BIOL 4750/6750: Programming for Biologists

Course notes and handout

William D. Pearse

Department of Biology
Utah State University

October 2016

Preamble

Programming is more a skill than a discipline, and as such it requires a lot of practice and a lot of mistakes. Learning to program is like learning to cook: you can't make an omlette without breaking some eggs, and you will waste many more eggs while you're learning.

Learning to program is not the same as studying computer science. Computer science *is* a discipline, but many of its concepts are useful to a programmer. A cook is not a chef; a chef understands the theory and development of cuisine and uses that knowledge to develop new dishes. A cook with some understanding of basic culinary skills and terms will make better dishes, and can read more advanced cookery books.

The biggest challenge in learning to program is yourself. It is easy to get frustrated with your computer, but you're going to be a lot calmer and happier if you acknowledge one fundamental rule: *computers don't make mistakes, we do*. If your code doesn't run, it's the fault of you or the maintainer of the software you're using, not the computer that's blindly following your instructions. Except in TRON, computers don't have inner lives or secret motives.

If you learn to accept that you are the reason there is an error message, and that, fundamentally, it doesn't matter, you're going to be much happier and more productive. I make mistakes every day, and there is only thing that distinguishes you from me (right now): when something crashes and I realise I've made a huge mistake, I laugh and put the kettle on. It's never worth getting angry over. Consider taking a break in the middle of these programming exercises if you're struggling—we do most of our thinking with our subconscious brain, which can only help you when you're not frustrated.

This course will give you experience in programming, along with a reasonable grounding in basic computer science. The exercises and problems I have set you are difficult by design, and I don't expect you to get all of this right. Every concept comes back over and over again, and so if you're feeling frustrated I encourage you to look back at what you were doing a few weeks earlier. You'll find you understand it all now, because you've gotten better. If, at the end of the course, you understand perfectly the first two sessions and some of the third and/or fourth sessions, then you will be a fantastic programmer.

End of line.

Table of contents

Part I

An introduction to R

Session 1

Fundamentals

Overview

This is the first of four sessions in the 'Introduction to `R`' series. Over these sessions I am going to teach you the fundamentals of programming, using the language `R` as a guide. The fundamentals you learn in the next few sessions will recur throughout this course: if you can spot those recurrences, you'll get more out of the course. In this session we're going to learn the basics of `R`: the differences between a `vector`, `matrix`, and a `list`, subsetting, and basic plotting. This is the only session where I make heavy use of footnotes; I footnote material that I think is useful to know but potentially confusing when you're inexperienced. Learning a programming language is like learning a real language: if you're paying attention, the most commonly-used parts of a language are actually the most confusing. I've flagged important, but potentially confusing, things here for you to refer back to when you're feeling a bit more settled: it's possible to be conversant in French without understanding every grammatical rule, and the same is true of `R`[1].

I hate to ask this, but please *forget everything you may have been taught about `R` in the past*. The terms I am going to use to describe things (*e.g.*, *atomics*, *coercion*, and *scope*) often sound a bit scary, and so are rarely used in introductory courses. These are the correct technical terms for concepts that many programming languages share, and are the terms used by the writers of `R`—trying to map these onto the simplified versions of concepts you have been taught in the past can make your life more difficult. Learning a new language requires some sweat; resist the effort to ask for help the moment you encounter a problem. Calmly read any error message you encounter and see if you can figure out what it might be trying to say. Please do ask me anything you want, but my first question will almost certainly be "what is the error message" and my second question will therefore be "and what do you think it means"... Not knowing the answer to the first question will make you look rather foolish, but the second question is the hardest and is something I am happy to help you with. Remember computers are very literal: something like `Error: object 'result' not found` often means you typed '`result`', not '`Result`'. Finally, a general piece of advice: the bonus exercises at the end of each session in this section are, of course, not mandatory. That said, many of them will be useful to you at points in the future—if you didn't have time to complete them during the course, you may find going back over them in your spare time afterwards useful.

[1] So feel free to read these footnotes, but don't be concerned if they don't make much sense.

1.1 Atoms

Everything in R is an object, and the basic building blocks of objects are *atomics*. There are many types of atomic; the common ones are `logical` (`TRUE`/`FALSE`), `integer` (whole numbers like 1), `double` (floating point numbers like 1.0 and 1.1), `complex` numbers (imaginary numbers;), `character` (strings of letters "like this"), and `raw` (bytes; things computers read). We treat `integer` and `double` as `numeric` and you can essentially just pretend there's an atomic called `numeric` and forget about the distinction[2]. All atomics are `vectors`: this means you can have more than one `numeric` value inside a single atomic variable (like vectors in maths). However, not all vectors are atomics: the simplest example of this is `factors`, which are described in '*Compound data types*' below. We can concatenate vectors together to make longer vectors; thus two vectors of length two can be combined to make a single vector of length four.

```
#Anything after a '#' is a 'comment' and ignored by R
# - use them to write notes to yourself in your scripts
empty.vector <- logical(0)        # 1
numbers <- c(1, 3, 5, 3)          # 2
vector.maths <- numbers * 4.5     # 3
```

(1) Can be read out loud as "empty dot vector gets (a) logical (vector of length) 0". `<-` is the assignment *operator* ("gets"); an operator is a thing like + or * which takes an object and does something with it [3]. In this case it's taking the output from the *expression* (a set of instructions to be executed by the computer) `logical(0)` and storing its output in a new variable called `empty.vector`. (2) is just showing the concatenate function, which concatenates vectors, and (3) shows how you can do mathematical operations across an entire vector at the same time (which is useful). As with all the examples I'm going to give you in this course, *type them into your computer yourself* (copy-pasting won't build your muscle memory for the language) and *examine their results* by typing things like `numbers` into the console to see what the code does. I will say it again: *typing, not copy-pasting, my examples is one of your assignments*, and doing so will give you experience of what error messages occur when you mis-type, as well as a deeper understanding of the language. You will learn faster if you type. Trust me.

One useful feature of vectors is *subsetting*. The best way to understand subsetting, which is sometimes called *slicing*, is to see it in action. To demonstrate this, I'm going to use a built-in vector called `letters`—this *variable* (an object in R, like a vector, that contains a value) is available whenever you start R. Type each of the following lines into R and see what you get, then I'll go through what's going on in each line.

```
letters              # 1
1:5                  # 2
letters[1:5]         # 3
x <- 1:26            # 4
```

[2] Well, almost. Computers care a great deal about whether a number is an integer or a floating point value; maths on floating point numbers is much harder for computers and they have dedicated processors just for it. My advice is to ignore it (for now), but to remember that this is why sometimes vectors are printed as `2` and other times as `2.0`; R cares a great deal, internally, about what is going on. The brave can read David Goldberg's "What every computer scientist should know about floating-point arithmetic" (ACM Computing Surveys 23.1 (1991): 5-48) once the course is over; if you make it to the end please email me as I have a number of programming projects for you to work on!

[3] Technically it's a way of calling a function—worry about this once we've covered functions.

```
x < 10                  # 5
letters[x < 10]         # 6
names(x) <- letters     # 7
x                       # 8
x["a"]                  # 9
```

(1) shows the contents of `letters`—all the letters! 2 shows a shortcut for generating a `numeric vector` with all the numbers between 1 and 5 using the `:` operator. (3) shows *slicing*, which uses the `[` and `]` operators. It means "take the vector inside the `[]` and give me those elements of `letters`". In this case, that's the first, second, ..., fifth elements—the letters a, b, ..., e. (4) makes a new vector, and (5) shows us a `logical` vector created using that vector and the < operator (>, <=, ==[4], and `!=` do what you would expect too). Play around with different operators to get a feel for how they work. In (6), we see that we can also use a `logical` vector for subsetting [unlike the numeric in (3)]. (7) assigns *names* to a vector, whose effect is obvious in (8). Finally, (9) shows we can also use a `character` vector to subset a named vector.

1.2 Multiple dimensions and advanced subsetting

Vectors are uni-dimensional, but a `matrix` has two dimensions and an `array` can have as many as you want. You can subset each dimension of these data-types separately, using the `,` operator to separate them. These variables must all be of the same atomic type, however. When working with multiple dimensions, `R` expects `matrix[row,column]`, and leaving a dimension blank gives you everything along that dimension. It's `row,col`, so remember: in a healthy relationship, a `row` is followed by a cuddle; if having a cuddle leads to a `row`, something's going wrong. "Row" is the British word for "argument"; it took me years to come up with this mnemonic and I'm afraid that's the best I can do. See if you can figure out how these data types work by running the examples below; notice how I subset on the output of `array` without even bothering to save an intermediate variable in the last line.

```
my.mat <- matrix(1:25, nrow=5, ncol=5)
my.mat
my.mat[1:5,]
my.mat[,1:5]
my.mat[1,3]
my.mat[-1,]
my.mat[,-1:-2]
#...negative numbers exclude elements...
my.mat[1:2,1:2]
array(1:8, dim=c(2,2,2))[,,1]
```

[4] *NOT* =. = is sort-of-nearly the same as <-, so be careful not to type == instead of = by mistake. While we're here, don't use = instead of <-; in very specific circumstances the two do not do the same thing and it will cause you a headache. Use = *only* when giving function arguments (more on that in a later session), and <- all the rest of the time. If you want to know the difference between = and <-, ask me during the final session in this section: it's an "interesting" story and involves the history of R and the design of computer keyboards in the 1960s...

While we're here, let me mention my two favourite things to use in R: %in% and match. %in% is an operator[5] that tells you whether the elements of the vector on its left are in the vector on its right. match tells you where elements in a vector are found in another vector. By default, match returns NA if there is no match. NA is a special R value that flags what I think of as 'logic errors'; if you ask where something is in a vector and it's not there, you get NA (it couldn't return anything else as that might be interpreted as a location). What happens when you run NA + 3? NULL is a value you will encounter occasionally as well, and it means 'nothing' ("null and void"). It's also often used to flag problems as well, but put simply it could just be described as a thing that indicates the absence of a thing (R is sometimes very Zen). R also has Inf, whose meaning is obvious if you divide by zero. Run these lines to understand what's going on.

```
characters <- c("a", "f", "3")
#...Note that "3" is a character, and 3 is a number...
characters %in% letters
match(characters, letters)
letters[match(characters, letters)]
letters[!characters %in% letters)]
#...a '!' negates a logical...
"a" == 'a' #" and ' are essentially interchangeable
```

1.3 Compound data types

There are many kinds of vector, but the most important non-atomic type is the list. A list is special: its elements need not be of the same type. This means you can have a list made up of a numeric, character, and a logical vector. Like all vectors, a list can be *named*, which allows you to pull out individual parts of a list (see also subsetting, below). To do that, you need to use the $ operator, as below. You can also subset a list, as you can any vector; the convention is to use [[]] when pulling out a single element of a list, however.

```
my.list <- list(n=c(1,3,5), c=c("hello","world"), l=c(TRUE,FALSE,TRUE))
my.list$n
my.list$c
my.list[[3]]
my.list[1:2]
```

A data.frame is a special kind of list that you will use a lot in your statistical work (but not so much in programming). It lets you use the named features of a list, but also the subsetting features of a matrix (see subsetting below).

```
my.df <- data.frame(n=c(1,3,5), c=c("hello","world"), l=c(TRUE,FALSE,TRUE))
my.df$n
my.df$c
my.df[1,]
my.df[,2:3]
```

[5] See my other footnote about operators; behind the scenes it calls the function is.element for you.

The most dangerous compound data type is a `factor`. Most people think factors are atomics, and they're dangerously wrong. A factor is a special data-type that makes statistical analysis of categorical data easier and more efficient to work with, but looks like a character vector. It contains a vector of possible character states (*levels*) and an internal data vector which stores what number level an element of the vector represents. This matters when people *coerce* vectors of different types into one-another; coercion turns a vector of one type (*e.g.*, `character`) into another (*e.g.*, `numeric`). In factors the data vector is stored as a number, and this is what is coerced, often counter-intuitive results. Below I show how to do this (and check the type of a vector), and also show the solution to the factor problem. Try them for yourself: factors are very confusing to explain, but if you examine the variable `dangerous` below (and its *levels*, and what happens when you coerce it) you'll figure out what's going on. If you're ever going to do statistics in `R`, keep playing around with this until it makes sense because if you don't understand this you will make a fatal mistake in the future!

```
is.numeric(1:5)
as.numeric(c("1", "3", "5"))
as.numeric(c("one", "three", "five"))
dangerous <- factor(c(3, 1, 5))
levels(dangerous)
as.numeric(dangerous) #!!!!!!!!
as.numeric(as.character(dangerous)) #fine
```

1.4 Functions

You have already been using functions, and will soon be writing your own. For now, it suffices to say that every time you have written `(` or `)` you have been *calling* a function. For example, `is.numeric(1:5)` *calls* the *function* `is.numeric`, and gives it the object `1:5` as its only *argument*. That function then *returns* `TRUE`. When a function returns, it gives us something back, and a function can only give us one thing back[6]. In `R`, what happens in a function stays in a function: it cannot alter the variables we give it as arguments. "Calling" a function might sound a bit strange: imagine you're yelling for the function to come over, to listen to your instructions (the arguments), carry out the work (execute its code), and then give you back (return) its results.

Arguments are ways of us telling the function how to operate. For example, calling `sort(c(1,3,5,1))` is different from calling `sort(c(1,3,5,1), decreasing=TRUE)`. By *default*, `sort` sets `decreasing` as `FALSE`, but we can change that by altering the argument by name (as we did above) or by making use of the order of the arguments (*i.e.*, sort(c(1,3,5,1),TRUE)). It's generally better to name your arguments, as there's no way to know that `decreasing` is the second argument other than by looking the function up in the help files (more later). If this seems confusing, don't panic, because we'll cover it in detail in the next session, but try to remember the gist.

[6]It could, of course, return one list that contains multiple things in it, and this is common.

1.5 Everyday basics

This is a course in programming, not statistical analysis in R. That said, it is useful to know about the sorts of things that users spend most of their days doing in R.

The read family of functions (*e.g.*, read.csv, read.table) take in spreadsheets (in various formats, hence the suffixes), and return data.frame objects. The write family of functions (there is one for every read function) will output spreadsheets in different formats. These are what we will come to call *generic* functions later in the course, and as such there are different members of this family for different kinds of data like phylogenies, DNA alignments, spatial grids, etc. The matriarchs of these families are the write and scan functions, which provide more control over the input and output of data, but are best avoided when you're learning.

You will spend a lot of time plotting things, and most functions related to plotting will let you either supply two variables to plot or a single formula object. This sounds complicated, but it means you'll either type plot(x, y) or plot(y ~ x). The formula type is useful, and the best one to use, because you'll need it to run statistical models (lm(y ~ x)). Plotting functions have a common set of graphical paramaters, which are all described in ?par (see the help section below). The list of options is long, but it's worth taking a quick skim through them—don't try and memorise them all, but know that these options are there for when you need them. The image function lets you plot matrices; its friend contour can also be fun.

You can load additional *libraries* with new functions and features by typing something like library(pez)[7]—but you must first have installed that library either from the menu or using the function install.packages.

1.6 Getting help

Every function you will use in R has a help file associated with it. You can look up a known function by running ?name.of.function, or search for a term by running ??search.term. Use quotes if you want to use spaces or are searching for an operator, *e.g.* ?"%in%" and ??"phylogeny AND ecology". *Google is your friend*: typing what you are trying to do, or the error message you have found, along with helpful keywords (*e.g.*, R or R ecology) will often reveal the answer. Don't blindly trust that the first person you read on a mailing list knows the best answer, however, and take things like a user's rating on *StackOverflow* or whether the response is from a known R developer into account. Finally, *vignettes* are excellent story-like tutorial documents that most packages contain; running vignette(package='name.of.package') will tell you the vignettes available for a package, which you can open using the same function.

R help files have a standard format, and as such if you learn the patterns and components in a help file you're going to have an easier time finding out what a function does. Error messages that you can't explain are a common symptom of "I didn't read the help file"-itus... Help files contain:

- *Title & Details*—Basic overview of what something does, often as you would explain to a layman.

[7] The eagle-eyed will be confused that you don't have to write library("pez") here (although you can). Internally, library is coercing an *expression* (pez) into a character ("pez"). It's a lot of hassle just to save you two key-presses.

- *Usage*—Not what it sounds like; these show the arguments (see below) and are incredibly easy to use once you know how to write a function (see the next session).
- *Arguments*—Detailed descriptions of what each argument changes in a function, and what values they can take.
- *Details*—More detailed information about either the underlying science of a function (*e.g.*, "what is a regression?" or details of how the function was written where they affect the user.
- *Note*—Warnings of things that could catch you out
- *Value*—What the function returns
- *See Also*—Other functions you might like to use with this function, or possibly the actual function you were looking for...
- *Examples*—Often detailed, annotated examples of how to use code.

1.7 Exercises

Your first task is to go through everything above and make sure you understand it. This is the case for every single session in this course: you simply cannot go through the exercises without doing this because you won't know how to tackle them. Make your own vectors and subset those; make your own matrices and pull out parts of them.

Your second task is to build up your R vocabulary. An important part of using R is knowing what the different features of base R are, and how to use them. So you're going to build yourself a little vocabulary cheat-sheet. Make an R script with comments and code that explain how all of the variable below work. For example:

```
# c - concatenate
#     - takes two (or more) vector and joins them together
c(1, 2, 3)
c(c(1,2,3), c(4,5,6))
#     - they need to be of the same type, though!
c(1,2, "three")
```

Would be a good example. Once you've done this, push your cheat-sheet to GitHub following the instructions I gave in the lecture (and are listed in the appendix). Below are your vocabulary functions to look up in the help files and describe. Remember: some of them are very complex (*e.g.*, `library`), so be sparing with the technical details. The gist is enough!

- `cat`
- `cbind`
- `col`, `row`
- `cut`

- `diff`
- `dim`
- `rownames`, `colnames`, `names`
- `expand.grid`
- `eigen`, `%*%`, `lower.tri`, `upper.tri`, `diag` (try `?"%*%"`, not `?%*%`, as this is an operator, and document the function, not the maths
- `gl`
- `identical`
- `image`
- `library`
- `length`
- `jitter`
- `ls`; what does `rm(list=ls())` do?
- `mean`, `median`, `max`, `min`
- `paste`
- `read.csv`, `read.table`, `write.csv`, `write.table`
- `rnorm`, `pnorm`, `dnorm`, `qnorm`
- `runif`, `rpois`
- `rank`
- `sort`, `rank`, `order`
- `outer`
- `rep`
- `rowSum`, `colSum`
- `seq`
- `source`
- `which`, `which.min`, `which.max`
- `setdiff`, `intersect`, `union`
- `table`
- `with`; make sure you read the examples as this is a simple function whose technical explanation can be complex. The `data` argument to many plotting functions allows you to do something similar.

If you make it through all of these, go through the list of `par` options and make notes to yourself on at least five options you think you will want to use. Don't cheat: `cex` (which is very important...) is a family of options, so don't document all of them individually—by grouping them together you'll also learn something about the different parts of an `R` plot.

1.8 Bonus exercise: regular expression matching

Regular expression matching (`?regex`) allows us to find sections in a character string that match certain conditions. Trust me when I say that it's a skill you will end up using a lot, particularly in bioinformatics ("where is the TATA-box again?"). Read the help file on regular expressions, (`?regex`), then find the following items in the text below (bonus question: it's the opening of what book?). Once you've done that, use `strsplit` to cut the entire string into sections on each new line (what's the regex code for that?) and then repeat the exercise using `grepl` (which won't take long).

```
It was the best of times, it was the worst of times, it was the age of
wisdom, it was the age of foolishness, it was the epoch of belief, it
was the epoch of incredulity, it it was the season of Light, it was the
season of Darkness, it was the spring of hope, it was the winter of
despair, we had everything before us, we had nothing before us, we
were all going direct to Heaven, we were all going direct the other
way- in short, the period was so far like the present period, that
some of its noiosiest authorities insisted on its being received, for
good or for evil, in the superlative degree of comparison only.
```

- All instances of "the"
- What lines end with vowels
- What lines begin with "was"
- Where "it" is written twice in a row
- Where three vowels are written in a row inside a single word
- All the words that come after the word "the"

Session 2

Control flow and functions

Overview

This is the second of four sessions in the 'Introduction to R' series. Today we're going to focus on the fundamental building blocks of R: control flow and functions. Control flow means "loops", which some of you may have already heard of, and while you've been using functions a lot already we're now going to learn how to write them. These are the fundamental building blocks of programming: without them, frankly, you can't program at all.

2.1 Control flow

Controlling the flow of program execution is, essentially, all a programmer does. Most programs need to check whether some input data match some given criteria, and execute code if they do. This is carried out using an `if` statement:

```
value <- 5
if(value <= 5){
  print("Good news!")
}
```

Try running the above code a few times, changing the value of `value` and seeing what happens. There are two new things above: the `if` statement itself, and the `{ }` brackets. An `if` statement looks a bit like a function, and requires that you give it one expression (bit of code) that returns either `TRUE` or `FALSE`: a logical vector longer than one, an `NA`, or *anything* else will raise an error. So you can use something like `is.numeric`, `identical`, or most operators (*e.g.*, ==) in there. The `{ }` indicate what is called a *block*. A block is a group of lines of code (*expressions*, technically) that all get executed in a group. Giving `if` a block like this allows us to group more than one line together to be executed if we want. You will use blocks a lot in R.

Negating something (*e.g.*, `if(!is.numeric(x))`) is a useful trick, but sometimes you want to do one thing if something is true, and another thing if it isn't. `else` lets you do this:

```
value <- 5
if(value < 5){
  print("Less than five!")
} else {
  print("Greater than or equal to five!")
}
```

Again, play around with the example above to get a feel for what `else` does. You cannot use an `else` on its own. An `else` to come immediately after whatever block could be executed by the `if` statement, as `R` has no way of knowing that it should wait longer for you to give it an `else`. People often forget to put the `else` on the same line as the end of the first block—don't be one of those people!

You don't always have to use brackets to create a block; just one line of code that can be executed is, very technically, a block, and so you can often get away with writing things a bit more simply like this:

```
value <- 5
if(value < 5)
  print("Less than five!")
```

I don't advise doing this today, but in the future you will end up doing it because it saves time and space and makes your code a bit quicker to read. Remember to *indent* you code, as I have been doing, to show where the block is: `R` doesn't need it, but humans' eyes do and it will help you in the future. *Always indent blocks for readability*. You can also get an `else` to work without brackets if you follow the rules I gave above, but I wouldn't advise doing so because you will end up making a very nasty mistake at some point and it's confusing for others to read.

2.2 Loops

Computers are meant to automate repetitive tasks, and so we often want a computer to do something repeatedly for us until we are satisfied with the result. Consider the following:

```
value <- 0
while(value <= 10){
  value <- value + 1
}
print("Finished!")
```

This is our first `while` *loop*. This loop continuously executes a block with a single instruction until the *condition* (`value <= 10`) is no longer satisfied. Each time the block in a loop is run, we call it an *iteration* of the loop. These kinds of loops are quite dangerous, because it's easy to get the computer stuck in an *infinite loop* because the condition to *terminate* (finish) the loop is never met. If that seems confusing, consider (but do not run!) the following piece of code:

```
value <- 0
while(value <= 10){
```

```
  value <- value - 1
}
print("Finished!")
```

Will this code ever finish running? No, because `value` will never be greater than 10. If you ever start an infinite loop, click the red cross on the console and/or hit the "escape" key repeatedly. If you're doing something quite computationally intensive, that often won't work and you will have to restart R...

Because of this tendency to go on forever, most programmers prefer to use `for` loops wherever possible:

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

The `in` operator can look a little strange, but essentially the above means "take this block and run it *for* each element *in* `1:10`, starting with the first one". `i` is the traditional *loop index* to use, and if you write a loop within a loop (which is slow, but you will end up doing at least once) `j`, then `k` is preferred for those *nested* loops. Use those letters: it means you will be able to read other people's code more easily, and they yours.

It is common (and often best practice) in loops to use loop indices and not to loop directly over the vector you might be interested in. So, for example, while the following is fine:

```
for(each in letters){
  print(each)
}
```

...it's mostly preferred to write:

```
for(i in 1:length(letters)){
  print(letters[i])
}
```

This might seem strange, but otherwise it's impossible to modify anything in the vector you're *looping over*. After a while, you will become so used to seeing a loop index that it'll be strange not to see it. Compare the following loops, and you'll see the problem with modifying if you don't have a loop index:

```
unchanged <- changed <- c("a", "c", "e")
for(each in changed){
  each <- toupper(each)
}
identical(changed, unchanged) #No change has happened
for(i in 1:length(changed)){
  changed[i] <- toupper(changed[i])
}
identical(changed, unchanged) #Success!
```

2.3 Advanced looping

Sometimes you need to break out of a loop altogether—perhaps because you've achieved the result you were looking for. Consider the following:

```
value <- 0                          # 1
max.iter <- 1000                    # 2
goal <- 2                           # 3
for(i in 1:max.iter){               # 4
  value <- rnorm(1)                 # 5
  if(value > goal){                 # 6
    break                           # 7
  }                                 # 8
}                                   # 9
if(i == max.iter){                  # 10
  stop("Max iteration reached!")    # 11
}                                   # 12
```

In the code above, we are trying to draw a random number from a distribution centred at zero (the default), and we want to stop once we get a number above 2. A strange thing to want to do, but bear with me. Since we don't know when we'll get a number greater than 2, we keep drawing (at most 1000 times) until we find a number greater than 2. `if` we find a number greater than 2, we `break` out of the loop—we stop the loop continuing. `R` is really quite clever, and so when it sees the *keyword* `break` and it's inside a loop, it knows to break out of whatever loop it's in and carry on with the rest of the script. Play around with the bits where I define the goal (3) to see what happens when you set it to something that's very unlikely to come up in a standard Normal distribution (what are the defaults of `rnorm` again?).

What's interesting about the above example is how I've used `break` to set a maximum number of iterations. The loop will only proceed for however many iterations are defined on line 2; if that many occurs then by the time we get to line 10 `i` will be equal to `max.iter` and we will have hit our limit, so the program spits out an error (which we create with `stop`; 11). So here you can see how I've used a `for` loop to do something that feels a bit like a `while` loop (keep looking until you find something), but done it safely with a maximum number of iterations to avoid an infinite loop. Of course, if we found the right value on the last iteration of the loop, then my code would spit out a false error. How would you fix this (and, in the process, make everything much simpler)?

The final loop-related keyword, `next`, allows you to skip one iteration of a loop and move straight on to the next one. The following, rather contrived, script that will only print random numbers if they're above a certain threshold gives an example:

```
value <- 0
threshold <- 1
for(i in 1:10){
  value <- rnorm(1)
  if(value <= threshold){
```

```
    next
  }
  print(value)
}
```

Finally, a piece of advice that you can ignore for a while if you prefer. Above I write lots of things like `i in 1:length(x)` to figure out how long a loop should be. Often, people prefer using the `seq` function to do this, and will write something like `i in seq(x)`. This is slightly faster and somewhat easier to read, and so is very popular. It is, however, a Very Bad Idea. Consider the following:

```
empty.vector <- numeric(0)
for(i in seq(empty.vector))
  print("uh-oh")
```

Two things to note: you can skip the `{}` in loops as well, if you prefer (I often do), but more importantly this code loops even though there's nothing for it to loop across! If you read the help file for `seq` incredibly carefully, you'll see that this is not a bug, but rather a feature of `seq`. Its default behaviour is to return `1:length(x)` when `x` has more than one element, but to return `1:x` if `(length(x)==1)`. Try change the above so that `empty.vector <- 10` and see what happens. This can cause Very Bad Things that will take you hours to find the solution to. Luckily, the `R` developers have come up with a solution: `seq_along` and `seq_len`. You could just specify arguments to `seq`, but why not use those two functions instead: your code will be fast, safe, and (to be frank) it's a bit of a signal to others that you're a careful programmer who knows what they're doing. Writing careful, *defensive* code (like this) that catches potential errors from the user before they cause a problem is a Very Good Idea.

2.4 Functions

We've been using functions throughout these sessions, and it's natural to want to know how to write your own. Here is a simple example:

```
double <- function(x){
  doubled <- x * 2
  return(doubled)
}
double(16)
```

The function `function` (confusing, I know, sorry) tells `R` we are defining a new function, and we give it as argument all the arguments our new function will take. After that comes a block, which is often called the *body* of the function, and does all the work. Once we've done all the work in our function, we use the `return` function to spit out the value of whatever variable we give it. Thus, on the final line, when we call our new function, we get the number 32 spat out.

In R, all functions are *call-by-value*. This means that the arguments to functions are copies of the objects that were passed to the function, not the objects themselves, and so anything done to those copies doesn't affect the real objects themselves. This sounds complicated, so look at the following example:

```
x <- 4
double <- function(x){
  x <- x * 2
  return(x)
}
double(x)
print(x)
```

As you can see, the value of `x` hasn't changed, even though it has within our `double` function. This is because two variables called `x` very briefly existed within R: one inside the *scope* of the function `double` as it was called, and the other within your workspace. When the `double` function returned, it spat out the value of `x` within its scope and then everything within its world was destroyed ("*went out of scope*", which means essentially the same thing). Thus `x` within your workspace was never changed—only a copy of it was modified, and it doesn't matter in the slightest to it that another variable called `x` briefly existed in another scope. All R packages have their own scope; if you read on the Internet someone writing something like `pez::fingerprint.regression` the `::` operator if being used to mean "the `pez` package has a function called `fingerprint.regression`". If `pez` is *attached* (you've called `library(pez)`), then the `::` is unnecessary[1]. `:::` is sometimes used, and allows you access to something in a package even if the author of the package didn't *export* that function (they didn't intend you to use it; you will learn about this when you make a package later), and is best avoided if possible.

Writing functions serves a number of purposes. By breaking your work up into chunks, and giving those chunks names, it makes it easier to follow the flow of your code. It also means you write DRY code (*Don't Repeat Yourself*): the moment you need to do something twice in a script, write a function that does what you need and use that instead. You'll only have to write out that piece of code once (making your script shorter), it'll be clearer to see what your script is doing if you give your function a sensible name, and it'll make it easier to spot how your code fits together and speed it up or re-write it later. Once you write a function, and you know it works, you can forget about it. You can forget the details of that code (*abstract* away the details), and focus on the things that matter. I constantly rely on code that I wrote and unit tested (see below) years ago; I couldn't tell you how it works now, and frankly I don't care, and I'll never have to.

[1] *Attach* is both a technical term and a function. In some introductory courses you will be encouraged to use `attach`: it is a Very Bad Idea to do so. You are creating another scope, as described above, and so copying variables and making all kinds of nightmares for yourself as you (and the poor developers you rely on) can't easily distinguish between editting your data or a copy of your data. Use the `with` function or the `data` argument instead (see last session).

2.5 Arguments and invisibility

When you call a function, you can provide values for arguments using their names, by position, or go with the authors' defaults. This is easier to understand with examples, and below I show how to define defaults (look at the function definition) and then make use of default (a), position (b) and named (c) arguments:

```
change.text <- function(text, before="Will says", after="", upper=FALSE){
  text <- paste(before, text, after)
  if(upper)
    text <- toupper(text)
  return(text)
}
change.text("brush your teeth") # (a)
change.text("brush your teeth", "Will's mum says") # (b)
change.text("ALRIGHT MUM", upper=TRUE) # (c)
```

It is possible to specify that arguments can only be one of several options using `match.arg`:

```
change.text <- function(text, person=c("will","mum")){
  person <- match.arg(person)
  text <- paste(person, "says", text)
  return(text)
change.text("hi") # Default is first element
change.text("hi", "will") # Fine
change.text("hi", "dave") #Error!
```

It's also possible to *invisibly return*. This means that a function will only return a value if it's asked to store it into a variable[2] and so nothing will be printed in your `R` console. This is used a lot by plotting functions like `boxplot`, and gives you a way to give details to the user (*e.g.*, how many numbers are in each category) without bothering them with details they don't want.

```
bond.james.bond <- function(x) invisible(x)
felix.leiter <- function(x) return(x)
bond.james.bond(10)
secret <- bond.james.bond(10)
print(secret)
felix.leiter(10)
```

The code snippet above shows how to use invisible returns, and one more thing: it's possible to define a function on a single line. In fact, it's possible to use functions without even giving them names, and we'll learn about these so-called *lambda* functions soon enough.

[2]Sort of. It suppresses `print` being called on things that are returned into the workspace. How it does that is complicated and requires the function to be hard-coded into `R`; don't worry about the details.

2.6 Exercises

1. Write a loop that prints out the numbers from 20 to 10
2. Write a loop that print out only the numbers from 20 to 10 that are even
3. Write a function that calculates whether a number is a prime number (hint: what does `2 %% 3` give you?)
4. Write a loop that prints out the numbers from 1 to 20, printing "Good: NUMBER" if the number is divisible by five and "Job: NUMBER" if then number is prime, and nothing otherwise.
5. A biologist is modelling population growth using a Gompertz curve, which is defined as $y(t) = a.e^{-b.e^{-c.t}}$ where y is population size, t is time, a and b are parameters, and e is the `exponential` function. Write them a function that calculates population size at any time for any values of its parameters.
6. The biologist likes your function so much they want you to write another function that plots the progress of the population over a given length of time. Write it for them.
7. The biologist has fallen in love with your plotting function, but want to colour y values above a as blue, and y values above b as red. Change your function to allow that.
8. You are beginning to suspect the biologist is taking advantage of you. Modify your function to plot in *purple* any y value that's above a and b. Hint: try putting `3==3 & 2==2` and `3==4 | 2==2` into an `if` statement and see what you get. Using this construction may make this simpler.
9. Write a function that draws boxes of a specified width and height that look like this (height 3, width 5):
```
*****
*   *
*****
```
10. Modify your box function to put text centred inside the box, like this:
```
*************
*           *
* some text *
*           *
*************
```
11. Modify your box function to build boxes of arbitrary text, *taking dimensions specified in terms of dimensions, not the text*. For example, `box("wdp", 3, 9, "hey")` might produce:
```
wdpwdpwdp
w  hey  w
wdpwdpwdp
```
12. In ecology, *hurdle models* are often used to model the abundance of species found on surveys. They first model the probability that a species will be present at a site (drawn, for example, from a Bernoulli distribution) and then model the abundance for any species that is present (drawn, for example, from the Poisson distribution). Write a function that simulates the abundance of a species at n sites given a

probability of presence (p) and that its abundance is drawn from a Poisson with a given λ. Hint: there is no Bernoulli distribution in R, but the Bernoulli is a special case of what distribution?...

13. An ecologist really likes your hurdle function (will you never learn?). Write them a function that simulates lots of species (each with their own p and λ) across n sites. Return the results in a matrix where each species is a column, and each site a row (this is the standard used for ecology data in R).

14. Professor Savitzky approaches you with a delicate problem. A member of faculty became disoriented during fieldwork, and is now believed to be randomly wandering somewhere in the desert surrounding Logan. He is modelling their progress through time in five minute intervals, assuming they cover a random, Normally-distributed distance in latitude and longitude in each interval. Could you simulate this process 100 times and plot it for him?

15. Professor Savitzky is deeply concerned to realise that the member of faculty was, in fact, at the top of a steep mountain in the fog. Approximately 5 miles away, in all directions, from the faculty member's starting point is a deadly cliff! He asks if you could run your simulation to see how long, on average, until the faculty member plummets to their doom.

16. Sadly, by the time you have completed your simulations the faculty member has perished. Professor Savitzky is keen to ensure this will never happen again, and so has suggested each faculty member be attached, via rubber band, to a pole at the centre of the site whenever conducting fieldwork [3]. He assures you that you can model this by assuming that the faculty member, at each time-step, moves $\alpha\times$ distance-from-pole latitudinally and longitudinally (in addition to the rate of movement you've already simulated) each time-step. Simulate this, and see how strong the rubber band (α) must be to keep the faculty member safe for at least a day.

17. (If you finish early: Most, if not all, faculty members are not as young as they once were. See what effect the faculty member tiring (and eventually sitting down and giving up) would have on their likelihood of survival. What would happen if a faculty member panicked and walked faster through time?[4])

2.7 Bonus exercises: pre-allocation

Your loops will go a lot faster in R if you *pre-allocate* your output variables. For example:

```
loop.length <- 50000
t <- 1
for(i in seq_len(loop.length))
  t[i] <- 10
#...quite slow...
```

[3] You have been tricked: these exercises are teaching you some biology after all. The previous exercises involved simulating "Brownian motion", which also describes the movement of gas molecules (as discovered by Einstein). This is an Ornstein-Uhlenbeck process, which is commonly used in both macro-evolutionary biology and movement ecology

[4] By completing this exercise, you will have implemented all existing macro-evolutionary models, but not quite all models of animal movement. If you can think of a more realistic model of how a lost hiker might move through space, write it and then contact me on GitHub. You might be on to something.

```
t <- numeric(loop.length)          # (!)
for(i in seq_len(loop.length))
  t[i] <- 10
#...very fast!...
```

The only difference between these two bits of code is that in the second example I pre-allocated (!) the vector `t`. What happens when you add things onto the end of a variable (as we do in the first example) is `R` has to copy that variable every time it wants to extend it. This means that it takes longer and longer (iteration 1: copy a vector of length 1 into a new vector of length 2, iteration 1000: copy a vector of length 1000 into a new vector of length 1001...). Allocating all that memory at the beginning of the loop by telling `R` you want a numeric vector of length `loop.length` saves us all this copying. Make sure you always pre-allocate loops: go back over your answers to the exercises above and see if you can figure out a way to make some of them more efficient in this way.

2.7.1 Bonus exercises: advanced debugging

All programmers make mistakes, and the difference between a good programmer and a bad programmer is how quickly they can find them. Programmers call tools to help them find the slowest parts of their code *profilers*, and the tools to find *bugs* (mistakes, errors, etc.) *debuggers*. "Premature optimisation is the root of all evil" is an old programmer's saying; I would advise you to steer clear of profilers and the obsession with making your code fast, and instead focus on finding and fixing bugs. What follows is less of an exercise and more of a suggestion of a set of tools you should experiment with if you're feeling confident.

`R` has a fantastic all-round debugging tool: `browser`. `browser` stops the execution of your code wherever it is called, and gives you control of the `R` prompt again so you can inspect what's going on with your code, run new lines, etc. It also lets you execute the **n**ext line of code, **c**ontinue, **s**tep into a function call (open `browser` inside whatever function a particular line calls). You can run each of these commands, and a few others (check the help file) by just typing their first letter (*e.g.*, `n`) at the `browser` prompt. Be careful: if you have a variable called `n`, typing `n` and pressing enter will just run the next line; you have to `print(n)` (or pick a better variable name...). For example:

```
bad.sum <- function(x){
  output <- 0
  for(each in x){
    if(is.character(each))
      browser()
    output <- output + each
  }
  return(output)
}
bad.sum(1:5) # all is well
bad.sum(c(1,3,5,"this.will.crash"))
```

Run the above code and you will quickly get a feel for how this works. You can trigger `browser` to be called whenever an error is called by running `options(error=recover)`. `recover` will be triggered whenever an error occurs, then wraps up whatever happened just before the crash and give you the option of calling `browser` at whatever level of the call stack you choose. What is a call stack? Each time you call a function, it gets added onto a *stack*, and the functions and their scopes get resolved with the deepest calls first. A *stack overflow* is when too many calls are made for the computer to resolve them, and is something all programmers fear because it can, in other languages, use all your computer's memory and crash everything—hence the website of the same name is dear to many programmers. Stack overflows are, therefore, also problems for programmers who rely on recursion (which we covered in the bonus exercise of session two). Recursion, since it relies on functions, is another trick of the functional programmer's trade. Try running `bad.sum` without my `browser` line once you've turned on `recover`; this will also make it immediately obvious what a call stack is. `options(error=NULL)` will turn all of this off, by the way.

All this re-writing of functions to insert `browser` can get tedious. Sometimes you want to put a `browser` statement inside code in a package you're using; perhaps you're getting an obscure error message and you could diagnose it in a few moments if you knew what they were doing with your data internally (I do this a lot, and it saves me emailing people for help). `trace(name.of.function, edit=TRUE)` will open up file window with all the code for that function which you can edit at will, inserting a `browser` statement or doing whatever you want. You then just call `untrace(name.of.function)` and everything is set back to how it was before you edited the function. You can use this trick on a function in another package—but do remember just typing the name of a function spits out all its code for you.

Session 3

Classes and object-orientation

Overview

This is the third of four sessions in the 'Introduction to `R`' series. Today we're going to be focusing on what is called 'object orientation'—the use of *classes* to help simplify our programming. You've been using classes throughout the last few sessions without either knowing or really understanding what it means, and knowing will help make a few strange things in the wider `R` ecosystem of packages a bit clearer.

3.1 Philosophy: what is a class?

Right now, you're probably sat on a chair in a room of some sort. The first time you walked into that room, you had likely never seen that chair before, but you immediately knew what to do with it. You recognised the *object* in the room as a chair because it had some sort of recognisable characteristic (a *type-signature*), and realised you were looking at just another example or *instance* of the *class* of things we call chairs. This means you knew what kinds of things you could and could not do with that chair—the *methods* you could apply to it. You also recognised that this chair has certain properties (*slots*), like its weight and height, whose values help refine your general impression of what a chair is ('this chair is heavier than that chair'). Grouping things into categories like this allows us to save time and energy by not forcing us to figure out what we can do with each chair we see (can I sit on this one? can I drive it?). We recognise it's a chair and so just set about doing all the things we know we can do with a chair.

Object-oriented programmers think the above paragraph is a reasonable description of how human beings operate, and so they replicate all of that logic in programmatic form. An object-oriented programmer gives every variable (*object*) a class, and writes functions (*methods*) that can be applied to specific classes. Grouping things like this helps programmers save time and effort: a programmer writing a function that plots data can write separate *methods* for when the user calls that function with a `data.frame` or a `matrix` and let `R` figure out which function to call on behalf of the user.

R has a strange history with classes; they were something of an after-thought because R was originally modelled on what are called *functional* languages (more next session). As a result there are actually six different implementations[1] of how classes operate! Luckily for us, what are called S3 classes (remember that R is an open source version of the S language) are the most common, and are what we will focus on today. In S3, we write methods for a particular class (a method is a function that applies only to a particular class), but we have no formal way of specifying what properties (*slots*) a particular class should have. S4 and S6 classes overcome this limitation, but frankly are so complex that hardly anyone bothers with them. If you're interested, ask me, but I would advise you to avoid using them in your code: most users don't even know how to use the help files for such classes.

3.2 Dexter is an instance of a dog

The easiest way to learn about classes is by doing. So I'm going to walk you through an example of how to make a `dog` class, how to write methods for that class, and how we can hierarchically nest classes (*inheritance*; a `dog` is a `mammal`). Once you've done that, you're going to implement some classes of your own.

3.2.1 Class 'definition'

Classes don't have formal definitions in R: to make an object of a cerain class, you just have to alter its class attribute like this:

```
dexter <- list(length=60, weight=40, breed="mongrel")
class(dexter) <- "dog"
```

That last line was all we needed—run `class(dexter)` now to check if you don't believe me. There is no formal way, in S3 classes, to specify what slots an object of class `dog` *should* contain. I therefore suggest you always write out what slots you think a class should have so it's clear in your head, or (even better) write a `make.class` or `new.class` function (call it what you want) that will make something for you. For example, the following function will make a `dog` with the `weight` and `breed` slots I want:

```
new.dog <- function(weight, breed){
  output <- list(weight=weight, breed=breed)
  class(output) <- "dog"
  return(dog)
}
```

Not the world's most interesting function, but it's better than nothing. Most languages require that you write a function like the above, in which case it's called a *constructor*.

[1] More like five really, since development stalled on S5 and the code was later worked into S6.

3.2.2 Class methods

A *method* is just a function that we associate with a particular class[2]. When writing a method for a particular class, you need to check whether the user has provided you with the right kind of class in the first place. For example:

```
race <- function(first, second){
  if(!inherits(first, "dog") |  !inherits(second, "dog"))
    stop("You haven't given me two dogs!")
  if(first$weight < second$weight){
    print("First dog won!")
    return(first)
  }
  print("Second dog won!")
  return(second)
}
```

Obvoiusly this isn't a very sophisticated race simulator. Notice I used `inherits` to check whether each of the arguments the user gave *inherited* from the class `dog`. We'll learn more about inheritance in a few paragraphs, but you can just treat that line as a magic checker on the class of an object for now. Notice also that I used the `|` operator, which we learnt about last session in the exercises (along with its friend `&`). That's it! We've written our first method!

3.2.3 Generic fuctions

Somehow, our first method wasn't very impressive, and you're probably wondering what all the fuss is about. Enter the use of *generic* functions: functions that take essentially any kind of object, then *dispatch* that object to the correct *method* once they've identified the class of the object. Ever wondered how it is that you can `print` a `numeric`, `character`, and `data.frame`? What do we see when we type `print` into the computer?

```
> print
function (x, ...)
UseMethod("print")
<bytecode: 0x20fb580>
<environment: namespace:base>
```

`function (x, ...)` means it's a function that takes one argument and a load of others (via the `...` keyword; see below). `<bytecode: 0x20fb580>` means this is a 'compiled' function, which means its underlying `R` code has been translated into machine codeso it will run faster (more in the `C++` session), and `<environment: namespace:base>` means the function is defined in the `base` package (which `R` always assumes has been

[2]The technical definition of a method changes a little bit depending on what the programming language you're dealing with is designed to do. If you look up 'method' online you'll find a lot of different defintions, and it can be confusing. Things could get even more confusing when we move onto Python, but don't get too caught up on the details. If you don't see the difference between 'function' and 'method' then relax, it won't matter and isn't clear in `R` programming

loaded). All that matters for today, however, is that `UseMethod("print")` means this is a *generic* function: this function is a really just a placeholder to remind R to look for the correct `print` method to call for the class that `x` is.

Let's define a method for class `dog` and see what happens. Make sure you try and `print` an instance of class `dog` to screen (*i.e.*, type the object into the console) before defining this:

```
print.dog <- function(x, ...){
  cat("This ", x$breed, "weighs ", x$weight, "kg\n")
}
```

That's it! Now you've defined that method, try printing a dog object again. All R did was look for a function called `print.class_name` (in this case, `print.dog`) and called that function. Note that I made sure the arguments of the two functions matched perfectly. Generics often use the `...` keyword to give you more flexibility in what you're writing. `...` grabs hold of any arguments the user gives that aren't defined by a function. It's possible to loop through those arguments using something like `list(...)`, but it also means that you can write code like the following:

```
print.dog <- function(x, verbose=FALSE, ...){
  if(verbose==TRUE){
    cat("This ", x$breed, "weighs ", x$weight, "kg\n")
  } else cat("Woof!\n")
}
```

Without R freaking out that `print.dog` has an argument `verobse` and other methods (like `print.numeric`) don't. By now you've hopefully figure out what `cat` and do, but you probably have one more question: how can I write a generic of my own?

```
askdog <- function(x) UseMethod("askdog")
askdog.default <- function(x) return("Woof! No-idea! Woof!")
askdog.numeric <- function(x) return("Woof! A number! Woof!")
askdog.dog <- function(x) return("Woof! A friend! Woof!")
```

Here I define a new generic function (`askdog`), give a fall-through *default* response, and then two somewhat pointless methods for two classes. In S3 classes, you can't dispatch on more than one argument, so writing a new `race` function that would check to see what class of thing a dog was racing would end up being a bit fiddly. Note that functions can be defined on a single line; typing `function(x) return(3)` is a perfectly valid (boring) function (does `function(x) 3` work?). We'll be using more one-line functions next time.

3.2.4 Multiple inheritance

Dogs are mammals, and they're also animals; chairs can be made of leather or plastic. Sometimes it's useful to recognise that objects can be of more than one class, and multiple inheritance lets you do that. It's simple to implement:

```
dexter <- list(length=60, weight=40, breed="mongrel")
class(dexter) <- c("dog", "mammal")
brian <- list(length=30, weight=40)
class(brian) <- c("mongoose", "mammal")
print.mammal <- function(x, ...) cat("What a great mammal!\n")
```

R will work its way through the `class` attribute of `brian` and `dexter`, stopping when it finds something, or falling through the `default` if it doesn't. Hence you were able to see the contents of `dexter` before you defined a `print.dog` method, and despite us not having a `mongoose` class you can `print brian`. This is also the reason we use `inherits(x, "class")` instead of `if(class(x)=="class")`, because if an object has more than one class we'll get an error. Try it out; it's common to accidentally check a long vector in an `if` statement so you should get used to that error.

3.3 Exercises

Apart from the first exercise, these exercises are intended to show you the value in building up complexity by first implementing basic classes (a series of points) and then working up from there.

1. Implement a `cat` class, complete with `race` and `print` methods.
2. Implement a `point` class that holds `x` and `y` information for a point in space.
3. Write a `distance` method that calculates the distance between two points in space.
4. Implement a `line` class that takes two `point` objects and makes a line between them.
5. Implement a `polygon` class that stores a polygon from `point` objects. *Hint: a polygon is really just a load of lines.*
6. Write `plot` methods for `point` and `line` objects.
7. Write a `plot` method for a polygon. *Hint: if this isn't trivial, you're doing something wrong*.
8. Create a `canvas` object that the `add` function can add `point`, `line`, `circle`, and `polygon` objects to. Write `plot` and `print` methods for this class.
9. Implement a `circle` object that takes a `point` and a radius and stores a circle. Don't make a circle out of lines!
10. Write `area` generic methods for `circle` and `polygon` objects.
11. Add support for `circle` objects to your canvas.
12. Add a `summary` method for `canvas` that calculates the height and width of the canvas, the fraction of the canvas covered in filled-in polygons and circles (if appropriate), and average distance between any points on the canvas (if appropriate).
13. Add *optional* colour support for your objects.

14. Professor Savitzky wants you to update your simulations from the last session to use your new classes. This is your first time *refactoring* code (changing it afterwards to make it more streamlined). It's very hard to do! Is it easier to write all your code again from scratch (it may not be)? If so/not, why?

3.4 Bonus exercises: unit testing

It's good to get into the habit of writing *defensive* code. Defensive code is code which checks on user input and on the results of its own calculations to make sure that nothing's going wrong. You've seen examples of it throughout this course: using *for* loops to avoid infinite loops, checking the class of objects using `inherits`, and checking for problems and then raising errors using `stop`.

But there's a more insidious kind of error that can creep into programs. Sometimes, you make a mistake in your code and you get the wrong kind of answer—maybe the code runs fine, but you got the equation wrong. There's no way to be sure you won't make these kinds of misakes, but *unit tests* are a good way to look out for them. The basic principle is you explicitly state what you think your function/class should do, then you check your code to make sure it does the right thing. Ideally, you would write those tests before writing your code and then write code that fulfills those tests (*test-driven development*), but this is much easier to do in a tech company than in a scientific lab because we don't always know the answer beforehand! Here's an example (note we have to load a package):

```
library(testthat)
double <- function(x) return(x*2)
expect_that(double(2), equals(4))
expect_that(double(NA), equals(NA))
expect_that(double("two"), throws_error("non-numeric argument to binary operator"))
```

Nothing will happen when you run the above, because everything runs as expected. Play around with it to see what happens when `expect_that` doesn't get what it wants.

Unit tests are really useful when you're writing complex pieces of software, because you can write tests for all the individual units and so reasonably certain that they work properly. As with everything in programming, you start from a sure base (simple functions with tests), build more complexity on top of that firm foundation, and then write tests for those complex functions too. They're particularly useful when you're working with large groups of people, because you can demand that all the tests that you've written must still pass: they can do whatever they want to your code (*e.g.*, speed it up) but those tests represent your statement to the world about what you need.

Most programmers write their unit tests in separate files, which they they check using `test_file` or `test_dir`. You can see an example of such a test directory in one of my packages (https://github.com/willpearse/pez/tree/master/inst/tests). I love unit tests, and use them all the time. But there is a trend in biological computing to assume that unit tests substitute for good, quality programming. They do not (https://willeerd.wordpress.com/2014/04/07/kill-your-unit-test-fetish/).

Your bonus exercise is to write unit tests for all today's exercises!

Session 4

Functional programming

Overview

This is the last of our sessions in the 'Introduction to R' series. Last time, we covered classes and how they are somewhat strangely implemented in R, because R is really a *functional* programming language. In this session, we're going to learn what that means, and how we can use it to our advantage in writing fast, efficient code. Thinking functionally is something that doesn't come easily to many programmers—if you find this session particularly difficult, that's OK. You may find that going over some of the concepts in this session will be useful to you in the future, particularly the bonus exercises which cover advanced debugging techniques that, while complex, I couldn't live without.

4.1 Philosophy: functional programming

Maths is all about functions. Functions are perfectly reproducible: if you give the same function the same input parameters, it will always produce the same result. In mathematics, equations and numbers don't really have any kind of internal *state*: 3 is 3 and will always be 3; numbers don't have modifiers—there's no such thing as a "small 3" or a "rapidly changing 3". If we change 3 in some way then the number we get out is not 3, it's a new number. Critically, we can refer to the process by which we got that new number and the number that we get at the end of that process in exactly the same way because all functions are perfectly reproducible. Thus in conversation $3+2$ and 5 are interchangeable and identical, and we can use whichever representation is most convenient to us.

Functional programming relies on all of the above to give us faster code execution and different ways of thinking about problems. Because there's no difference between the function we're going to execute $(3+2)$ and the result (5), our computer can use *lazy evaluation* and delay the calculation of results until the last possible moment when we definitely need them. Lazy evaluation often happens behind the scenes, but sometimes you can see it in action when code that doesn't need to be run isn't even checked for sense by R:

```
my.fun <- function(x){
  return(x)
  this.nonsense.line(is, lazily, never.evaluated)
}
```

This also allows your computer to distinguish between *mutable* (changeable) and *immutable* (constant) types. Really, R handles all of this behind the scenes for us, but by breaking our problems down into sub-problems whose results aren't dependent on one-another, we can execute our code in *parallel* (on multiple processors at the same time) without doing any extra programming. This is fantastically powerful, because most laptops have 2–4 CPUs (main processors), hundreds of GPUs (graphical processors; we won't be covering those today as they're a lot more complex), and most server computers (used in High Performance Computing, or HPC) have at least 24 CPUs. In fact, getting parallel processing to work without thinking functionally is so complex in R that I'm not going to show you how to do it, and wouldn't advise you to even try.

4.2 First functional steps: the `apply` family

The `apply` family of functions allows you to take an input vector of some kind (*e.g.*, a `list` of vectors, a `data.frame` (which is really a list, remember) of columns/vectors, etc.) and *apply* a function to each element of that vector. The names relate to either the kind of input the functions expect, or the kind of input you'll receive at the end. If you bear that in mind, it's actually quite simple. Ignore what you will read on the Internet and focus only on what's below: this is a topic about which the ignorant shout loudly.

4.2.1 One input vector: l, s, and v

```
input <- list(1:10, letters[1:6], c(TRUE,FALSE,TRUE))
lapply(input, length)
```

Here we took the function `length`, and applied it to the three vectors within `input`. Simple! There are three members of the apply family that work exactly like this, and are distinguished only by their return type. You can run all three of these functions on any vector (including lists, `data.frames`—whatever).

- `lapply`—returns a **l**ist.
- `sapply`—returns a **s**implified version of whatever return value it gets. So if every element of our input returns a vector is of length one when the function is applied to it, you get a vector. If everything returned is of the same length and type, you get a matrix. Otherwise, you'll probably get a list (sometimes it can still be a bit cleverer).
- `vapply`—returns a **v**ector of whatever type you specify. It's just like `sapply`, only by specifying what you want back it's quicker and slightly safer because it complains if it can't simplify to what you wanted (*i.e.*, you made a mistake).

4.2.2 More than one input: m, a, and t

mapply is a really useful, but rarely considered, function. It takes two or more inputs, and applies a function to both of them.

```
mapply(paste, letters[1:3], letters[2:4], letters[3:5])
```

It uses ... so that you can provide any number of input variables, and as such the function has to come first. Fantastically useful, and fantastically simple.

Think of apply as the mother of this function family. Its intended to be applied over any array (it doesn't follow the family naming scheme because aapply would sound silly). For its purposes, a matrix is just an array with only two dimensions. Because these objects have dimensions, you need to specify the dimension along which you want to slice your input and apply your function. Use my row–cuddle mnemonic to remember that the rows of matrices are dimension 1, then the columns, and then... however many dimensions you have in an array.

```
input <- matrix(1:4, 2)
input
apply(input, 1, sum)
apply(input, 2, sum)
```

People on the Internet get very confused about data.frames and the apply functions, because either they weren't taught properly or they weren't paying attention when they were. In our first session, I told you that *a data.frame is just a list*. This means you cannot use apply on a data.frame, and if you try you will often get the wrong result. A data.frame is really just a list and so use sapply, or lapply to apply a function to each of its columns. People on the Internet scream "it doesn't make sense": it does, because you can't apply something to the rows of a data.frame as the columns could be completely different data-types. Consider: how could apply mean to a data.frame with categorical and numerical data? The whole thing makes perfect sense if you just remember that *a data.frame is just a list* that someone called `class(data.frame) <- "data.frame"` on for you.

The last member of the family is tapply. The help file is tremendously unhelpful; it takes an input vector that you want to group in some way (according to another vector), and then applies your function to those groups. This is tremendously helpful if you want to summarise data, for example to make a bar graph.

```
input <- 1:10
grouping <- rep(letters[1:2], 5)
tapply(input, grouping, sum)
```

4.2.3 Why apply (and why not aapply)

What is the purpose of these families? They're incredibly fast. You could write a loop that would achieve everything you can do with the apply family, but it would be much slower[1]. R is a functional language, which

[1] If you did the bonus exercises in our functions session, you'll have noticed there is no need to pre-allocate anything with the apply family. This is part of the reason.

means that once things are in vector form and you tell it to apply something to all elements of that vector, it can use fast, internal, pre-compiled code (remember bytecode from the last session?) to do everything much quicker than you could in pure R. There are, of course, some times that you can't use apply functions, perhaps because you're doing something at random (although do check the `replicate` function). That's fine, but every time you think functionally and break your problem down into elements that can be written as some sort of vector or matrix, you can use an apply function and everything will be much faster.

If you Google around about these functions, you will quickly notice that there is a package called `plyr` (and many others that are members of the "*tidyverse*") that makes it much easier to move things from one type to another. For example, you can take an `array`, apply a function to it, and get a `vector` or a `data.fame` back using the `aapply` and `adppply` functions respectively. "Advice is a form of nostalgia; dispensing it is a way of fishing the past from the disposal, wiping it off, painting over the ugly parts and recycling it for more than it's worth". My advice to you, when programming, is to avoid using these functions at all costs. These tools were designed to be used, once, to get everything into the correct format to start statistical analysis. Yet users seem to love using these functions to repeatedly mangle data between formats. Programmers realise that changing the format in which data is stored involves computationally expensive copying, and that all of this can be avoided if they just thought about their problem in the first place. I rarely see a user of these functions who is not using them constantly because they never took a moment to consider how their data should be structured, and instead engage in thoughtless, purposeless, and expensive movement of data that often harms their analysis. Better to have one sharp tool that you can use well than a hundred that are blunt.

4.3 Lambda calculus and parallel execution

Lambda calculus (λ-calculus) is a formalised way of writing mathematical statements about functions (don't worry we won't be writing any equations). In λ-calculus, functions are just as real as numbers and so they can be manipulated and transformed. What makes R so exciting and powerful is that it was designed to operate in exactly this way—R is blisteringly fast when you take advantage of this design feature, and can be agonisingly slow when you don't.

Functional programmers like to use *lambda functions* (sometimes called *anonymous functions*)—functions that aren't given a name that you can call. You don't need to learn anything new to use them, because in R functions are *first-class objects*—you can treat them and examine them as you can any object (try calling `str(ls)` to examine the `ls` function, or `print(ls)` to see its code[2]). All you have to do to use a "lambda function" is just not assign the function to a variable using `<-`.

```
input <- 1:10
grouping <- rep(letters[1:2], 5)
tapply(input, grouping, function(x) sd(x)/sqrt(length(x)))
```

[2] You probably don't realise, but very few languages let you do this! You can even modify functions on-the-fly (see the bonus exercises).

Simple! I didn't give my lambda function a name, but if I did it would probably be `std.err.mean`. And, because functions are first-class objects that can be manipulated, I can even pass around functions as if they were objects, use them, and even *manipulate* those functions!

```
safe.apply <- function(x, func){
  not.nas <- Negate(is.na(x))(x)
  x[not.nas] <- func(x[not.nas])
  return(x)
}
safe.apply(c(1,3,5,7,NA,1), function(x) return(x*2))
```

In the snippet above I define a func ion (`safe.apply`) that takes an input vector (`x`) and applies an arbitrary function (`func`) to all non-`NA` members of `x` using a manipulated form of `is.na`: `Negate(is.na(x))`. I then call my function using a lambda function, just for style (note that, thanks to scoping, I can use `x` as an argument for both functions). This was a somewhat contrived example (I could have just written `x[!is.na(x)]`), but I want you to learn `Negate` as it's useful when using functions like `Filter`. If you can get familiar with this kind of stuff you will find your code runs a lot faster, and you can write much more *expressive* code (code that gets more done with fewer lines).

You will also find it easier to run code in *parallel*: to give instructions to multiple processors simultaneously[3]. Parallel computing is easy in `R`:

```
library(parallel)
input <- list(1:10, letters[1:6], c(TRUE,FALSE,TRUE))
mclapply(input, length, mc.cores=2)
#...which is the same as...
mcMap(length, input, mc.cores=2)
```

That's it! Done! `mclapply` is exactly the same as `lapply`, only it uses as many cores as you specify in `mc.cores` to get your job done using as many processors as you specify. Easy! Note there's also an `mcmapply` (you can guess what that does), and stick-in-the-muds like myself prefer to use `mcMap` (the parallel equivalent of `Map`) which is essentially identical to `mclapply`, only the name ("map") is more similar to the terminology used in other programming languages. `mcMap` *et al.* will try and auto-detect how many cores your computer has, but in my experience you don't want to trust this as it depends on your administrator setting up your computer correctly (I am my own administrator, hence my lack of faith).

[3]Technically, what we will be doing is asking the computer to open multiple *threads* of execution, which you can think of as trains of thought. Processors are only able to run one thread at a time, but modern operating systems are capable of scheduling the execution of multiple threads so it looks like many threads are running at once. This is how you can have your music player open while you're writing an email; it's tremendously complex, and we're only able to handle this in a trivial fashion because the immutable nature of our functional code means we don't have to worry about what would happen if multiple threads needed access to the same part of memory at the same time.

4.4 Exercises

Not all of these require you to use the functional programming techniques we have covered today, but you will find them easier to complete if you do.

1. `replicate` is an important function that allows you to quickly generate random numbers. Use it to create a dataset of 10 variables, each drawn from a Normal distribution with different means and variances. This can be achieved in *one line*.
2. Make your own version of the `summary` function for continuous datasets (like the one you generated above). You don't have to slavishly replicated `summary.data.frame`; write something you would find useful.
3. Write a `summary` function to summarise datasets containing only categorical (...`!is.numeric`...) data.
4. Finally, make a `summary` function capable of covering both kinds of data. Hint: if your function doesn't call the functions above, you're likely doing it wrong.
5. A molecular biologist you owe a favour approaches you with a problem. They have a DNA sequence (*e.g.*, 'ACGATATACGA') that they need to group into codons (groups of three) and translate into proteins (ignoring all complexities of translation and transcription). Write them a function that will take an arbitrary input sequence and an arbitrary codon lookup table, and output the translated sequence. Hint: `expand.grid` will help you make a demo lookup table.
6. The molecular biologist now asks if you would write a function that will take multiple sequences, translate them, and then flag where the sequences match-up (overlap).
7. One more thing: could you also write a summary-type function that would report percentage overlap across sequences?
8. The molecular biologist's advisor has shouted at them for ignoring start-codons and stop-codons. Modify your function from (5) to cut off all bits of the sequences before (and including) the start codon, and then chop off everything after (and including) the stop codon. The advisor is certain you've re-used your code from (5) for (6), so this will help with that function too, right?
9. Given recent events, Professor Savitzky has decided to keep more thorough tabs on the people (graduate students, post-docs, and faculty members) and resources (lab equipment, chemicals, and computers) in the Department of Biology. His friends in other departments would like something similar. He asks you to make a `list` with two slots: `people` and `equipment` that can store each of the above kinds of information for departments. Hint: you will probably want to make three classes: `department` (the list), `person` (with slots for name, title, and (optionally) advisor) and `equipment` (with slots for owner, type, and kind). Make sure you write print and summary methods for departments.
10. Professor Savitzky needs to present information about the department to the provost, and asks you to write a function that will calculate how many pieces of each type of equipment every member of faculty owns. Write a separate function that calculates the people who work for each member of faculty (remember that people aren't equipment).

11. The provost is very happy at how the Biology Department has grown, but is concerned that too many members of faculty have too many students and pieces of equipment. Write wrapper functions that let you add new staff and equipment to the department, assigning them to faculty members. Write the function such that if a member of faculty asks to have more than five pieces of equipment or four employees, the function won't let them.

12. Don't tell the provost, but modify your function such that it gives the option of promoting a post-doc to a member of faculty or throwing away a piece of equipment to give a faculty member more "space". Warning: this last exercise is harder than it seems, and will require you to learn to handle user input (`readlines`).

4.5 Bonus exercises: Recursion

Recursion is a neat trick to have up your sleeve, and is a major component of functional programming (though is often avoided in R; see below). There is an old saying about recursion: "you'll understand recursion when you understand recursion". Recursion is the use of a function to calculate the results of that function. It sounds like something from a Christopher Nolan film, but it's actually not too bad: you simply *recurse* through whatever arguments a function is given until you hit the *edge condition*. The edge condition is an end-point that you've defined: as long as the function always recurses to at least one edge condition, your function will terminate.

An example helps. What is the factorial of 5 (often written as 5!)? Well, it's 5 multiplied by the factorial of 4. What's the factorial of 4? Well, it's 4 multiplied by the factorial of 3. We would recurse on until we got to the factorial of 1, which is 1. We can use this to write a factorial function:

```
factorial <- function(x){
  if(x == 1){
    return(1)
  } else {
    return(x * factorial(x-1))
  }
}
```

If you can help it, try very hard not to use recursion as it's very slow in R. However, sometimes it's a life-savers, and in other languages it's can be very fast (*e.g.*, Haskell and Lisp). If you must do it in R, use the `Recall` function to warn R that you're doing it and save a little time. I give an example below, and also manage to drop the `else` statement. How do I do that?

```
factorial <- function(x){
  if(x == 1){
    return(1)
  return(x * Recall(x-1))
}
```

Now try to write functions that do the following recursively and non-recursively in R:

1. Calculate the n^{th} Fibonacci number
2. Find the largest number in a list
3. Sum a list of numbers
4. Is a number prime (you've already solved this one non-recursively...)?

Part II

R-World

Session 5

Terrain

Overview

This is the first of four sessions in the 'R-World' series. Over these sessions we are going to be programming a simulation of a world, starting with simulated terrain (complete with mountains, rivers, and lakes), adding multiple species of plants (that will grow and compete for space), herbivores (that will eat and sometimes kill the plants), and finally packaging everything together as an R package. All of the biology underling these simulations is loosely based on chapter four in the third edition of 'Theoretical Ecology: Principles and Foundations' edited by Robert May and Angela McLean. The purpose of these exercises is to give you an idea of how a real programming problem is approached, and to show you how your developing skills can be applied to approach problems of real interest and importance in biology (specifically, meta-population ecology). I'm not going to hold your hand: there is less to cover in each session, but the material is harder, and my intention is for you to *experience what real programming is like*. You will get stuck, you will get confused, and you will push through and eventually the heavens will open, it will all make sense, and it will all work! I promise you it will all work eventually! For this session, we're going to focus on producing the *terrain* that will make up our world: simulating a landscape of heights using the *diamond-step algorithm*, adding lakes, and then (optionally) adding some rivers to our terrain.

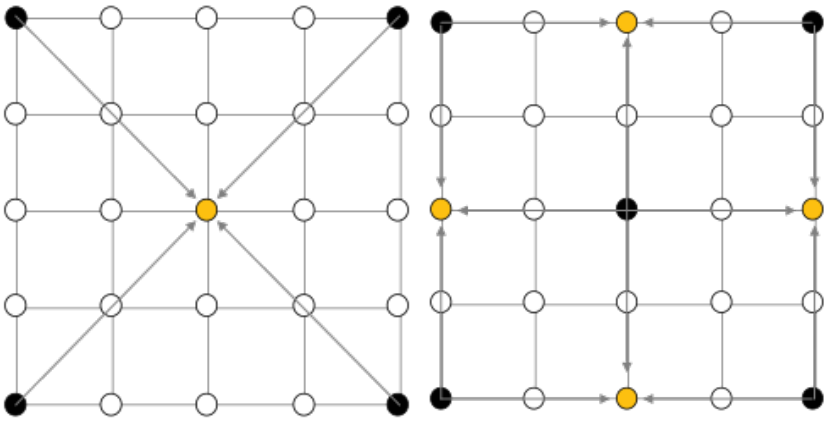
5.1 Terrain and the diamond-step algorithm

R-World is made up of grid cells (the elements of a `matrix`), and its basis will be its terrain. This will be a `numeric matrix` whose elements represent the height of a grid-cell above (or below, if negative) sea-level. Cells whose value are `NA` will represent cells filled with water. When we simulate plants and animals later, each cell will be able to hold at most one plant and at most one animal.

We could laboriously devise a landscape for each simulation, or use an equation to generate each landscape, but that would be no fun and wouldn't allow for a lot of variation. So we're going to *implement* (write) an *algorithm* (a set of instructions to carry out a task; we'll learn more about them later) that makes them for us. We can describe the algorithm using *pseudo-code*:

1: Make a square matrix with odd dimensions
2: Pick starting heights for four corners
3: **repeat**
4: DIAMOND-STEP(matrix)
5: SQUARE-STEP(matrix)
6: **until** Matrix filled with values

This pseudo-code doesn't make much sense, however, until we know that a `diamond-step` or `square-step` is. Figure 5.1 outlines each of these two steps; go through it and the figure legend now. The names of each step can be somewhat confusing; focus on what they do, not what they are called. **Before reading on, write on a piece of paper the basic steps you would take to implement this algorithm.**



(a) Diamond step (b) Square step

Fig. 5.1 **Overview of diamond-square step algorithm**. In the diamond step (a), the values of the four corner cells of the matrix (in black) are averaged to produce a new value for the central cell (in yellow). A small amount of noise (a random number) is added to this new value to give variety to the landscape. In the following square step (b), the centres of the edges of the square from the previous step (in yellow) are filled in with the averages of the cells above, below, and to the sides of them (in black). As before, a small amount of noise is added to these values. This process is then repeated; in the square step we have filled in four squares that then each have the diamond-step (a) applied within them, and so on, and so on, until the entire grid is filled. Each time the algorithm is repeated, the amount of noise added is reduced to ensure that changes at the beginning of the algorithm have a bigger impact. This produces a more pleasing effect, and (in our case) makes a more realistic landscape. You can experiment for yourself! Modified from Christopher Ewin—https://commons.wikimedia.org/w/index.php?curid=42510593

To crack this algorithm, you're going to break it down into manageable chunks. This is an approach we will use time and time again. You will write two functions, called `diamond.step` and `square.step`, that will carry out those two parts of the algorithm. You'll then write another function, called `diamond.square.step`, that will call each of those two functions in turn until the entire matrix is filled up. `diamond.square.step` will have to *initialise* everything: seed the matrix with starting values, create the matrix, etc.

There are lots of different ways of writing these functions. The first, and perhaps the easiest, is to write two functions that expect to be given a square matrix, and will simply carry out each step on that matrix. Your `diamond.square.step` function will have 'step through' your matrix, calling those functions with smaller and smaller chunks of the matrix as the algorithm progresses. The second way (which is, for some reason, the approach I chose) is to use the inherent *spacing* in each iteration of the algorithm and write functions that apply

the diamond and square steps to each cell within a spacing of a given size (in figure 5.1 the spacings are 4, for example). You pass the `square.step` and `diamond.step` functions the matrix and the spacings, and they fill in all the steps for each spacing along the matrix. This makes `diamond.square.step` a lot simpler to write.

People's minds work in different ways: the first method is the simplest, makes the most sense to most people, and is probably the fastest, so if the second method makes very little sense don't worry about it and just write the first one. *Don't forget* that you need to add a little bit of noise (use `rnorm`) with each iteration, and that the noise (the `sd` argument) should decrease with each iteration. Remember that `image` will let you plot out a matrix.

5.2 Just add water (and documentation)

We're now going to write a function called `make.terrain` that's going to act as a *wrapper* for our diamond-step algorithm. The purpose of a wrapper is to 'wrap-up' all the complex things that we don't want our users to worry about (*e.g.*, starting values for the matrix and changes in `sd`), making it easy for a non-specialist to use the function. It will also give us a convenient place to give the option to make our terrain look a little more realistic by adding in water. That part is simple—give the user the option to, if they wish, make all the negative heights water-logged by replacing those values with `NA`s.

Now, of course, you have been commenting your code throughout the last few sessions (*haven't you?*), but now I want you to start being more thorough and systematic with what you're doing. Using a systematic commenting style makes it much easier for you, and others, to work with your code—you're not writing these comments for you now, but for you in six weeks/months/years. I speak from experience: code written today is useless in a year because you won't remember what it does. Even more exciting, if you follow the instructions below we will be able to turn your comments into help files automatically in the last session of this section! Below is an example of the comments I made for the wrapper function I described above, in what is called `roxygen` format:

```
#' Make an elevational grid with optional lakes and rivers
#'
#' A light wrapper around \code{diamond.square} and \code{add.rivers}
#' @param n Size of grid will be a 2^n +1 grid (default: 6; a 64 x 64
#'     grid)
#' @param lakes logical whether to make all terrain lower than 0
#'     height underwater (default: TRUE)
#' @param rivers number of rivers to add to terrain (default: 5)
#' @param noise range for random noise to be added at each step of
#'     diamond.square algorithm; vector of length two, first
#'     element is the first standard deviation (used to draw the seeds
#'     for the corners of the grid) and the second is the last
#'     standard deviation for noise at the finest spatial
#'     scale. Standard deviations are evenly spread out across all
#'     depths of the diamond.square algorithm.
```

```
#' @return a terrain matrix; numeric elements indicate heights, and
#'     NAs indicate cells filled with water
#' @examples
#' huge.terrain <- make.terrain(8, rivers=20)
#' image(huge.terrain)
make.terrain <- function(n=6, lakes=TRUE, rivers=5, noise=c(5,.5)){
    #...all my code...
}
```

Importantly, each line start with the comment character '#' and then '' ' (note that space!). These let `roxygen` (which we will cover in the future) know that these aren't just comments, they're to be turned into documentation. The first line is a brief description of the function, which will be the help file's title page, and then (after an empty line) I write out a slightly more detailed description that could be many sentences. I then describe the `@parameters` (which, unhelpfully, is the keyword to describe function *arguments*). Give the name of the argument, and then its description. I give the `@return` value, an example (or two), and we're done! I then, of course, define the function itself.

5.3 Bonus exercises: chasing waterfalls and plotting

If you've made it this far, you have probably had trouble tearing yourself away from looking at the beautiful terrains you've generated. There are, however, two additional exercises you could try that would make your terrain even more exciting and easy to look at. These are optional, but they are (of course) fun!

5.3.1 Rivers

You might have noticed, in my documentation example, that I gave the option to make rivers. Why not write an `add.river` function that will add a river to your terrain, starting at a random spot and moving downhill until it can't go any further without going back on itself, going uphill, or going off the edge of the terrain. The river should probably stop if it finds itself in a tile adjacent to a lake as well. This exercise isn't too tricky if you break everything down into sub-problems (*e.g.*, write a function to check if the river should stop moving, write a function to suggest the best location the river could travel to, etc.). Your terrain will look a lot prettier with rivers running through it!

5.3.2 Plotting

Looking at your terrain using `image` is fine for debugging, but it's not a very attractive or user-friendly method. Why not write a wrapper function that takes a terrain matrix and plots it for the user? See if you can use sensible colours for the terrain heights, and maybe plot rivers and lakes in blue (the `add` argument may be helpful here). If you want, you could even create a new *class* for your terrain, and then write `print`, `plot`, and `summary` methods for it.

Session 6

Plants

Overview

This is the second of four sessions in the 'R-World' series. In this session, you're going to simulate two (or, if you wish, more) species of plant that are going to live on the terrain that you built in the last session. Slowly building up the complexity of these plants, we're going to start by making them survive (and, of course, die), then reproduce, and then compete with each other. While very simple, the rules governing how these plants will interact with each other are based on published meta-population literature, which you can read about in chapter four in the third edition of 'Theoretical Ecology: Principles and Foundations' edited by Robert May and Angela McLean. It will be up to you to find the parameters in your model that will enable these species to co-exist! This is real biology: if you can think of extensions to this work, then you've just thought of a paper that you could publish!

To bring living things into our simulation, we're going to have to incorporate time, which, for our purposes, will advanced in discrete *timesteps*. Every plant will be of a certain *species*, and each species will have a probability of *surviving* in each time-step of our simulation and a probability of *reproducing*. When an individual reproduces its offspring will appear in a nearby cell of our grid; this means that eventually individuals from two species will be brought into *competition* with each other and so we will have to have probabilities determining which species are likely to survive. This may sound very intimidating, but in reality it's going to be quite simple because we're going to break down our problem, tackle each of these aspects one-at-at-a-time, and build up the complexity of our model part-by-part.

6.1 Life history parameters

Defensive programming is all about writing code that actively checks to make sure the user can't make mistakes by giving us the wrong kind of data, and actively checks results of intermediate steps to make sure nothing's gone wrong. We're going to start by defensively programming a function that creates an object that describes the species in our simulation, so we know we always can rely on the right kind of information going forward.

You're going to write a function, called `setup.plants`, that takes as input `reproduction`, `survival`, and `competition` parameters for our plant species. The program will check that both the `reproduction` and `survival` vectors are the same length (the number of species in our simulation), and that the `competition` argument is a matrix with dimensions equal to the length of the other variables (this matrix determines the likelihood that each species will will when in competition with another). You're going to use the `stop` function to 'raise' an error if those conditions aren't met—I imagine you know what an error message in `R` looks like by now! So you'll write something like this:

```
setup.plants <- function(repro, survive, comp.mat, names=NULL){
    if(is.null(names))
        names <- letters[seq_along(repro)]
    if(length(repro) != length(survive))
        stop("Reproduction and survival parameters needed for all species")
    #...some more tests...
    repro <- setNames(repro, names)
    #...what does the line above do? Do you want more like it?
    return(list(repro=repro, survive=survive, comp.mat=comp.mat,
                                                    names=names))
}
```

6.2 First steps: storing plants and keeping them alive

Now we have a function to set up an object that describes each of our plant species, we're going to write a function that determines whether a particular species will survive (`survive`), another that runs one timestep of our simulation across the whole simulated ecosystem (`plant.timestep`), and finally a wrapper function that takes a terrain, a load of parameters for plants, and simulates our world (`run.plant.ecosystem`). Our plants are going to be stored in a `matrix`, just like our terrain. Only this matrix will contain `character` data: the `name` of the species in a particular cell, a blank (`' '`) character if nothing's there, and an `NA` if it's a water cell.

6.2.1 `survive`

This function is simple: given the content of a grid cell, and information about all the plant species (the output of `setup.plants`), it will randomly draw whether that individual survives according to our simulation parameters. So it will look something like this:

```
  survive <- function(cell, info){
    #...some code to check whether cell is empty or has water...
    if(runif(1) <= info$survive[plant])
      #$The plant survived! so do something...
    }
```

...take a look at what `runif(1)` does. **Do you understand why comparing that with a probability helps us draw something with that probability?** Think it through, ask your neighbour (if your neighbour likes stats), and if you're stuck ask me. It's a useful trick to understand in statistical programming, and you will use it a lot in the next few sessions.

6.2.2 `plant.timestep`

Now we need to make time tick by for an entire matrix of plants, and you can write this function on your own! Write a function that takes, as an input, a matrix of plants (the character matrix I described above), then loop over that matrix and apply the `survive` function to all its contents. I know, I know, loops are slow, but make sure you use one (...or maybe two...) because they're more flexible and we'll need that flexibility later. If you want, you can define your `survive` function inside `plant.timestep`, as it will keep things tidier and easier to keep track of later. Like this:

```
plant.timestep <- function(plants, terrain, info){
  survive <- function(plant, info){
    #...survive function...
  }
  #...looping et al...
  return(new.plants.matrix)
}
```

6.2.3 `run.plant.ecosystem`

Most of the work for this final function is done—we've already written the time-step bits—but setting everything up can be tricky. When I wrote my function, I assumed the user already had a terrain they wanted to simulate everything on, but I still had to 'seed' it with starting plants, and get information from the user on what kinds of plant parameters they want to use.

You'll want to store your information about your plants in an `array`, which is just like a `matrix` but has three dimensions. You can use the third dimension to keep track of how your plants change through time, so, for example, the first load of plants that you seed in will go in `plants[,,1]`, the second in `plants[,,2]`, etc. My advice would be to randomly add however many individuals the user wants into the matrix at random, and then to go in *afterwards* and make `NA` any plants that happened to land on water. To do that, and make sure you keep track of where the water is in your matrix, take a look at the code below. Of course, you could just pass the `terrain` matrix to every function that works with plants and use that to check where the water is, but that will get a bit cumbersome and will slow our program down—better to take the hit now and just get it out the way.

```
plants <- array("", dim=c(dim(terrain),timesteps+1))
#...why timesteps+1, do you think?...
for(i in seq_len(dim(plants)[3]))
  plants[,,i][is.na(terrain)] <- NA
```

6.3 Reproduction

You've probably noticed that, through time, your plants are dying out. That's because we haven't programmed them to reproduce yet. Reproduction is pretty simple (...to simulate...)—we're just going to write a function called `reproduce` and add a call to it in our `plant.timestep` function:

```
plant <- reproduce(row, column, plants, info)
```

...notice that we're passing the entire `plant` matrix here, because if/when a plant reproduces its progeny will appear in a new cell surrounding its parent. I'll give you a hint on how to figure out where a new species could go; your `reproduce` function will look something like this inside:

```
reproduce <- function(row, col, plants, info){
  possible.locations <- as.matrix(expand.grid(row+c(-1,0,1), col+c(-1,0,1)))
  #...now filter out which ones are not water-logged and reproduce there...
  #...being careful to check you do have somewhere to reproduce to!...
  return(plants)
}
```

6.4 Competition

Because you're a very intelligent Utah State University biological programmer, you probably noticed something quite concerning about the reproduction step: sometimes there's already another plant in a grid cell! Such a case represents *competition*: only one individual can survive, and so which should it be? This is what the `comp.mat` part of our `setup.plants` function is all about. There are as many rows and columns as there are species, and so each `comp.mat[row,column]` gives the probability that a particular species will win and live in a grid cell. So to determine which species should be in a cell, we just need to run something like:

```
sample(species_names, 1, prob=comp.mat[row,column])
```

...to determine who wins in the fight. With this in mind, you should now be able to write a `fight` function and add it in the appropriate place within your `reproduction` function (why should it go there?).

6.5 Biology

Congratulations! You now have a working simulation of a plant meta-community! Of course, that doesn't mean that these plants co-exist. So, your challenge for the end of the session is to use your knowledge of biology to figure out what combinations of parameter values lead to stably co-existing species with roughly equal abundances in the ecosystem. As a hint, below I've given some starting parameters that give two species that don't co-exist. Have

```
unbalanced <- run.plant.ecosystem(terrain, 100, survive=c(.95,.95),
                    repro=c(.4, .6), comp.mat=matrix(c(.7,.3,.3,.7),2))
```

6.6 Bonus exercises: plotting

See if you can extend your functions, and classes, from the previous session to handle plants. You'll have to think carefully about colour schemes, and also about how to represent multiple species of plant in a simulation. You may also want to think about providing functions that plot species' abundances over time, making use of `apply` on the third dimensions of your `plants` matrix. Maybe you can even figure out how to use the `animation` package to make videos of how your ecosystems have been changing over time. You can find an example of that package in action on my GitHub page (https://github.com/willpearse/willeerd/blob/master/R/fiber.plot.R#L60).

Session 7

Herbivores

Overview

This is the third of four sessions in the 'R-World' series. In this session, you're going to add moving animals to your ecosystem and, sadly, they're only going to survive by eating your creations from the previous session. While the rules governing their behaviour are going to be very simple, you'll find that the effect they have on the populations dynamics of the plants will be quite dramatic. By playing with the demographic parameters of your herbivores, you might be able to find population cycles, and possibly even force your herbivores to extinction!

Our herbivores are going to have three life-history parameters that will determine their probabilities of *eating* a plant, of *killing* a plant when they eat, and of *reproducing* once they eat. To complicate matters somewhat, but also to make the herbivores a little more realistic, they're not always going to feed every time-step, and when they don't feed they're going to move. Whether a herbivore feeds or not will be related to how long since they last fed, and if they go too long without food they will die! While the dynamics of herbivore death might sound complicated, in reality it's not going to be very difficult as long as we give some thought to what *data structure* we use to store our herbivores.

7.1 Life history parameters and data structures

Like their plant food, our herbivores are going to be stored in `matrix` format, but unlike them they're going to be in `numeric` matrices. `NA`s will still represent water, but herbivores will be represented by the number of timesteps until they die; cells with `0` will contain no (living) herbivores, and once a herbivore has eaten its cell value will be set to the maximum number of timesteps defined by the herbivore's life history parameter. *There will be only one species of herbivore*, although making another is an extended exercise for the interested! In each timestep, a herbivore can either move or eat, and if it eats it may reproduce; life-history parameters govern all of these events (`eat` and `repro`, respectively), just as they did for our plants.

Importantly, a herbivore's likelihood of eating (and not moving) if it's in a cell with a plant is determined by how long it can go until feeding. This sounds complicated, but is simple to implement: `eat` is going to be a

`vector`, and you will simply subset `eat` to pull out the relevant probability using the value for each individual in the herbivore matrix (something like `eat[herbivore[row,col]]`). Once a herbivore eats, its value in the herbivore matrix gets reset to its maximum; you'll probably want to keep track of what that is with a parameter called `sated` (what would happen if this value were greater than the length of `eat`?). If a herbivore eats, it will either lightly graze on the plant (doing no real damage) or kill that plants, with a probability determined by its `kill` life-history parameter.

You now know enough to write your own `setup.herbivores` function, exactly like you did for the plants last time.

7.2 Speed isn't everything

In many ways, the herbivores are much easier to simulate than the plants, because there's only one species and so no competition. However, if you think about the problem, you'll realise that herbivores are going to do a lot more moving around than plants: they both move *and* reproduce, and each of those steps will require you to make a list of the locations a herbivore could move to. My advice, therefore, is to start by writing a `new.loc` function that will take a given herbivore matrix and come up with a set of possible new locations. You should make sure that herbivores don't walk on water (this is a strictly secular simulation), that they don't walk off the edge of the grid, and that they don't walk into a cell where there's another herbivore.

Once you've got that `new.loc` function working, we need to think about how to write the `herbivore.timestep` function. What makes herbivores a bit fiddly is that so much depends on whether they eat: *if* there's food, and *if* they eat, their value gets reset to their maximum and then *if* they reproduce you need to make a new herbivore somewhere. You could take the approach we used in our plant model last time, and write separate `survive`, `reproduce`, and `eat` functions for all of these aspects. Personally, I found my code was more concise if I just did everything inside one internal `herbivore` function, which I lightly wrapped wrapped with my `herbivore.timestep` function.

There is no correct way of writing this—everyone has their own unique programming style, and (whether you know it or not) you've been developing yours throughout this course. You could even try implementing this aspect of the exercise in both of the ways I describe above, to see which you prefer (you do not have to do this if you don't want to). Remember that the *clarity* of code (how understandable is your code without comments?) is often as important as how *concise* (fewer lines are easier to read and understand) and *efficient* (little time and memory spent on execution) your code is. You will rarely find that you can maximise all three of these things, and I can't tell you which is best. However, I can tell you that computing power and memory are constantly increasing: your cell phone has more processors, which are faster, and has vastly more memory, than the computer I first learnt to program on. Also, it takes time to write efficient code; if it takes two days to make something two hours faster, is it worth it if you will only run the code once? So, personally, I value the first two more than the third, as long as something is fast enough to work with. It's your choice, however!

7.3 Abstraction

Once you're written `herbivore.timestep`, it's reasonably trivial to write a `run.ecosystem` function that simulates both plants and herbivores. However, you'll notice quite quickly that you have to seed both the plants and herbivores across your terrain. Seeding plants and populating herbivores might seem like two very different processes, but if we mentally take a step back—*abstract*—we might be able to see some commonalities between them. Both involve taking a matrix, both involve putting some sort of value randomly within that matrix (`characters` for plants, `numerics` for herbivores), and both involve setting things to `NA` where they are `NA` in another matrix. So I wrote an internal `seed` function that created an `array`, seeded `matrix[,,1]` with elements, and filled in `NA`s throughout the matrix. Abstracting out a problem like this into a function made my code shorter and easier to read, and also will make it easier to use this code in the future in other problems. Most programmers keep a few files lying around with abstracted functions that solve problems they frequently find themselves coming up against, and when we learn about algorithms you'll find it even easier to make such a file yourself. If you want to see mine, I turned it into a package (`library(devtools);install_github("willpearse/willeerd")`), which is a great way of building your CV and can, eventually, turn into a package on CRAN!

7.4 Biology

Now you have herbivores in your system, play around with different values for the `eat` parameter—what happens when you make that longer or fill it with different values? What happens to the plants when you have herbivores in the system? The presence of higher trophic levels in a meta-community can often dampen the population dynamics of plants—do you see that here? Take a moment and congratulate yourself on what you've achieved over the last few sessions: you've created a quite sophisticated meta-population model that replicates dynamics that theoretical biologists are studying the consequences of today. If you can think of an extension to write for this model, then you could study its dynamics and write a paper on it!

7.5 Bonus exercises

Programmatically, extending the classes you've developed over the last few sessions' bonus exercises (or starting now if you haven't already!) is a very valuable thing to spend your time on. Indeed, it could become very useful for the work you'll be doing in the next session, and if/when we learn about `shiny` apps later you'll be able to use that work then.

Biologically, any extension to this system would be interesting, and I strongly encourage you to follow your own interests. That said, here are some things that could be interesting to explore:

- Predators. Predators could be quite simple to program—their internal logic is essentially the same as herbivores, only they feed from the herbivore matrix. What effect do they have on the population dynamics of this system?

- Build a mountain range with a very narrow pass down the middle of it (if you can do this programmatically, even better!). What effect does this have on the population dynamics of your system? What happens if different plants and herbivores are seeded on different sides of this pass?
- Herbivores sometimes prefer certain plant species. What happens if you program different `eat` probabilities for the different plant species? What happens if you program *two* kinds of herbivore, each with different preferences?
- Animals rarely move one cell at a time throughout a landscape, and movement ecologists often study what is called a Lévy walk where sometimes animals move very far, and most of the time go very short distance. Can you program herbivores to move in that way? You may already have quite a lot of code to help you simulate other kinds of movement from a previous exercise you carried out for Professor Savitzky...

Session 8

Packaging

Overview

This is the last of the four sessions in the 'R-World' series. In this session, you're going to put everything you've done so far in this series together and create a package! Learning how to make a package is an important step in understanding how R works as an both a programming language and ecosystem of different packages. This session also serves as an opportunity to tidy up your code from previous sessions, and attempt some of the bonus exercises if you haven't already.

WARNING: Do not upload your R package to CRAN; only upload your package to GitHub. Any students attempting to upload their package to CRAN will have wasted the valuable time of the CRAN maintainers and so will fail this course. Don't worry, it's essentially impossible to accidentally upload your package to CRAN; to do so you must go to a special website, click several buttons, and fill out boring forms.

8.1 What makes up a package

In the lecture I describe two ways of making a package; using RStudio's built-in tool (just make a new package using wizard in the file menu), and using the `package.skeleton` function in R. Either of those routes will give you the basic outlines of a package. Your main task for today is to turn your code from the last three sessions into an R package. Every package must contain the following components:

- `DESCRIPTION` file. This is what you see on a package's website on CRAN, and at the top of the output from `library(help=package_name)`. The entries are fairly obvious (and all are essentially mandatory); a common gotcha is not indenting each new line that doesn't indicate a new field with some spaces (four is common). This isn't a made-up convention; it's the standard for Linux configuration files (if you're a nerd this makes your life easier). Packages often need other packages, and there are three ways of specifying this: `suggests` means 'these packages complement this package well; they might be needed to build the vignette or examples in the help files (see below)', `imports` means 'these packages are necessary to run this package's code', and `depends` means 'same as `imports`, but this is so important for

you that when you type `library(package_name)` I'll run `library(depended_package)` as you'll be using that package all the time'. Specifying version numbers (*e.g.*, `ape (>= 3.1-4)`) is a tremendously good idea, as otherwise someone might try and use an older version of a package that doesn't have features you need.

- `NAMESPACE` file. This describes what functions in your package are *exported* (intended for user use) and which functions from other packages you used and *imported*. This file can be auto-generated by `roxygen2` (see below). The list of functions that are available within a package is often called a *namespace* (it's essentially a fancy kind of *scope*), hence the name of the file.
- `R` folder. This is where all your `R` code goes, in `.R` files. You can use any grouping of code within files you want.
- `man` folder. This is where all the `.Rd` documentation files (the `man`ual of your package). These files can be auto-generated by `roxygen2` (see below).

There are also a few optional components, or (in the case of directories), components that can be empty:

- `NEWS` file. List, in whatever format you want, changes in your package. Tremendously useful, and users love it when you name-check them for helping out (*e.g.*, https://cran.r-project.org/web/packages/pez/NEWS).
- `LICENSE` file. Programmers get very touchy about what copyright (or copyleft) conditions a package is distributed under. If you're using something non-standard, or have to give additional information, your license file can contain this. Otherwise, if you put one of the standard ones (GNU is common) in your `DESCRIPTION` file, this will be auto-generated for you. This is a very dull, but very important, aspect of releasing a package or software—everything should have a license, even if that license says "mine, leave alone!". Read http://choosealicense.com/ for more information.
- `data` folder. If your package contains data, which a user can load in with something like `data(cars)`, then its `.RData` file goes here.
- `vignettes` folder. Vignettes are *incredibly useful*, and if you're serious about releasing a package (or using one) you really should write (or read) one. The raw material for building a package goes here (LaTeX, `Markdown`, or other `knitr`-able files) go here, or simply the outputted PDF. We will not be covering how to build such a file here, but there are plenty of tutorials online. (I realise that LaTeX looks a lot more complicated than other options (it is), but bear in mind that most of its 'complexity' is features that you will need. If you're about to write a PhD thesis, learning it is an excellent idea—you'll thank me when your friends are trying to edit a ~200 page Word document and it's trivial for you).
- `inst` folder. Sort of a storing-house for junk (and where vignettes used to live); now it's a good place to have a folder called `tests` for your unit tests
- `src` folder. If you're writing a package that uses `C`/`C++` (we might be later) or `Fortran` code, that code will live here.

8.2 Putting it all together

Much of your package can be auto-generated by `roxygen2`. By loading that package into `R` and then running `roxygenize("~/my_package_location")`, it will auto-generate all your documentation etc. for you if you have been following my commenting advice in the first session of this section. There are some extra formatting things that are useful to know in `roxygen2`, so have a read of an example of one of my files (https://github.com/willpearse/pez/blob/master/R/comparative.comm.R) and make sure you understand (and can use!) all of the `@`-tags and `\commands` in it. Using those and `roxygen2`, fill out the documentation for your package.

Now comes the trickiest part: checking your package. If you're in `RStudio` control-shift-E will do that for you; otherwise open up Terminal/command prompt and run `R CMD check directory_of_package`. This generates a lot of output (stay calm); focus on 'errors' and then move onto 'warnings'. Your main task today is to methodically work through all of these until your package builds and checks cleanly—the errors often make sense if you read them, or of course there's Google (and me). It's generally a better idea to `build` your package before you check it, but don't worry about that for now. Other useful flags (options) are `--as-cran` (run extra checks CRAN will run) and `-no-build-vignettes` (saves time). `RStudio` is great, but writing your own build-test-check script is quite simple, and will save you a bit of time in the long-run. As an example, here is my own checking script for `pez`[1] (which will work on any Mac or Linux machine, and parts of it on Windows); can you figure out what it does?

```
Rscript -e "require(roxygen2);roxygenize('~/Code/pez/')"
R CMD build pez --no-build-vignettes
Rscript -e "install.packages('pez_1.1-0.tar.gz')"
Rscript -e "require(pez);require(testthat);test_dir('pez/inst/tests')"
```

8.3 Sharing your code

Eventually, all packages intended for wider use should make their way onto CRAN (or Bioconductor; https://www.bioconductor.org/)—we won't be doing that today. However, it's a great idea to use GitHub to share your package once it's in a state where others could use it, because you can get feedback and help the community. The 'release cycle' of GitHub is also much faster; you can push bug-fixes immediately, while CRAN requires checks and so can only be updated every month or so.

To put a package up online at GitHub, make a new Git repository (this is done for you in `RStudio` and make sure you `add` all the files (including documentation!). Then `push` your repo up. That's it! Your users can now install you package from `R` by running `devtools::install_github("username/package_name")`! Easy!

[1] (If you're a real BASH nerd, re-write this script so it will run the tests on a remote server, copying everything over using scp et al.)

8.4 'Bonus' exercises: classes

If you finish early, and your package is up online and all is working well, go back through the other sections in this section and carry out those bonus exercises. I find it impossible to believe that you can have tried every possible biological extension of this project; if you're stuck for ideas please ask me because there are lots of things (like incorporating (co-)evolution!) that make for fun programming exercises depending on your interests. If you're desperate for programming extensions to try, make sure you have unit tests (take a look at my script above for inspiration) or try your hand at using `knitr` (http://kbroman.org/knitr_knutshell/) to make vignettes and manuscripts containing `R` code. Ask me for more details if you're stuck!

Appendix A

How to install the software you need for this course

R

Windows, Mac, and Linux users can go to https://www.rstudio.com/products/rstudio/download3/ and download and then run whatever installer is appropriate for their computer. This software is free.

`RStudio` is a wrapper around `R` that most beginners find more pleasant to use. While I advise all users who are getting started with `R` to use `RStudio`, there may come a point where you tire of its user interface, or would rather use something that integrates more cleanly with whatever text editor or development environment you prefer. If so, you can download `R` without the `RStudio` from the offical `R` website (https://cran.r-project.org/).

The following applies only to people who alrady have a text editor preference; this will seem like nonsense otherwise and so please ignore. Emacs users (like me) should take a look at ESS, and vim and SublimeText users will find excellent development environments for them as well. Atom users are fine if they're on Mac and (I think) Windows, but otherwise you're out of luck.

Python

Windows, Mac, and Linux users can go to https://www.python.org/downloads/ and *must download Python version 3.something, not version 2.something*. This will install a Python editor that is reasonable to use for you. Do be aware that MacOS X and most flavours of Linux come pre-installed with some form of Python; you will still want to install a verison from here because that may not be the second major version of `Python` (see below).

In 2008, `Python` released a new major version (3) that included a number of features that were not backwards-compatible with `Python` version 2. This caused a lot of hassle, and to this day the two versions are essentially co-existing around the world. I am teaching you version 3, since version 2 will eventually stop being supported, but it can make things a bit confusing as it is often difficult to tell what version of `Python` has been installed

already on a computer. Operating systems like Mac and Ubuntu that need a particular verison of `Python` will not be affected by installing version 3 from the above website; the two versions can happily co-exist and the operating systems are set up to handle that.

The following applies only to people who alrady have a text editor preference; this will seem like nonsense otherwise and so please ignore. Emacs users (like me) should take a look at ESS, and vim and SublimeText users will find excellent development environments for them as well. Atom users are fine if they're on Mac and (I think) Windows, but otherwise you're out of luck.

The following applies only to people who alrady have a text editor preference; this will seem like nonsense otherwise and so please ignore. `Python` support in essentially all text editors is very mature; `Python` is rapidly becoming the lingua franca of programming and so not having decent support out of the box is enough for most people to switch. Emacs users (like me) will likely find that `python-mode` is sufficient, I can't remember ever having configured anything for it while I still actively developed in the language and my `init` file currently only contains `(add-hook 'python-mode-hook (lambda () (electric-indent-mode t)))`.

Git

You have two choices when using `Git` and `GitHub`: using a graphical front-end or the command-line. You may not be surprised to hear that I prefer the command-line; the commands you will be using every day are not too complex. Windows and Mac users can go to https://desktop.github.com/ and use the front-end that `GitHub` have developed; Linux users are probably going to prefer using the terminal anyway but graphical programs do exist for them too elsewhere. You need a `GitHub` account to use their desktop program.

Otherwise, go to https://git-scm.com/downloads and then follow the instructions. Windows users will find a Git console is installed, which means they have to enter command-line instructions to use Git. This is to be expected. Mac users will find they have a program they can open via the terminal. Linux users should open their terminal and type `sudo apt install git` (or use `yum`, or whatever else you use).

GitHub

Go to https://github.com/ and register for an account (it's free). Once you have done this, go to this address https://classroom.github.com/classrooms/21243925-programming-for-biologists-2016 to register for the class on GitHub. This is the course's home on `GitHub`, and will make it possible for you to share your homework with me, get help with code, etc. You will also want to register for an academic account on GitHub, which will allow you to have your own (unlimited) private repositories, here https://education.github.com/pack. You'll be interested in very little of the free stuff that they give you with this pack, with the potential exception of `atom` (which isn't very good for `R`, but I hear is fine for `Python`).

Appendix B

Using and GitHub

It is very important to recognise than git is *not* GitHub. Many people on the Internet do not understand this, and it's not really in the interests of GitHub to correct them.

git is *version control* software that keeps track of changes of whatever files within a *repository* (folder) you tell it to. Whenever you want, you can *commit* those changes to keep a record of everything you've done up to that point. You can *commit* as often as you like, and at any point you want you can restore all of your files to the way they looked at any previous commit. It's a bit like saving your progress in a computer game: you can save before going ahead to fight the big monster (write a new piece of code), and if you make a mistake you can just load (go back to a previous commit) and start again.

By keeping track of changes across all your files at the same time, git makes it easier to track back through previous changes and find where you first introduced bugs and then put back whatever you messed up. It also makes it trivial to incorporate other people's changes: git keeps track of the changes they made, and then flags them for you and remembers who did what.

GitHub is a website that store git repositories, and makes it easy to share code with others. Using git without a central repository like the kind GitHub (and GitLab, and BitBucket, and...) offers is fine, but for collaborative projects it's much easier with something like GitHub. When you come back to a project, you'll *pull* down all the changes that have been made on the central server, and once you've *committed* some changes you'll *push* them up. By breaking things down into chunks like this, you'll make it easier for others to work with your code code without breaking it—all your changes are kept together.

Within git you can also *branch* off the main body of someone's code, implement a new feature, and then *merge* it back in afterwards. This is trivial within git, and GitHub makes it even easier using what are called *pull requests*. This is a way of saying "hey person I'm collaborating with, I made a feature—would you check what I've done, and then pull/merge it into your code?". In fact, taking a look at this pull request (https://github.com/ropensci/fulltext/pull/62) might show you why I love GitHub so much. In my first post, you can see I mention I have resolved a GitHub *issue* (number 61), you can see where we discuss the changes and reference specific lines in the code for refactoring, and you can also see (if you look very closely) some red crosses where GitHub warned us that our unit tests (which it automatically ran) hadn't passed. Indeed, while

the process was going on, GitHub gave us information about whether my changes would conflict with any other work going on, all automatically!

B.1 Enough git to get started

In what follows, I assume you've opened a terminal window of some sort. In Mac and Linux, that means opening the program called Terminal, and in Windows that means opening the GitShell. It is also possible to do all of this using GitHub's Desktop program: I don't give instructions for this because I can't use that program. It looks to be very easy to use regardless.

To start a git repository, type `git init`. This creates a hidden Git folder (called `.git`) that keeps track of all your changes to files. If you're loading code from the course website, you will of course not have to do this since you'll be loading a git repository from that website. Go ahead and start working on your project: make files and folder as you would normally.

Eventually, you'll come to a point where you'll want to save some of your work. Perhaps you've implemented the first feature of your program, or maybe you're just about to go out for coffee. Either way, now is a great time to make your first *commit*: to take a snapshot of your work as it stands right now. Type `git status` to see the status of you git repository—right now, it's tracking nothing. Type `git add name_of_file` to tell git to keep track of files within your repository. If you need to, you can ask to it track all the files in a folder by typing `git add folder_name/*`. Note that you can't just track an empty folder—git ignores any attempt to ask it to track such a folder.

Once you've added all your files, it's time to make that *commit*. Type `git commit -am "A brief description of what you've done"`. We've given git two flags: `a` means "*record all my changes*" and `m` means "*I'm about to give you a brief commit message that describes the work I've done*". That's it. You've just used git. It was that simple.

As soon as possible, you should back up your data to GitHub: you need to *push* your code up onto the site so that if you lose your laptop it's backed up, and so that others can see it. Go to the GitHub website, click the plus button, then select "New repository". Give your repository a name and description, and then click "create repository". You'll then be taken to a sort-of holding page that GitHub uses for empty repositories with a quick guide on how to get started. Go back to your Terminal, then type something like:

```
git remote add origin https://github.com/user_name/repostiory_name.git
git push -u origin master
```

...you can just copy-paste this code from the holding page. You'll be prompted for your username, and then... that's it! You're done! The next time you want to push your repository to GitHub, you can just type `git push`. Simple!

B.2 Enough `git` to be dangerous

`git` and `GitHub` have loads of features, and there's absolutely no point in me describing all of them, in part because there are so many fantastic resources to look up about the two (*e.g.*, https://git-scm.com/book/en/v2). The best way to learn is to poke around, but here are a few things to get you started:

- `git pull` Eventually, you will start sharing a repository with someone else, and they will make changes that you want to merge with your work. Typing `git pull` will *pull* down whatever changes have been made on the remote repository you're sharing (almost certainly `GitHub`). If there are *conflicts* between what you've done and what they've done (you've made changes that are incompatible) these will be flagged, and `git` will guide you through the process of *resolving* them. Don't worry about this until it happens... Look it up on the book link above when it does.
- `git branch` Sometimes you want to experiment, and you want to be able to give up on your experiment once you realise what you've done wrong. `git branch name_of_experiment` makes a new *branch*, which is a clone of your repository at that moment that you can switch back and forth between.
- `git checkout` You can use this to switch between branches (*e.g.*, `git checkout experimental`) or to check out a previous commit using its hash (*e.g.*, `git checkout 123a654`). Hashes are listened on the `GitHub` website, and you can also find them using `git log`.