehension.)

Powerful though environments are, a few cautions are in order:

1. Environment chains can also make your code *spectacularly* incomprehensible, if you try hard enough. Some of the people who wrote the base-R code seem to enjoy doing that. Try figuring out how on earth `lazyLoad()` works, or how `srcref`'s are implemented. Grrrrr!

2. Unlike almost everything else in R, an environment is a *reference object*; it just "points" to a collection of stuff in memory, rather than *being* that collection. In particular, two environments with different names can actually "point" to exactly the same collection, and changing the contents of one will change the other. R just doesn't do that with anything else! (Though C etc do.) So the following code might be a surprise:

   ```
   A <- new.env()
   A$toothpaste <- 'okapi'
   B <- A
   B$toothpaste <- 'lambrusco'
   A$toothpaste # surprise!
   ```

   If you changed the first line to `A<-list()`, you would *not* get the same behaviour. But don't panic: when you set up an environment, it does make an actual separate copy of anything you put into it, so e.g. this won't cause problems:

   ```
   A <- new.env()
   ```

```
earwax <- 5
A$earwax <- earwax
A$earwax <- 27
earwax # still 5
```

That is: the "collection-of-stuff" which one (or more) environments point to, can only be accessed via those environment(s), and is completely separate from any other copies of the things inside it.

3. Be careful of the distinction between the environment that you *call* a function from (usually either `.GlobalEnv`, or the temporary environment of some other function that's executing), vs. the temporary-and-permanent environments of the function itself. They are different— except if they happen to be the same ;). Does that sound confusing? Well, it is. But it's the basis of *lexical scoping*. Google is your friend here :)

7 The `debug` package

My `debug` package is to help with debugging functions— the main objective— and nowadays also scripts, even `knitr` scripts (sort of). You don't absolutely *need* it. But it might help your R life out a lot. And it might save a lot of public $ that you would otherwise $pend trying to get stuff right. And even if your own code is Perfect (Well Done You!!!), using `debug` might help you understand what in hell is going on with other people's monstrosities!

Plus, if you've actually followed the instructions, you've already installed it. So why not have a look? `package?debug` will get you started; then follow the links, e.g. to the main function `mtrace()`.

Here are some comments I've heard on R debugging, and my responses (cleaned up for general audience):

- "*R/Rstudio has already got built-in debugging facilities, like `browse`*": Mmmm yeah kinda sorta— but they're pretty rudimentary. I had better debugging tools available when I started coding "properly" in the late 1980s, with Turbo Pascal. (C++ programmers do not know what they are missing, BTW.)

- "*I just paste statements one-at-a-time into the console to test them*": Sure, we all do— but you can't when you're debugging a `for`-loop or `if`-statement, for example. And, when you are hoping to imitate what goes on *inside* a function but from the command line, goodness knows what leftovers might be in your `.GlobalEnv` to confuse matters.

- "*I just use `print` statements*": Well, you *can* light a fire by rubbing sticks together, too— maybe[4]. Or you can use matches. People in the know seem to prefer the latter.

[4]Actually it's quite difficult.

- "*If one approaches one's coding Properly, e.g. in accordance with ISO1778-14, one will find that one has no need for a debugger.*": Ummm, well, "that's nice, dear". Meanwhile, back on Planet Earth...

It's also worth looking at Simon Wood's web-page on such matters: `https://www.maths.ed.ac.uk/~swood` (NB it is out-of-date on a few details of `debug`; I've improved a number of things in `debug` in recent years.)

Curiously enough, this CKMR course is one of the less-useful instances for `debug`, because I've used `autoloop()` so heavily and `debug` doesn't currently look inside that (though `debug` can look inside other things, like `with())` . Really, CKMR code is so simple that you may not need a debugger (much), even if you don't follow ISO17.... But `debug` sure helped when I was trying to get the simulator going.