Defensive Programming in R

Sarah Cobey

Defensive Programming

- **Goal:** Convince new and existing programmers of the importance of defensive programming practices, introduce general programming principles, and provide specific tips for programming and debugging in `R`.
- **Audience:** Scientific researchers who use `R` and believe the accuracy of their code and/or efficiency of their programming could be improved.
- **Installation:** For people who have completed the other modules, there is nothing new to install. For others starting fresh, install `R` and `RStudio`. To install `R`, follow instructions at `cran.rstudio.com`. Then install `Rstudio` following the instructions at `goo.gl/a42jYE`. Download the `knitr` package.

Motivation

Defensive programming is the practice of anticipating errors in your code and handling them efficiently.

If you're new to programming, defensive programming might seem tedious at first. But if you've been programming long, you've probably experienced firsthand the stress from

- inexplicable, strange behavior by the code
- code that seems to work under some conditions but not others
- incorrect results or bugs that take days or weeks to fix
- a program that seems to produce the correct results but then, months or years later, gives you an answer that you know must be wrong... thereby putting all previous results in doubt
- the nagging feeling that maybe there's still a bug somewhere
- not getting others' code to run or run correctly, even though you're following their instructions

Defensive programming is thus also a set of practices for preserving sanity and conducting research efficiently. It is an art, in that the best methods vary from person to person and from project to project. As you will see, which techniques you use depend on the kind of mistakes you make, who else will use your code, and the project's requirements for accuracy and robustness. But that flexibility does not imply defensive programming is "optional": steady scientific progress depends on it. In general, we need scientific code to be perfectly accurate (or at least have well understood inaccuracies), but compared to other programmers, we are less concerned with security and ensuring that completely naive users can run our programs under diverse circumstances (although standards here are changing).

In the first part of this tutorial, we will review key principles of defensive programming for scientific researchers. These principles hold for all languages, not just `R`. In the second part, we will consider R-specific debugging practices in more depth.

Part 1: Principles

Part 1 focuses on defense. You saw a few of these principles in Basic Computing 2, but they are important enough to be repeated here.

1. Before writing code, draft your program as a series of modular functions with high-level documentation, and develop tests for it.

2. Write clearly and cautiously.
3. Develop one function at a time, and test it before writing another.
4. Document often.
5. Refactor often.
6. When you run your code, save each "experiment" in a separate directory with a complete copy of the code repository and all parameters.
7. Don't be *too* defensive.

In part 2, we will focus on what to do when tests (from Principle 3) indicate something is wrong.

Principle 1. Before writing code, draft your program as a series of modular functions with high-level documentation, and develop tests for it.

Many of us have had the experience of writing a long paper only to realize, several pages in, that we in fact need to say something slightly different. Restructuring a paper at this stage can be a real chore. Outlining your code as you would outline a paper avoids this problem. In fact, outlining your code can be even more helpful because it helps you think not just generally about how your algorithm will flow and where you want to end up, but also about what building blocks (functions and containers) you'll use to get there. Thinking about the "big picture" design will prevent you from coding yourself into a tight spot—when you realize hundreds of lines in that you should've been tracking past states of a variable, for instance, but your current containers are only storing the current state. Drafting also makes the actual writing much more easy and fun.

For complex programs, your draft may start as a diagram or flowchart showing how major functions and variables relate to one another or brief notes about each of step of the algorithm. These outlines are known as pseudocode, and there are myriad customs for pseudocode and programmers loyal to particular styles. For simple scripts, it is often sufficient to write pseudocode as placeholder comments. For instance, if we are simulating a population in which individuals can be born or die (and nothing else happens), we could write:

```
# initialize population size (N), birth rate (b), death rate (d), total simulation time (t_max), and ti
# while N > 0 and t < t_max
# ...generate random numbers to determine whether next event is birth or death, and time to event (dt)
# ...update N (increment if birth, decrement if death) and update time t to t+dt
```

Here, `b` and `d` are per capita rates. This is an example of the Gillespie algorithm. It was initially developed as an exact stochastic approach for simulating chemical reaction kinetics, and it is widely used in biology, chemistry, and physics.

Can you see some limitations of the pseudocode so far? First, it lacks obvious modularity, though this is partly due to its vagueness. The first step under the `while` loop could become its own function that is defined separately. Second, it is missing a critical feature, in that it's not obvious what is being output: do we want the population size at the end of the simulation, or the population size at each event? If the latter, we may need to initialize a container, such as a dataframe, in which we store the value of `N` and `t` at every event. After further thought, we might decide such a container would be too big—perhaps we only need to know the value of `N` at 1/1000th the typical rate of births, and so we might introduce an additional loop to store `N` only when the new time `(t+dt)` exceeds the most recent prescribed observation/sampling time. This sampling time would need to be stored in an extra variable, and we could also make the sampling procedure into its own function. And maybe we want a function to plot `N` over time at the end.

The next stage of drafting is to consider how the code might go wrong, and what it would take to convince ourselves that it is accurate. We'll spend more time on this later, but now is the time to think of every possible sanity check for the code. Sometimes this can lead to changes in code design.

Exercise

What sanity checks and tests would you include for the code above?

Some examples:

- Initial values of `b`, `d`, and `t_max` should be non-negative and not change
- The population size `N` should probably start as a positive whole number and never fall below zero
- If the birth and death rates are zero, `N` should not change
- If the birth rate equals the death rate, on average, `N` should not change when it is large
- The ratio of births to deaths should equal the ratio of the birth rate to the death rate, on average
- The population should, on average, increase exponentially at rate `b-d` (or decrease exponentially if `d>b`)

Some of these criteria arise from common sense or assumptions we want to build into the model (for instance, that the birth and death rates aren't negative), and others we know from the mathematics of the system. For instance, the population cannot change if there are no births and deaths, and it must increase on average if the birth rate exceeds the death rate. It helps that the Gillespie algorithm represents kinetics that can be written as simple differential equations. However, because the simulations are stochastic, we need to look at many realizations (randomizations, trajectories) to ensure there aren't consistent biases. We must use statistics to confirm that the distribution of `N` in 10,000 simulations after 100 time units is not significantly different from what we would predict mathematically.

The bottom line is that we have identified some tests for the (1) inputs, (2) intermediate variable states (like `N`), (3) final outputs to test that the program is running correctly. Note that one of our tests, the fraction of birth to death events, is not something we were originally tracking, and thus we might decide now to create a separate function and variables to handle these quantities. We will talk in more detail about how to implement these tests in Principle 3. Generally, tests of specific functions are known as **unit tests**, and tests of aggregate behavior (like the trajectories of 10,000 simulations) are known as **system tests**. As a general principle, we need to have both, and we need as much as possible to compare their outputs to analytic expectations. It is also very useful to identify what you want to see right away. For instance, you may want to write a function to plot the population size over time *before* you code anything else because having immediate visual feedback can be extremely helpful.

> "Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do." -Donald Knuth

This is also a good time to draft very high-level documentation for your code, for instance in a `readme.MD` file. What are the inputs, what does the code do, and what does it return?

> **Exercise**
>
> Assume there are two discrete populations. Each has nonoverlapping generations and the same generation time. Their per capita birth rates are `b1` and `b2`. Some of the newborns migrate between populations.
>
> - Write pseudocode to calculate the distribution of population frequencies after 100 generations.
> - Is your code optimally modular?
> - What are the inputs and outputs of the program? Of each function?
> - How could you test the inputs, functions, and overall program?
> - Discuss your approach with your neighbor.

Principle 2. Write clearly and cautiously.

You're already on your way to writing clearly and cautiously if you outline your program before you start writing it. Here we'll discuss some practices to follow as you write.

A general rule is that it is more important for scientific code to be readable and correct than it is for it to be fast. Do not obsess too much about the efficiency of the code when writing.

> "Premature optimization is the root of all evil." -Donald Knuth

Develop useful conventions for your variable and function names. There are many conventions, some followed more religiously than others. It's most important to be consistent within your own code. *

Don't make yourself and others guess at meaning by making names too short. It's generally better to write out `maxSubstitutionRatePerSitePerYear` or `max.sub.rate.per.site.yr` than `maxSR`. * However, very common variables should have short names. * When helpful, incorporate identifiers like `df` (data frame), `ctr` (counter), or `idx` (index) into variable names to remind you of their purpose or structure. * Customarily, variables that start with capital letters indicate global scope in `R`, and function names also start with capital letters. People argue about conventions, and they vary from language to language. Here's Google's style guide.

Do not use magic numbers. "Magic numbers" refer to numbers hard-coded in your functions and main program. Any number that could possibly vary should be in a separate section for parameters. The following code is not robust:

```
while ((age >= 5) && (inSchool == TRUE)) {
  yearsInSchool = yearsInSchool + 1
}
```

We may decide the age cutoff of 5 is inappropriate, but even if we never do, being unable to change the cutoff limits our ability to test the code. Better:

```
while ((age >= AgeStartSchool) && (inSchool == TRUE)) {
  yearsInSchool = yearsInSchool + 1
}
```

Most of the time, the only numbers that should be hard-coded are 0, 1, and pi.

Use labels for column and row names, and load functions by argument.

Don't force (or trust) yourself to remember that the first column of your data frame contains the time, the second column contains the counts, and so on. When reviewing code later, it's harder to interpret `cellCounts[,1]` than `cellCounts$time`.

In the same vein, if you have a function taking multiple inputs, it is safest to pass them in with named arguments. For instance, the function

```
BirthdayGiftSuggestion <- function(age,budget) {
  # ...
}
```

could be called with

```
BirthdayGiftSuggestion(age=30,budget=20)
# or
BirthdayGiftSuggestion(30,20)
```

but the former is obviously safer.

Avoid repetitive code. If you ever find yourself copying and pasting a section of code, perhaps modifying it slightly each time, stop. It's almost certainly worth writing a function instead. The code will be easier to read, and if you find an error, it will be easier to debug.

Principle 3. Develop one function at a time, and test it before writing another.

The first part of this principle is easy for scientists to understand. When building code, we want to change one thing at a time. It makes it easier to understand what's going on. Thus, we start by writing just a single function. It might not do exactly what we want it to do in the final program (e.g., it might contain mostly placeholders for functions it calls that we haven't written yet), but we want to be intimately familiar with how our code works in every stage of development.

The second part of this principle underscores one of the most important rules in defensive programming: **do not believe anything works until you have tested it thoroughly, and then keep your guard up.**

Expect your code to contain bugs, and leave yourself time to play with the code (e.g., by trying to "break" it) until you can convince yourself they are gone. This involves an extra layer of defensive programming beyond the straightforward good practices discussed in Principle 2. Testing the code as you build it makes it much faster to find problems.

Unit tests. Unit tests are tests on small pieces of code, often functions. An intuitive and informal method of unit testing is to include `print()` statements in your code.

Here's a function to calculate the Simpson Index, a useful diversity index. It gives the probability that two randomly drawn individuals belong to the same species or type:

```
SimpsonIndex <- function(C) {
  print(paste(c("Passed species counts:", C),collapse=" "))
  fractions <- C/sum(C)
  print(paste(c("Fractions:", fractions), collapse=" "))
  S <- sum(fractions^2)
  print(paste("About to return S =",S))
  return(S)
}

# Simulate some data
numSpecies <- 10
maxCount <- 10^3
fakeCounts <- floor(runif(numSpecies,min=1,max=maxCount))

# Call function with simulated data
S <- SimpsonIndex(fakeCounts)
```

```
## [1] "Passed species counts: 332 772 252 807 629 407 928 429 769 625"
## [1] "Fractions: 0.0557983193277311 0.129747899159664 0.0423529411764706 0.13563025210084 0.105714285
## [1] "About to return S = 0.113253640279641"
```

It's very useful to print values to screen when you are writing a function for the first time and testing that one function. When you've drafted your function, I recommend walking through the function with `print()` and comparing the computed values to calculations you perform by hand or some other way. It can also be useful to do this at a very high level (more on that later).

The problem with relying on `print()` is that it rapidly provides too much information for you to process, and hence errors can slip through.

Assertions

A more reliable way to catch errors is to use assertions. Assertions are automated tests embedded in the code. The built-in function for assertions in `R` is `stopifnot()`. It's very simple to use.

Let's remove the `print` statements and add a check to our input data:

```
SimpsonIndex <- function(C) {
  stopifnot(C>0)
  fractions <- C/sum(C)
  S <- sum(fractions^2)
  return(S)
}
```

If each element of our abundances vector `C` is positive, `stopifnot()` will be `TRUE`, and the program will continue. If any element does not satisfy the criterion, then `FALSE` will be returned, and execution will terminate. Explore for yourself:

```
# Simulate two sets of data
numSpecies <- 10
maxCount <- 10^3
goodCounts <- floor(runif(numSpecies,min=1,max=maxCount))
badCounts <- floor(runif(numSpecies,min=-maxCount,max=maxCount))

# Call function with each data set
S <- SimpsonIndex(goodCounts)
S <- SimpsonIndex(badCounts)
```

This gives a very literal error message, which is often enough when we are still developing the code. But what if the error might arise in the future, e.g., with future inputs? We can use the built-in function `stop()` to include a more informative message:

```
SimpsonIndex <- function(C) {
  if(any(C<0)) stop("Species fractions should be positive.")
  fractions <- C/sum(C)
  S <- sum(fractions^2)
  return(S)
}
```

Now try it with `badCounts` again.

What about warnings? For instance, our calculation of the Simpson Index is an approximation: the index formally assumes we draw without replacement, but we compute $S = \sum p^2$, where p is the fraction of each species. It should be $S = \sum \frac{n(n-1)}{N(N-1)}$, where n is the abundance of each species n and N the total abundance. This simplification becomes important at small sample sizes. We could add a warning to alert users to this issue:

```
SimpsonIndex <- function(C) {
  if(any(C<0)) stop("Species fractions should be positive.")
  if((mean(C)<20) || (min(C)<5)) warning("Small sample size. Result will be biased. Consider corrected i
  fractions <- C/sum(C)
  S <- sum(fractions^2)
  return(S)
}

smallCounts <- runif(10)
S <- SimpsonIndex(smallCounts)
```

```
## Warning in SimpsonIndex(smallCounts): Small sample size. Result will be
## biased. Consider corrected index.
```

The main advantage of `warning()` over `print()` is that the message is red and will not be confused with expected results, and warnings can be controlled (see `?warning`).

You could make a case that `warning()` should be `stop()`. In general, with defensive programming, you want to halt execution quickly to identify bugs and to limit misuse of the code.

> **Exercise**
>
> What other input checks would make sense with `SimpsonIndex()`? You can see how the code would be more readable and organized if most of that function were dedicated to actually calculating the Simpson Index. Draft a separate function, `CheckInputs()`, and include all tests you think are reasonable. When you and a neighbor are done, propose a bad or dubious input for their function and see if it's caught.

There are many packages that produce more useful assertions and error messages than what is built into `R`.

See, e.g., `assertthat` and `testit`.

Exception handling

Warnings and errors are considered "exceptions." Sometimes it is useful to have an automated method to handle them. `R` has two main functions for this: `try()` allows you to continue executing a function after an error, and `tryCatch()` allows you to decide how to handle the exception.

Here's an example:

```
UsefulFunction <- function(x) {
  value <- exp(x)
  otherStuff <- rnorm(1)
  return(list(value,otherStuff))
}

data <- "2"
results <- UsefulFunction(data)
print(results)
```

Now `results` is quite a disappointment: it could've at least returned a random number for you, right? You could instead try

```
UsefulFunction <- function(x){
  value <- NA
  try(value <- exp(x))
  otherStuff <- rnorm(1)
  return(list(value,otherStuff))
}
results <- UsefulFunction(data)
print(results)
```

```
## [[1]]
## [1] NA
##
## [[2]]
## [1] -1.310053
```

Even though the function still can't exponentiate a string (`exp("2")` still fails), execution doesn't terminate. If we want to suppress the error message, we can use `try(..., silent=TRUE)`. This obviously carries some risk!

We could make this function even more useful by handling the error responsibly with `tryCatch()`:

```
UsefulFunction <- function(x){
  value <- NA
  tryCatch ({
      message("First attempt at exp()...")
      value <- exp(x)},
    error = function(err){
      message(paste("Darn:",err," Will convert to numeric."))
      value <<- exp(as.numeric(x))
      }
    )
  otherStuff <- rnorm(1)
  return(list(value,otherStuff))
}
results <- UsefulFunction(data)
```

```
print(results)
```

It is also possible to assign additional blocks for warnings (not just errors). The `<<-` is a way to assign to the value in the environment one level up (outside the `error=` block).

Exercise

The new package `ggjoy` works with package `ggplot2` to show multiple distributions in a superimposed but interpretable way. Let's say we want to run the following code:

```
library(ggplot2)
library(ggjoy)
ggplot(iris, aes(x = Sepal.Length, y = Species)) + geom_joy()
```

You probably don't have `ggjoy` installed yet, so you'll get an error. Use `tryCatch()` so that the package is installed if you do not have it and then loaded.

Test all the scales!

It's important to consider multiple scales on which to test as you develop. We've focused on unit tests (testing small functions and steps) and testing inputs, but it is easy to have correct subroutines and incorrect results. For instance, we can be excellent at the distinct activities of toasting bread, buttering bread, and eating bread, but we will fail to enjoy buttered toast for breakfast if we don't pay attention to the order.

With scientific programming, it is critical to simplify code to the point where results can be compared to analytic expectations. You saw this in Principle 1. It is important to add functions to check not only inputs and intermediate results but also larger results. For instance, when we set the birth rate equal to the death rate, does the code reliably produce a stable population? We can write functions to test for precisely such requirements. These are **system tests**. When you change something in your code, always rerun your system tests to make sure you've not messed something up.

It's hard to overstate the importance of taking a step back from the nitty-gritty of programming and asking, Are these results reasonable? Does the output make sense with different sets of extreme values? Schedule time to do this, and update your system tests when necessary.

Principle 4. Document often.

It is helpful to keep a running list of known "issues" with your code, which would include the functions left to implement, the tests left to run, any strange bugs/behavior, and features that might be nice to add later. Sites like GitHub and Bitbucket allow you to associate issues with your repositories and are thus very helpful for collaborative projects, but use whatever works for you. Having a formal list, however, is much safer than sprinkling to-do comments in your code (e.g., `# CHECK THIS!!!`). It's easy to miss comments.

Research code will always need a `readme` describing the software's purpose and implementation. It's easiest to develop it early and update as you go.

Principle 5. Refactor often.

To refactor code is to revise it to make it clearer, safer, or more efficient. Because it involves no changes in the scientific outputs (results) of the program, it might feel pointless, but it's usually not. Refactor when you realize that your variable and function names no longer reflect their true content or purpose (rename things quickly with `Ctrl + Alt + Shift + M`), when certain functions are obsolete or should be split into two, when another data structure would work dramatically better, etc. Any code that you'll be working with for more than a week, or that others might ever use, should probably be refactored a few times. Debugging will be easier, and the code will smell better.

Important tip, repeated: Run unit and system tests after refactoring to make sure you haven't messed anything up. This happens more than you might think.

Principle 6. When you run your code, save each "experiment" in a separate directory with a complete copy of the code repository and all parameters.

When you're done developing the code and are using it for research, keep results organized by creating a separate directory for each execution of the code that includes not only the results but also the precise code used to generate the results. This way, you won't accidentally associate one set of parameters with results that were in fact generated by another set of parameters. Here's a sample workflow, assuming your repository is located remotely on GitHub, and you're in a UNIX terminal:

```
$ mkdir 2017-09-15_rho=0.5
$ cd 2017-09-15_rho=0.5
$ git clone git@github.com:MyName/my-repo
```

If we want, we can edit and execute our code from within `R` or `RStudio`, but we can also keep going with the command line. Here we are using a built-in UNIX text editor known as emacs. If you are a glutton for punishment, you could instead use vi(m).

```
$ cd my-repo
$ emacs parameters.json // (edit parameters, with rho=0.5)
$ Rscript mycode.R
```

Keeping your experiments separate is going to save your sanity for larger projects when you repeatedly revise your analyses. It also makes isolating bugs easier.

Principle 7. Don't be *too* defensive.

This is not necessary:

```
myNumbers <- seq(from=1,to=500,length.out=20)
stopifnot(length(myNumbers)==20)
```

It's fine to check stuff like this when you're getting the hang of a function, but it doesn't need to be in the code. Code with too many defensive checks becomes a pain to read (and thus debug). Try to find a balance between excess caution and naive optimism. Good luck!

Part 2: Debugging in R

Part 1 introduced principles that should minimize the need for aggressive debugging. You're in fact already debugging if you're regularly using input, unit, and system tests to make sure things are running properly. But what happens when despite your best efforts, you're not getting the right result?

We'll focus on more advanced debugging tools in this part. First, some general guidelines for fixing a bug: * *Isolate the error and make it reproducible.* Try to strip the error down to its essential parts so you can reliably reproduce the bug. If a function doesn't work, copy the code, and keep removing pieces that are non-essential for reproducing the error. When you post for help on the website stackoverflow, for instance, people will ask for a MRE or MWE—a minimum reproducible (working) example. You need to have not only the pared code but also the inputs (parameter values and the seeds of any random number generators) that cause the problem. * *Use assertions and debugging tools to hone in on the problem.* We've already seen how to use `stop()` and `stopifnot()` to identify logic errors in unit tests. We'll cover more advanced debugging here. * *Change/Test one thing at a time.* This is why we develop only one function at a time.

Tracing calls

If an error appears, a useful technique is to see the call stack, the sequence of functions immediately preceding the error. We can do this in R using `trackback()` or in RStudio.

The following example comes from a nice tutorial by Hadley Wickham:

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
f(10)
```

You can see right away that running this code will create an error. Try it anyway in RStudio. Click on the "Show Traceback" to the right of the error message. What you're seeing is the stack, and it's helpfully numbered. At the bottom we have the most recent (proximate) call that produced the error, the preceding call above it, and so on. (If you're working from a separate R file that you sourced for this project, you'll also see the corresponding line numbers next to each item in the stack.) If you're in R, you can run `traceback()` immediately after the error.

Seeing the stack is useful for checking that the correct functions were indeed called, but it can suggest how to trace the error back in a logical sequence. But we can often debug faster with more information from RStudio's debugger.

Examining the environment

Let's pretend we have a group of people who need to be assigned random partners. These partnerships are directed (so person A may consider his partner person B, but B's partner is person F), and we'll allow the possibility of people partnering with themselves. Some code for this is in the file `BugFun.R`. (It's not terribly efficient code, but it is useful for this exercise.)

```
source("BugFun.R")
peopleIDs <- seq(1:10)
pairings <- AssignRandomPartners(peopleIDs)
```

Try running this a few times. We have an inconsistent bug.

We can use breakpoints to quickly examine what's happening at different points in the function. With breakpoints, execution stops on that line, and environmental states can be inspected. In RStudio, you can create breakpoints by clicking to the left of the line number in the `.R` file. A red dot appears. You can then examine the contents of different variables in debug mode. To do this, you have to make sure you have the right setting defined: Debug > On Error > Break in Code.

> **Exercise**
>
> Use breakpoints to identify the error(s) in the AssignRandomPartners() function. Go to `BugFun.R` and attempt to run the last line. Decide on a place to start examining the code. If you have adjusted your settings, the debug mode should start automatically once you define a breakpoint and try to run the code again. (If a message to source the file appears, follow it.) Your console should now have `Browse[1]>` where it previously had only `>`.
>
> The IDE is now giving you lots of information. The green arrow shows you where you are in the code. The line that is about to be executed is in yellow. Anything you execute in the console shows states from your current environment. Test this for yourself by typing a few variables in the console. You can see these values in the Environment pane in the upper right, and you can also see the stack in the middle right.

If you hit enter at the console, it will advance you to the next step of the code. But it is good to explore the Console buttons (especially 'Next') to work through the code and watch the Environment (data and values) and Traceback as they change. By calling the function several times, you should be able to convince yourself of the cause of the error.

Much more detail about browsing in debugging mode is available at here.

When you are done, exit the debug mode by hitting the Stop button in the Console.

In R, you can insert the function `browser()` on some line of the code to enter debugging mode. This is also what you have to use if you want to debug directly in R Markdown.

Programming Challenge

Avian influenza cases in humans generally arise from two viral subtypes, H5N1 and H7N9. An interesting observation is that the age distributions for H5N1 and H7N9 cases differ: older people are more likely to die from H7N9, and younger people from H5N1. There's no evidence for age-related differences in exposure. A recent paper showed that the risk of severe infection with avian influenza from 1997-2015 could be well explained by a simple model that correlated infection risk with the subtype of seasonal (non-avian) influenza a person was first exposed to in childhood. Different subtypes (H1N1, H2N2, and H3N2) have circulated in different years. H1N1 and H2N2 were the only strains before 1968, and H3N2 has circulated since 1968. In 1977, H1N1 reappeared and circulated with H3N2. Perhaps because H3N2 is closely related to H7N9, people with primary H3N2 infections seem protected from severe infections with H7N9. The complement is true for people first infected to H1N1 and H2N2 and later exposed to H5N1.

Of course, we do not know the infection history of any person who died from avian influenza. We only know the person's age and year of death. To perform their analysis, the authors needed to calculate the probability that each case had a primary infection with each subtype, i.e., the probability that a person born in a given year was first infected with each subtype. Your challenge is to calculate these probabilities.

The authors had to make some assumptions. First, they assumed that the risk of influenza infection is 20% in each year of life. Second, they assumed that the frequency of each circulating subtype could be inferred from the numbers of isolates sampled (primarily in hospitals) each year. These counts are given in `subtype_counts.csv`. [1]

The challenge: For every year between 1960 and 1996, calculate the probability that a person born in that year had primary infection with H1N1, H2N2, and H3N2. You must program defensively to pull this off.

When you have found the solution, go to `https://stefanoallesina.github.io/BSD-QBio3/` and follow the link `Submit solution to defensive programming challenge` to submit your answer (alternatively, you can go directly to goo.gl/forms/NjdZUyAjs9HBWMqL2.

[1] The counts are actually given for each influenza season in the U.S., which is slightly different from a calendar year, but you can ignore this. You'll notice that "1" and "0" are used where we know (or will assume) that only one subtype was circulating. The authors made several other assumptions, but this is good enough for now.