Contents

1 A brief introduction to phylogenetics in R

1.1 Introduction

This book is about carrying out phylogenetic comparative analyses in the R statistical computing environment.

In this chapter, we will:

1. Introduce the general field of evolutionary research called *phylogenetic comparative biology* and discuss how the R scientific computing environment can be used in the analysis of phylogenetic data.
2. Present and explain the general structure of this volume, including how we expect it to be read and used.
3. Introduce the major R function libraries (called "packages") used to analyze phylogenetic data in the R environment.
4. Examine the `"phylo"` object: an important data structure that is used by most phylogenetic R packages to encode a tree.
5. Finally, illustrate a number of important R functions for phylogenetic analysis, including how to read and write trees and phylogenetic data, how to plot trees in various styles, how to manage phylogenetic trees and data, and how to conduct a simple phylogenetic comparative analysis in R.

1.1.1 What is phylogenetic comparative analysis?

Phylogenetic comparative analysis[1] is the general endeavor of using a phylogenetic tree, frequently combined with phenotypic trait data for the species in the tree, to learn something about evolution (Harvey and Pagel 1991; Pennell and Harmon 2013).

Although the modern field of phylogenetic comparative analysis is relatively young (tracing back largely to Felsenstein 1985), phylogenetic comparative methods have diversified in scope, number, and importance in recent decades (reviewed in Harmon 2019).

[1] Often called *phylogenetic comparative methods*, *PCMs*, or sometimes just *the comparative method*.

Contemporary phylogenetic comparative methods now encompass an enormous range of topics. For instance, phylogenetic comparative analyses have been employed to measure the relationships between characters while taking the phylogeny into account (Martins and Hansen 1997), to infer the rates of species proliferation and extinction through time (Nee 2006), and to fit sophisticated mathematical models to phylogenies and comparative data in an attempt to explain the diversity of life that we see around us on this planet (Maddison et al. 2007). Comparative methods have also been used to track the spread of diseases (Stadler and Bonhoeffer 2013), to understand contemporary threats to species (Greenberg and Mooers 2017), and to describe the dynamics of evolution over thousands or millions of years (Uyeda et al. 2016). Phylogenetics comparative methods have even been used to study the global SARS-CoV-2 pandemic that started in 2019 (e.g., Wang et al. 2020; Sjaarda et al. 2021).

1.1.2 Phylogenetic comparative analysis in R

Over the past decade, the scientific computing environment R (R Development Core Team 2020) has grown to play a key role in phylogenetic comparative methods. Many developers of PCMs tend to work in R, and many PCM users prefer to conduct their analyses in R. This synergy between users and developers means that R has become an essential tool for scientists interested in employing the comparative method in their research.

The purpose of this book to teach users how to carry out phylogenetic comparative methods using R. We only briefly cover R basics, so readers completely new to the R environment might think about complementing this volume with a simpler book focused on introducing the R computing environment.

This book is designed to complement, not replace, a more complete theoretical treatment of phylogenetic comparative methods. As such, we do not fully explain the mathematics and conceptual basis of all comparative methods covered herein. For a more comprehensive review of the theoretical basis of phylogenetic comparative analysis, we recommend Harmon (2019), Garamszegi (2014), Nunn (2011), or one of several excellent books that cover parts of phylogenetic comparative biology as part of a larger treatment of phylogeny inference (e.g., Felsenstein 2004; Yang 2006). We expect that this book will be used either in parallel with a full course of study on phylogenies and the comparative method (using, for instance, Harmon 2019) or by scientists already familiar with much of the theoretical basis of phylogenetic inference, phylogenetic comparative methods, or both—and ready to immerse themselves in the phylogenetic comparative biology in the R computing environment. Our goal in this book is thus highly practical: to give scientists the tools they need to start analyzing their own comparative data.

The book is not designed to cover all phylogenetic comparative methods. First, we focus exclusively on phylogenetic comparative methods implemented in the R computing environment. Several important phylogenetic analyses (e.g., Pagel and Meade 2013; Rabosky 2014) are implemented in software that run outside of R. As such, we largely consider these methods to be out of scope for the book.[2] Second, we focus especially on methods implemented in our own packages *phytools* (Revell 2012) and *geiger* (Harmon et al. 2008; Pennell et al. 2014), as well as in the core phylogenetics R package *ape* (Paradis et al. 2004). In part, this is due to our own intrinsic biases; however, it's also motivated in equal measure by a desire to ensure that this book remains useful over the medium to long term. As package authors and maintainers, it's much easier for us to guarantee that updates and extensions of the *phytools* and *geiger* R libraries will always remain compatible with the code presented in this book. As such, when a method is implemented in both *geiger* or *phytools* and another R package, we will generally prefer to use

[2]Although we promise to discuss the relationship of some of these important methodologies to those that we are covering.

our packages—unless functionality is vastly different between the different implementations. On the other hand, in this book we do cover many important methods implemented in R packages other than our own, and in those cases (obviously), we show how to use these function libraries.

One quick note on controversies in the field. Phylogenetic comparative methods grew organically, with new methods being added rapidly—and sometimes with very little testing or evaluation. Occasionally, methods are shown to have undesirable properties. In other cases, statistical approaches that are commonly used have nuances that can only be appreciated with extensive simulation studies. We subscribe to the philosophy that comparative methods are "normal statistical methods." Consequently, we tend to describe these critiques in terms of standard statistical concepts such as statistical error, model adequacy, identifiability, and so on.

This book is largely based on the content that we developed for a series of classes and workshops that we've taught over the past half-dozen years or so across at least eleven countries and four continents. These workshops were not developed or taught in a vacuum and owe their existence to a long list of collaborators, including (but not restricted to) M. Alfaro, R. Betancur, A. Crawford, S. De Esteban-Trivigno, A. Gonzalez-Voyer, R. Zenil-Feruguson, and J. Tavera, among others.

1.1.3 How to use this book

As noted in the previous section, this book is designed to *complement*, not replace, more comprehensive theoretical treatments of phylogenetic comparative methods, such as Harmon (2019).

We anticipate that some readers will progress through this book from start to finish in a "self-study" course, while others will leap from one chapter to another, depending on their prior R phylogenetics experience, specific questions, or particular topics of interest. As such, we have designed each chapter to "stand apart," in that we reiterate reading input data from file, checking data for completeness, and so on, even if it duplicates computational steps of a prior chapter with the same files. The chapters of the book still do build from beginning to end, so background *explanations* of each R computation or analysis step are not constantly repeated in the text.

We envision that most readers will use this book as a manual or guidebook to undertaking real phylogenetic analyses in an interactive session of R. We picture readers with the book propped open next to their laptop or desktop computer, transcribing (or adapting) our scripts from the book into R. All files that we use in this book are available from download through the book's website,[3,4] so there should be no limit in the reader's ability to follow along.

1.2 Preliminaries

R is at the same time a statistical software, a scientific computing environment, and a programming language.

[3]The book website is https://press.princeton.edu/books/phylogenetic-comparative-methods-in-r; however, for quick access to the files we'll use throughout this volume, readers can refer to http://www.phytools.org/Rbook/.

[4]Readers might notice in frustration that throughout this book, we have used data files that contain phylogenetic trees in different formats, or with mismatched taxa labels, or with data that need to be reorganized or subsampled before analysis. This was an intentional decision, with the aim of helping our readers become more comfortable in working with realistic (and thus sometimes a little *messy*) data sets in the R environment. Please forgive us!

R is distributed free and open source. This means that it is not only free to download and use, but any user or developer can also see the entire source code of the project—and even potentially modify it as they see fit!

1.2.1 The R command line

Although R can be intimidating, most users of R are not doing R programming and can find relatively simple ways to carry out their analyses. However, use of R *does* typically require that you enter commands into a text-based command-line interface, which may be slightly disorienting at first.[5]

One goal of this book in general, and the current chapter in particular, is to help users get comfortable with the commands and language of R.

1.2.2 Packages and resources

The rich functionality of R is built almost entirely on what are called *contributed packages* created by members of the R community of users and developers.

Contributed R packages are best thought of as small libraries of new, usually thematically related R programs known as *functions*. The majority of contributed packages are stored in a public repository called CRAN,[6] an acronym for the *Comprehensive R Archive Network*.

In this chapter, we'll review some of the basics of working with phylogenies in the R environment. We'll assume that the reader has *some* prior experience with R and already knows a little to a lot about phylogenies and the phylogenetic comparative method. Many excellent introductions to R are available both in book form and on the web. Felsenstein (2004) remains an incredible reference for all things phylogeny. Harmon (2019) is (in the humble opinion of the authors) the most comprehensive resource developed to date for phylogenetic comparative analyses.

1.2.3 Code chunks and R output

This chapter and all other chapters in this book have been written by the authors but were assembled using R. As such, all the *gray boxes* consist of what we'll refer to as "code chunks": one or various lines of R script meant to be run in an interactive R session.

All the intervening `courier text` sections and all of the figures are the expected output from R.

That means that to follow along with the R activities of this book, it is possible to simply enter the scripts from the gray boxes into your R session and run them. In fact, this is what we'd recommend!

1.2.4 Entering R commands using a GUI

Rather than typing the commands directly into your R interactive session command prompt, we always suggest entering your R commands first into a scripting window and then executing the code in R.[7]

[5] Particularly for computer users raised in a post-MS-DOS world!

[6] https://cran.r-project.org/.

[7] This is easiest to do from within an R *graphical user interface* or GUI, such as *RGui* for Windows or *RStudio* (*RStudio* Team 2020) on pretty much any platform.

This is a good habit to get into not only when learning how to use R for the first time but also down the road when you begin to apply R to analyze your own data. That is because doing so will permit you to easily save all the commands that we've run in R so that you can readily review them, modify them, and rerun them later if necessary. It also permits you to easily publish all the steps of your data analysis alongside your scientific papers or reports, facilitating reproducibility of research by your peers.

Once you have entered your commands into a scripting window, you do not need to copy and paste your code from one window to the other. Instead, most R graphical user interfaces (GUIs) permit us to directly execute lines from our script in our R session with a simple shortcut. In the R Windows GUI (*RGui*), this can be done by typing CTRL-R with the cursor located on the line you want to execute or with various lines selected and highlighted. In *RStudio* for Windows, the shortcut is CTRL-ENTER, whereas in *RStudio* on a Mac, it is Command-ENTER.

1.3 R phylogenetics

R phylogenetics is built on the contributed packages for phylogenetics in R, and there are many such packages. A partial list of the R packages that contain phylogeny-related functionality is available on a website called the *CRAN phylogenetics task view*.[8]

In this book, we'll only be working with a subset of these packages.

1.3.1 Installing R packages and checking version numbers

We can begin by installing a few of the most critical of R phylogenetics packages: *ape* (Paradis et al. 2004; Paradis and Schliep 2019), *phangorn* (Schliep 2011), *phytools* (Revell 2012), and *geiger* (Harmon et al. 2008; Pennell et al. 2014). To ensure that we get the most recent CRAN package versions, we need to have the most up-to-date R version installed on our computer!

In an interactive R session, it's pretty straightforward to see which R version you have installed.

At the time of writing, the most recent version of R was version 4.1.1.[9]

```
R.version
```

```
##                _
## platform       x86_64-w64-mingw32
## arch           x86_64
## os             mingw32
## system         x86_64, mingw32
## status
## major          4
## minor          1.1
## year           2021
## month          08
## day            10
```

[8]https://cran.r-project.org/web/views/Phylogenetics.html.
[9]Although it will certainly be long out of date by the time this book arrives to your shelf!

```
## svn rev        80725
## language       R
## version.string R version 4.1.1 (2021-08-10)
## nickname       Kick Things
```

Next, let's proceed to install the various packages that we intend to use in this chapter. This can be done easily using the R function `install.packages` as follows.[10]

```
install.packages("ape")
install.packages("phangorn")
install.packages("phytools")
install.packages("geiger")
```

We can proceed to verify the package versions that we've installed by using the *base* R function `packageVersion`:

```
packageVersion("ape")

## [1] '5.5'

packageVersion("phangorn")

## [1] '2.7.1'

packageVersion("phytools")

## [1] '0.7.96'

packageVersion("geiger")

## [1] '2.0.7'
```

Some packages are updated frequently, others less often, but you shouldn't be surprised to see a mismatch between the versions shown above and the package versions you have installed on your computer. Just be aware that sometimes errors can result from using packages that are out of date and thus incompatible with one another.

Installing automatically from CRAN using `install.packages` installs not only your target package but also any libraries on which that package *depends*, if that package has not yet been installed.

For instance, a package dependency of R package *B* on R package *A* means that package *B* uses functions of *A* "internally" (that is, inside of its own functions). Consequently, use of package *B* requires that *A* be installed and loaded. Fortunately, R takes care of these details for us. If a dependent package can't be found or loaded, R will give an error warning us that the missing package needs to be installed.

[10]This works for packages that are on CRAN, which covers most common R packages for comparative methods. Some other packages we use in this book must be installed from GitHub using the R package *devtools*.

1.4 *ape* and the "phylo" object in R

Now we've installed some critical R phylogenetics packages (*ape*, *phangorn*, *phytools*, and *geiger*).

The most important "core" package for phylogenies in R is called *ape* (Paradis et al. 2004; Paradis and Schliep 2019), which stands for **A**nalysis of **P**hylogenetics and **E**volution in R.[11]

1.4.1 Loading the *ape* package

Although we *installed* our main R phylogenetics packages, to make best use of a contributed package, we must proceed to *load* it in our current R session.

Here, we'll do this using the *base* function `library` as follows.[12]

```
library(ape)
```

1.4.2 Reading a phylogenetic tree file

ape does many different things. To get started, let's read a "toy" phylogenetic tree of vertebrates from a relatively simple Newick text string.[13]

```
text.string<-
    "(((((Robin,Iguana),(((Cow,Whale),Pig),Bat),
    (Lemur,Human))),Coelacanth),Goldfish),Shark);"
vert.tree<-read.tree(text=text.string)
```

1.4.3 Plotting a phylogenetic tree

We can plot this tree in our R session using the *ape* package `"phylo"` S3 `plot` method as follows.[14] We see the result in figure 1.1.

[11] A good way to think of what makes *ape* a core package in phylogenetics also has to do with dependency relationships between packages. *Many* other R phylogenetics packages depend on *ape*, or depend on packages that depend on *ape*, while *ape* does not itself depend on other phylogenetics packages.

[12] Note that a highly similar function called `require` will do pretty much exactly the same thing. `library` and `require` are subtly different, but for our purposes, they are interchangeable, and you should feel free to use whichever one you prefer!

[13] A *Newick* string—named, believe it or not, after a lobster restaurant in New Hampshire—is a simple way to encode a phylogenetic tree using a series of nested parentheses. More closely nested species are more closely related. For instance, the simple Newick tree *((chimp,human),gorilla);* tells us that the operational taxa *chimp* and *human* are more closely related to each other than either is to *gorilla*. There are other ways that phylogenetic trees can be represented in machine-readable text, but the Newick string is by far the most common.

[14] The terminology *S3 method* refers to a way that R uses to assign a generic function to an object class. This is helpful, because if our object is a set of points in two dimensions and we send this object to `plot`, R knows—unless we tell it otherwise—to draw a scatterplot. Likewise, if our object is a phylogeny, R knows to draw a tree. Commonly used methods are `plot`, `print`, `summary`, and `predict`, but there are many others, and it's even possible for R programmers to develop their own new generic methods! One tricky aspect of S3 generic methods is that lazy R programmers can develop methods for new object classes without documenting them—so long as the arguments are nominally equivalent.

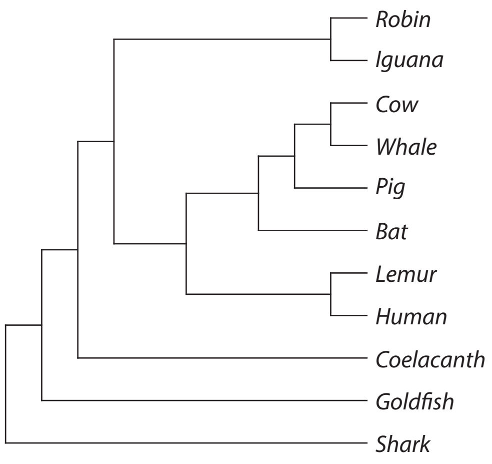

Figure 1.1
A simple phylogenetic plot of vertebrate species drawn with the *ape* method `plot.phylo`.

```
plot(vert.tree,no.margin=TRUE)
```

1.4.4 Getting help for an R function

It's easy to identify ways in which this plot might be improved. For instance, perhaps the lines could be thicker, the font size larger, the margins smaller, and so on. In fact, all of these options are available in the function.

In general, to see the help page for a function, you can call the function `help`[15] on the name of the function you need help with: in our case, `plot`.

```
help(plot)
```

As we are using an S3 method to plot, however, we have to do something different. If we want to see the help page for the `plot` function applied to *phylo* objects, we must run[16]

```
help(plot.phylo)
```

1.4.5 Function arguments and values

Help pages in R are very useful for novice and experienced R users alike. They have a standardized format that details what *arguments* the function takes as input, what the function does, and what *value* the function returns to the user.

Function arguments are best thought of as the *options* and *inputs* of the function. These might include our data, as well as any specifications that the function needs to run our analysis or to generate a plot.

[15]Just entering `?function_name` at the command prompt in R will have the same effect—and is quicker too.

[16]This is generally true for S3 methods. That is to say, if the method has been documented for a particular object class, this documentation will be found at `nameOfMethod.classOfObject`.

The function value is what is returned by the function. For some functions, all results are printed to screen or used to make a graph. Many functions, however, return the results of their execution to the user in the form of one or more numerical values or a special object.

Once we've familiarized ourselves with a function via its help page, it is often useful to use the helper base R function `args` in interactive sessions to obtain a list of the arguments that the function accepts:

```
args(plot.phylo)
```

```
## function (x, type = "phylogram",
## use.edge.length = TRUE, node.pos = NULL,
## show.tip.label = TRUE, show.node.label =
## FALSE, edge.color = "black",
## edge.width = 1, edge.lty = 1, font = 3,
## cex = par("cex"),
## adj = NULL, srt = 0, no.margin = FALSE,
## root.edge = FALSE,
## label.offset = 0, underscore = FALSE,
## x.lim = NULL, y.lim = NULL,
## direction = "rightwards", lab4ut = NULL,
## tip.color = "black",
## plot = TRUE, rotate.tree = 0, open.angle
## = 0, node.depth = 1,
## align.tip.label = FALSE, ...)
## NULL
```

1.4.6 Different ways to plot a phylogenetic tree

Reviewing the help page for `plot.phylo`, as well as the long list of function arguments listed above, suggests that we can visualize our phylogenies in R in a remarkably large number of different ways, even just using this function (and thus not considering all the other various contributed package functions designed to plot phylogenies).

To see this, let's plot our phylogeny in three different styles (figure 1.2).

```
par(mfrow=c(2,2),mar=c(1.1,1.1,3.1,1.1))
plot(vert.tree)
mtext("(a)",line=1,adj=0)
plot(vert.tree,type="cladogram")
mtext("(b)",line=1,adj=0)
plot(unroot(vert.tree),type="unrooted",
    lab4ut="axial",x.lim=c(-2,6.5),
    y.lim=c(-3,7.5))
mtext("(c)",line=1,adj=0)
```

Now why don't we have a look at what this code does line by line.[17]

[17] We'll try to do this as much as possible throughout the book.

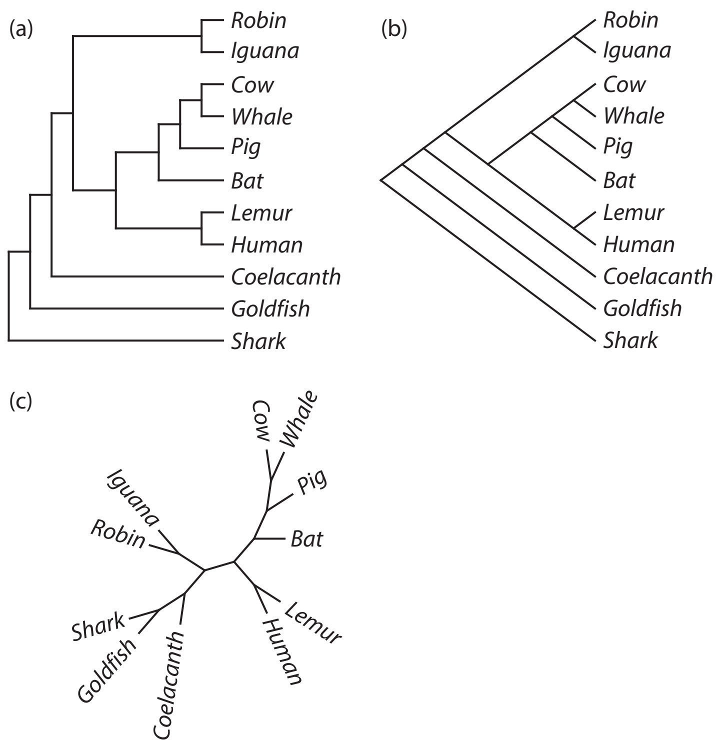

Figure 1.2
A phylogeny plotted in three different styles. (a) A right-facing square cladogram/phylogram. (b) A slanted cladogram/phylogram. (c) An unrooted style. All three graphs were drawn using the *ape* plotting method, `plot.phylo`.

The first line, `par(mfrow=c(2,2),mar=c(1.1,1.1,3.1,1.1))`, tells R to divide our plotting device into four subplots for a 2 × 2 grid—done via the argument `mfrow`. By way of the argument `mar`, it also tells R to set the margins to custom values. The order of this vector is *bottom*, *left*, *top*, and *right*—so we see that we are setting all the margins to 1.1 units, except the upper margin, which we set to 3.1.[18]

The second line, `plot(vert.tree)`, plots the tree using the S3 `plot` method with most of its default arguments—but changed `font=1` to print the tip labels in regular (instead of *italic*) font.

The third line, `mtext("(a)",line=1,adj=0)`, adds a subplot label (“(a)”).[19] The `mtext` argument `line=1` tells R to put the text one line above the figure margin, while the argument `adj=0` tells R to align the text to the left of the plot area.

Finally the fourth, fifth, sixth, and seventh lines repeat the same pattern for each of the subplots but with different plotting styles: first a slanted cladogram (`type="cladogram"`) in figure 1.2b and then an unrooted tree (`type="unrooted"`) in figure 1.2c.[20] We also adjusted

[18] We'll see a lot more of `par` throughout the book.

[19] The function name `mtext` is short for *margin text*.

[20] In the lattermost of these, the argument `lab4ut="axial"`—which stands for *labels for unrooted tree*—tells R to orientate the tip labels in the same direction as the terminal branches of the phylogeny. Who would've guessed? This is what help pages are for!

the *x* and *y* limits of the plot (using the arguments `x.lim` and `y.lim`, respectively)—but this was simply because we discovered that for `type="unrooted"` with `lab4ut` turned on, R was cutting off some of our taxon labels. The specific values that we used are idiosyncratic to the particular tree we're plotting—but these arguments are nonetheless useful to remember should the readers find themselves in a similar situation or want to leave space around their plotted tree for any other reason.[21]

1.5 The internal structure of a tree in R

When we read any phylogeny from file or from a text string (as we did in the previous section), we create an object in the R workspace.

Normally, it won't be necessary to interact directly with this object's internal structure. Instead, we usually pass the object unchanged to other functions—such as when we plotted our phylogeny in different styles to create figure 1.2.

Nonetheless, we believe that for users who commonly work with phylogenies in the R environment, it can be extremely useful to develop a basic working understanding of the structure of phylogenetic objects in memory during an interactive R session.[22]

1.5.1 Trees as lists

This object—that is, the one created in memory when we simulate or estimate a phylogeny or read one from an input file—is a *list* of class `"phylo"`.

In R, a list is just a customizable object type that can combine two or various objects of different types.

For instance, a list might contain a vector of real numbers (with mode `"numeric"`) as its first element, then a vector of strings (with mode `"character"`) as its second element, and so on.

Lists are virtually endlessly flexible, because they can also include other lists[23] (and even functions) among their elements.

Assigning our phylogenetic list with a special class, `"phylo"`, is just a convenient way to tell other functions in R, particularly S3 methods, how to treat that object.

1.5.2 Elements of the `"phylo"` list

An object of class `"phylo"` always consists of at least three elements.

These components of the object are normally "hidden" from view. That is to say, just typing the name of your `"phylo"` object does not reveal the structure of the object in memory, as it would for a standard list in R.

```
vert.tree
```

```
##
## Phylogenetic tree with 11 tips and 10 internal nodes.
```

[21] Such as to add additional graphical elements or features to the plot later; see chapter 13.

[22] In fact, we estimate that if we had a penny for every *geiger* or *phytools* user issue that could have been resolved through knowledge of the structure of the `"phylo"` object, we'd have at least two dollars!

[23] Or lists of list, or lists of lists of lists, and so on.

```
##
## Tip labels:
##   Shark, Goldfish, Coelacanth, Human, Lemur, Bat, ...
##
## Rooted; no branch lengths.
```

What's happened here? Why do we see a summary of the object instead of its structure?

What has occurred is that something called an S3 `print` method has been activated to (guess what?) print a *summary* of some of the important attributes of that object.

In the case of a `"phylo"` object, this summary is designed to give us a printout of the number of terminal taxa in the tree and a list of some of their labels.

R lets us, however, reveal the internal structure of this (and, in fact, virtually any) R object using the handy function `str`[24] as follows.

```
str(vert.tree)
```

```
## List of 3
## $ edge : int [1:20, 1:2] 12 12 13 13 14 14 15
17 21 21 ...
## $ Nnode : int 10
## $ tip.label: chr [1:11] "Shark" "Gold_fish"
"Coelacanth" "Human" ...
## - attr(*, "class")= chr "phylo"
## - attr(*, "order")= chr "cladewise"
```

This tells us that our `vert.tree` object is a list composed of (in this case) three different elements, along with a couple of different attributes.

More specifically, the different parts of our object include

1. `edge`: a 20×2 (in this case) matrix containing starting and ending indices for the nodes subtending each branch of the phylogeny. By convention, tip nodes (that is, those corresponding to species or operational taxa) are numbered 1 through N for N species, while internal nodes are numbered $N + 1$ (at the root) through $N+$ the number of internal nodes.[25]
2. `Nnode`: an integer value giving the total number of internal nodes in the tree.
3. `tip.label`: a character vector of length N containing the labels for all the tips or terminal taxa in the phylogeny.

1.5.3 Node indices

Now let's see how these different components relate to the structure of the tree by replotting our phylogeny, but this time overlaying the numerical indices from the matrix `edge` onto the nodes and terminals of the tree (figure 1.3). We can do that in R as follows.

[24] Short for *structure*.

[25] There will be $N - 1$ of these if our tree is both rooted and perfectly bifurcating. An unrooted, bifurcating tree has $N - 2$ internal nodes. Trees with polytomies can have fewer nodes still, while trees with unbranching nodes can have more.

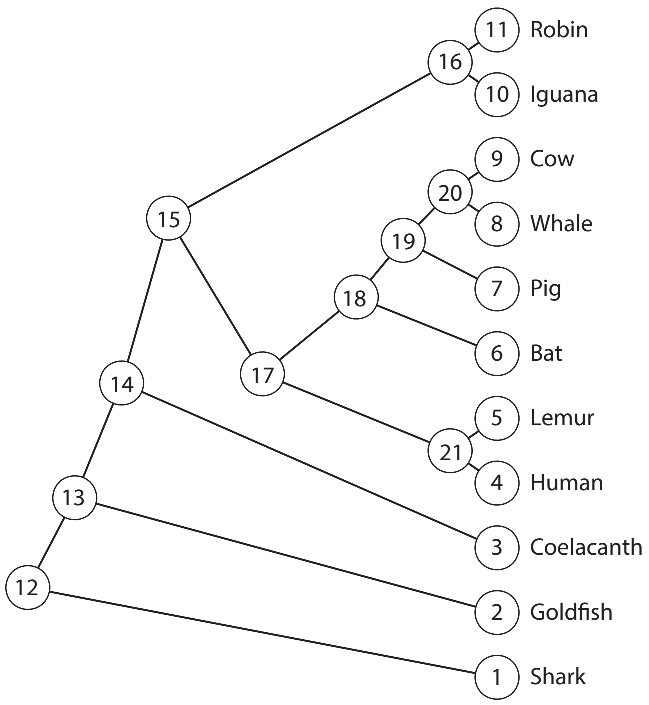

Figure 1.3
A simple phylogeny of vertebrate species with nodes labeled by their indices in the `"phylo"` `edge` matrix. We created the plot using the *phytools* function `plotTree` and added node labels using `labelnodes` (although the latter could have also been done just as easily with the *ape* function `nodelabels`).

```
library(phytools)
plotTree(vert.tree,offset=1,type="cladogram")
labelnodes(1:(Ntip(vert.tree)+vert.tree$Nnode),
    1:(Ntip(vert.tree)+vert.tree$Nnode),
    interactive=FALSE,cex=0.8)
```

Reviewing our code line by line, we first loaded an additional R package called *phytools* (`library(phytools)`).

We then plotted our tree, but instead of using the S3 method, we elected[26] to use the *phytools* function `plotTree`.

Finally, we used the *ape* function `labelnodes` to add numerical labels to all the internal and "external" (that is, tip) nodes of the phylogeny.[27]

Just to reiterate, here what we have done is simply *plotted* our tree, and then we've *overlain* the "node numbers" onto the plotted tree. The node numbers are simply the indices from the `"phylo"` object element `edge`, which is itself a matrix containing the starting and ending indices for each branch of the phylogeny!

[26]For no particular reason.

[27]We could have also used the *ape* functions `nodelabels()` and `tiplabels()` without any arguments—but that doesn't look quite as nice, in our opinion. Try it and see if you agree!

```
vert.tree$edge
```

```
##       [,1] [,2]
##  [1,]   12    1
##  [2,]   12   13
##  [3,]   13    2
##  [4,]   13   14
##  [5,]   14    3
##  [6,]   14   15
##  [7,]   15   17
##  [8,]   17   21
##  [9,]   21    4
## [10,]   21    5
## [11,]   17   18
## [12,]   18    6
## [13,]   18   19
## [14,]   19    7
## [15,]   19   20
## [16,]   20    8
## [17,]   20    9
## [18,]   15   16
## [19,]   16   10
## [20,]   16   11
```

If we now go ahead and compare `vert.tree$edge` to our plot in figure 1.3, we should see that each *row* of the matrix corresponds to one and only one branch in the tree. In other words, the edge matrix completely represents the topology of our tree using a simple table!

We should also notice

1. that `edge` has a number of rows that are *equal* to the number of branches (20) in this phylogeny, and
2. that each branch starts and ends with a unique pair of indices, just as we learned above.

1.5.4 Tip labels and node counts of a phylogeny

As we already saw, the other components of our `"phylo"` object include the vector `tip.label` and an integer `Nnode`, which gives the number of interior nodes in the tree.

Let's take a look at these two elements now as well.

```
vert.tree$tip.label
```

```
##  [1] "Shark"      "Goldfish"  "Coelacanth"
##  [4] "Human"      "Lemur"      "Bat"
##  [7] "Pig"        "Whale"      "Cow"
## [10] "Iguana"     "Robin"
```

```
vert.tree$Nnode
```

```
## [1] 10
```

By convention, the *order* of the tip labels in `tip.label` corresponds to the numerical order of the tip indices (scored from 1 through *N*, remember) in our phylogeny.

The component `Nnode` has an even more straightforward interpretation—which we think doesn't require any additional explanation.

1.5.5 The `"phylo"` class

An object of class `"phylo"` also (by definition) has at least one attribute—its class. This is just a value to tell various functions—and, particularly, S3 methods—in R what to do with an object of this type.

For instance, if we call the generic method `plot`, the object class attribute is what instructs R to use the method `plot.phylo` that has been exported by the R package *ape*.

An object of class `"phylo"` can have other components too. The most common of these is `edge.length`: a vector of class `"numeric"` containing all the branch lengths or our tree. Although our object `vert.tree` does not include branch lengths, if it did, we would see that the numeric vector `edge.length` contained the branch lengths of the phylogeny in precisely the order of the rows of `edge`.

In addition, other elements and attributes can be added for special types of phylogenetic trees. Some R functions will behave differently if these additional elements or attributes are present in our `"phylo"` object. We'll see more about this in later chapters of the book!

1.6 Reading and writing phylogenetic trees

Naturally, R can easily read and write trees to and from files.

1.6.1 Reading a tree from a file

For example, let's download the tree file `Anolis.tre` (Mahler et al. 2010, available from the book's website[28]) and read it into R.

For this task, we'll use the *ape* function `read.tree`.[29]

As soon as you have the tree file in your current working directory in R,[30] you can read it in

```
anolis.tree<-read.tree(file="Anolis.tre")
anolis.tree
```

```
##
## Phylogenetic tree with 100 tips and 99 internal nodes.
##
## Tip labels:
##   ahli, allogus, rubribarbus, imias, sagrei, bremeri, ...
```

[28]The site is http://www.phytools.org/Rbook/, as indicated earlier. Henceforward, we'll only provide the URL of the book website on the first instance that it's referenced in each chapter.

[29]`read.tree` and, likewise, `read.newick` in the *phytools* package read phylogenies in *simple Newick format*. Different functions of *ape*, *phytools*, and other packages can be used to read trees that have been written to file in other formats.

[30]To see your current working directory in R, type `getwd()` at the command prompt. To change your working directory, use the function `setwd`.

```
##
## Rooted; includes branch lengths.
```

```
plotTree(anolis.tree,ftype="i",fsize=0.4,lwd=1)
```

This is a tree containing

```
Ntip(anolis.tree)
```

```
## [1] 100
```

100 species of lizards in the neotropical lizard genus *Anolis* (figure 1.4).

1.6.2 Writing a tree to a file

In addition to *reading* a tree from file, we can also write them. For instance, we can easily write our vertebrate tree from earlier in the chapter to a simple text file in Newick format.

```
write.tree(vert.tree,file="example.tre")
```

This is what the resultant text file `example.tre` should look like.[31]

```
cat(readLines("example.tre"))
```

```
## (Shark,(Gold_fish,(Coelacanth,(((Hu ...
```

1.7 Plotting and manipulating trees

We've already seen a few in this chapter, but there are a wide range of ways in which we can plot and manipulate trees in R.

Next, let's take a look at a few more of the most common ways that phylogenies are plotted in R.[32]

Meanwhile, we can also see how R can be used to (for lack of a better word) manipulate phylogenies.

Common types of manipulation of phylogenies in R include dropping or "pruning" species from a tree, extracting subtrees, and shrinking or stretching trees to have a particular total length. We'll focus on the former two types of manipulation here.

1.7.1 Pruning taxa from the phylogeny

A convenient and popular plotting method for large *rooted* trees is a circular or "fan" tree. We can start by plotting our *Anolis* tree in this way and then go from there.

[31] You can also open the file on your computer using any text editor to check if you'd like.

[32] R is an extremely flexible plotting environment, so there are many plotting options that we are not seeing here; however, some of these will be visited in subsequent chapters.

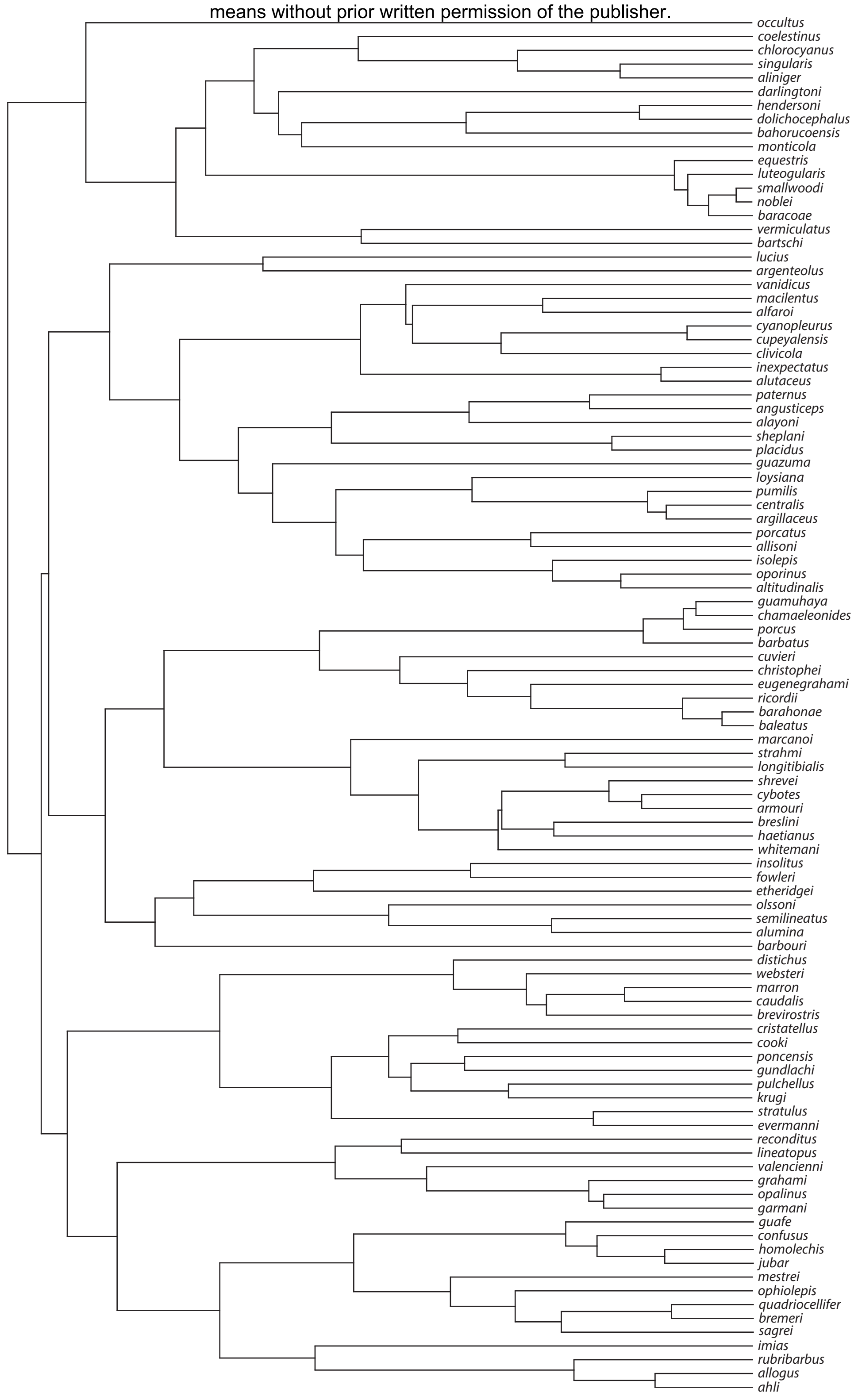

Figure 1.4
A phylogenetic tree of *Anolis* lizards plotted in a right square phylogram style using the *phytools* function `plotTree`.

In comparative analyses with phylogenetic data, we are often called upon to prune species out of the tree or to extract one clade or another.

This might be the case, for instance, when we have phylogenetic data for one set of taxa and morphological, phenotypic, or biogeographic data for a different, but nonetheless overlapping, set. Fortunately, pruning taxa and extracting clades are relatively straightforward operations in an interactive R session.

Let's imagine, for instance, that instead of working with the 100-taxon *Anolis* tree, we would like to analyze a phylogeny that contained only a subset of these taxa.[33] We can focus our attention on the anoles from Puerto Rico, which (in this phylogeny) consist of *A. cristatellus*, *A. cooki*, *A. poncensis*, *A. gundlachi*, *A. pulchellus*, *A. stratulus*, and *A. evermanni* (which form a clade), as well as *A. occultus* and *A. cuvieri*.

As a first step, let's find these Puerto Rican anoles on our complete phylogeny.

The following script uses a *phytools* function called `add.arrow` to add red[34] arrows pointing to particular tips on the phylogeny that we are interested in.

We have to plot the arrows by indicating the tip *numbers* (not the labels) of the terminal taxa that we want to mark. To find these, we will use the R *base* function `grep`.

`grep` matches a character pattern to a vector and returns the positions of the elements of the vector in which that pattern is found. Here, we're going to use it to match the specific epithets[35] of the Puerto Rican anoles to the vector comprising all the tip labels of the tree.

Perhaps the reader is beginning to see how useful it can be to know something about the structure of the `"phylo"` object—because otherwise, we might not know that these labels can be found in the vector `anolis.tree$tip.label`!

```
pr.species<-c("cooki","poncensis",
    "gundlachi","pulchellus","stratulus",
    "krugi","evermanni","occultus","cuvieri",
    "cristatellus")
nodes<-sapply(pr.species,grep,x=anolis.tree$tip.label)
nodes
```

```
##        cooki    poncensis    gundlachi
##           26           25           24
##   pulchellus    stratulus        krugi
##           23           21           22
##    evermanni     occultus      cuvieri
##           20          100           54
## cristatellus
##           27
```

[33]Although *Anolis* is a clade with over 400 described species across the tropical and subtropical Americas, our phylogeny includes only representatives from the Greater Antillean region of the Caribbean.

[34]Here and throughout the volume, we'll refer to the colors that would be plotted if you reproduced our code in R. The figures that you'll *actually* see in the print version of the book, however, have been recolored in grayscale by the publisher to help ensure that the book can be printed, and sold, at a reasonable price. Hopefully this isn't too confusing!

[35]The *specific epithet* is the second part of the Latin binomial name of a species, so for the species *Homo sapiens*, the specific epithet is *sapiens*. In our *Anolis* tree, all tips belong to the same genus, so they've been labeled using *only* the epithet.

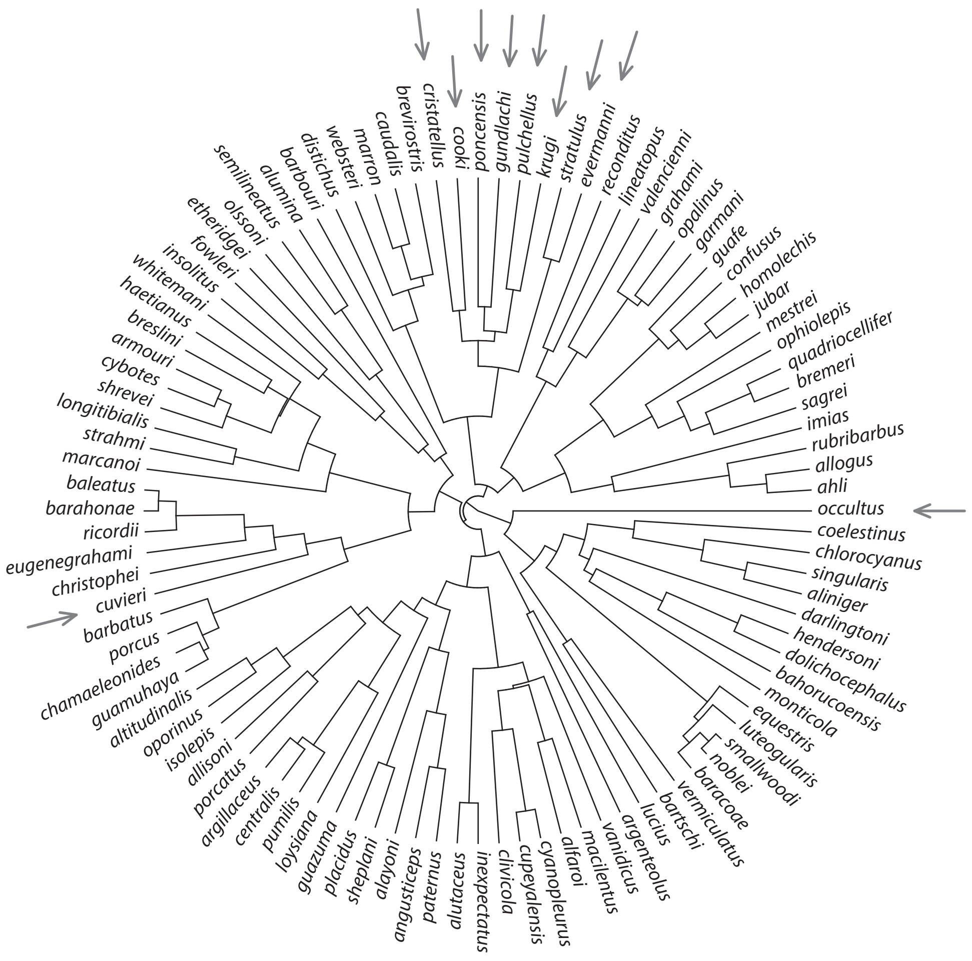

Figure 1.5
Phylogenetic tree of *Anolis* lizards. We plotted the tree using the *phytools* function `plotTree` and then highlighted the species from Puerto Rico using the function `add.arrow`.

Note that we've used a function belonging to the so-called `apply` family of functions—this one called `sapply`. `apply` family functions are designed to iterate operations over the elements of a matrix, vector, or list, without the necessity of writing a loop.[36]

The easiest way to interpret our `sapply` call, `nodes <- sapply(pr.species, grep, x=anolis.tree$tip.label)`, is as "apply to the elements of `pr.species` the function `grep` with the argument `x` of `grep` set to `anolis.tree$tip.label`."

We'll see more uses of various `apply` family functions throughout this chapter and the rest of the book.

Now that we have identified the tip node numbers of all the Puerto Rican *Anolis* lizards in our tree, we can plot our tree and label these species using arrows just as we planned. The result is seen in figure 1.5.

[36]The most common loop programming structure in R, as well as in many other programming languages, is called a *for* loop. *for* loops can be very useful in R. We'll see examples of *for* loops later on the book.

```
plotTree(anolis.tree,type="fan",fsize=0.6,lwd=1,
    ftype="i")
add.arrow(anolis.tree,tip=nodes,arrl=0.15,col="red",
    offset=2)
```

The orientation of the arrows in figure 1.5 should exactly match the orientation of the tip branch.[37]

Now let's prune the species that we marked with arrows out of the tree.

```
anolis.noPR<-drop.tip(anolis.tree,pr.species)
plotTree(anolis.noPR,type="fan",fsize=0.6,lwd=1,
    ftype="i")
```

We should see that the function we used here, `drop.tip`, cuts not only the terminal branch but any branch that leads exclusively to tips that are being pruned. We've plotted the pruned phylogeny in figure 1.6.

1.7.2 Extracting a clade

Alternatively, let's imagine that we want to *extract* the main clade of Puerto Rican *Anolis* species. In our example, this is the clade that includes all but two of the species found on the island.

To extract a clade, we need to identify the node index of the *most recent common ancestor* (MRCA) of the members of the clade we want to prune.

In our case, this corresponds to the MRCA of all the species *except* for *Anolis cuvieri* and *Anolis occultus*.[38] We can find the node number of the MRCA of a set of taxa using `getMRCA` from the *ape* package.

```
node<-getMRCA(anolis.tree,pr.species[
    -which(pr.species%in%c("cuvieri","occultus"))])
node
```

```
## [1] 123
```

Just for fun, before we pull it out, let's go ahead and visualize the clade that we plan to extract.

To do so, we'll use the *phytools* function `paintSubTree`.[39] We can also combine this with the function `arc.cladelabels` to add a nice clade delimiter to our plot. The result is shown in figure 1.7.

[37]This is harder to guarantee when we make this kind of figure using a point-and-click image editor!

[38]Here we used *negative indexing* to pull out undesired elements from our vector of taxon names. Negative indexing returns all the elements of a vector, matrix, or list with the exception of those that were indexed. Does that make sense?

[39]This function will also reappear later in the book!

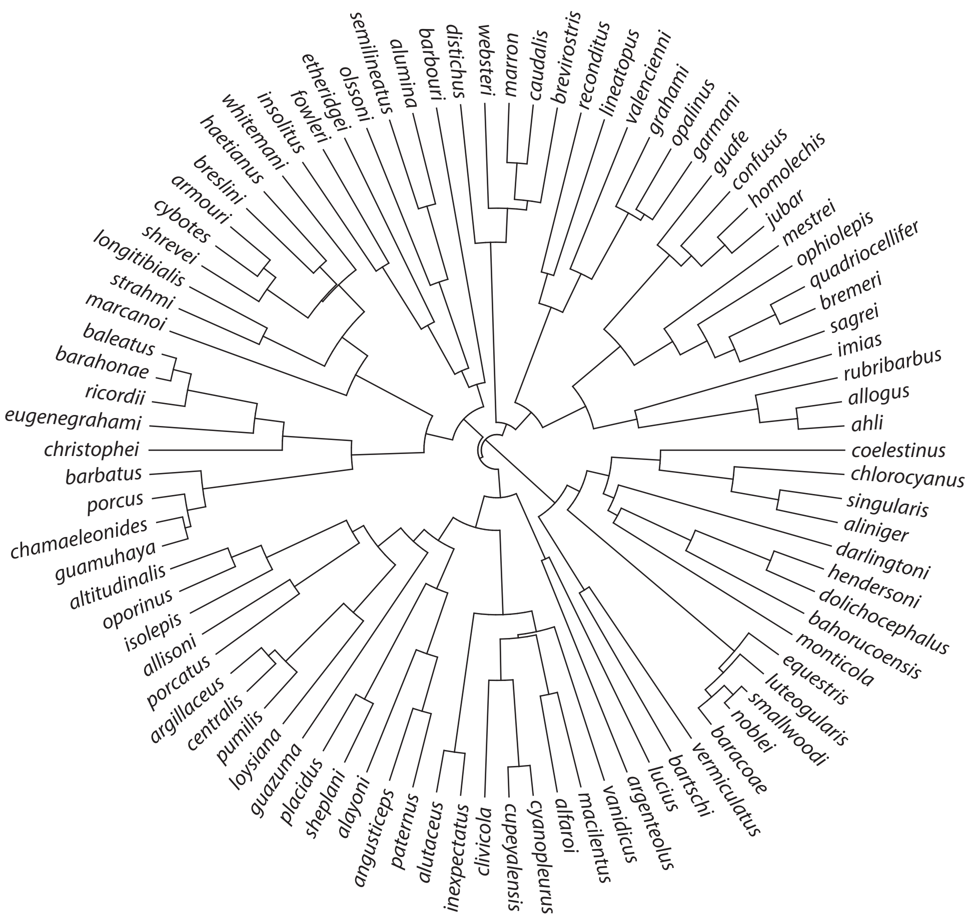

Figure 1.6
Phylogeny of *Anolis* in which we first pruned all Puerto Rican species from the tree using the *ape* function `drop.tip`.

```
plot(paintSubTree(anolis.tree,node,"b","a"),
    type="fan",fsize=0.6,lwd=2,
    colors=setNames(c("gray","blue"),c("a","b")),
    ftype="i")
arc.cladelabels(anolis.tree,"clade to extract",node,
    1.35,1.4,mark.node=FALSE,cex=0.6)
```

The numbers `1.35` and `1.4` have no special significance—in this case, they merely set the relative offset of the clade line and the clade label from the tips of the tree. Readers who want to duplicate this plot with their own phylogeny will probably have to use different values (although sometimes the defaults can work fairly well).

Now, we can proceed to extract the clade of interest using the *ape* function `extract.clade` as follows.

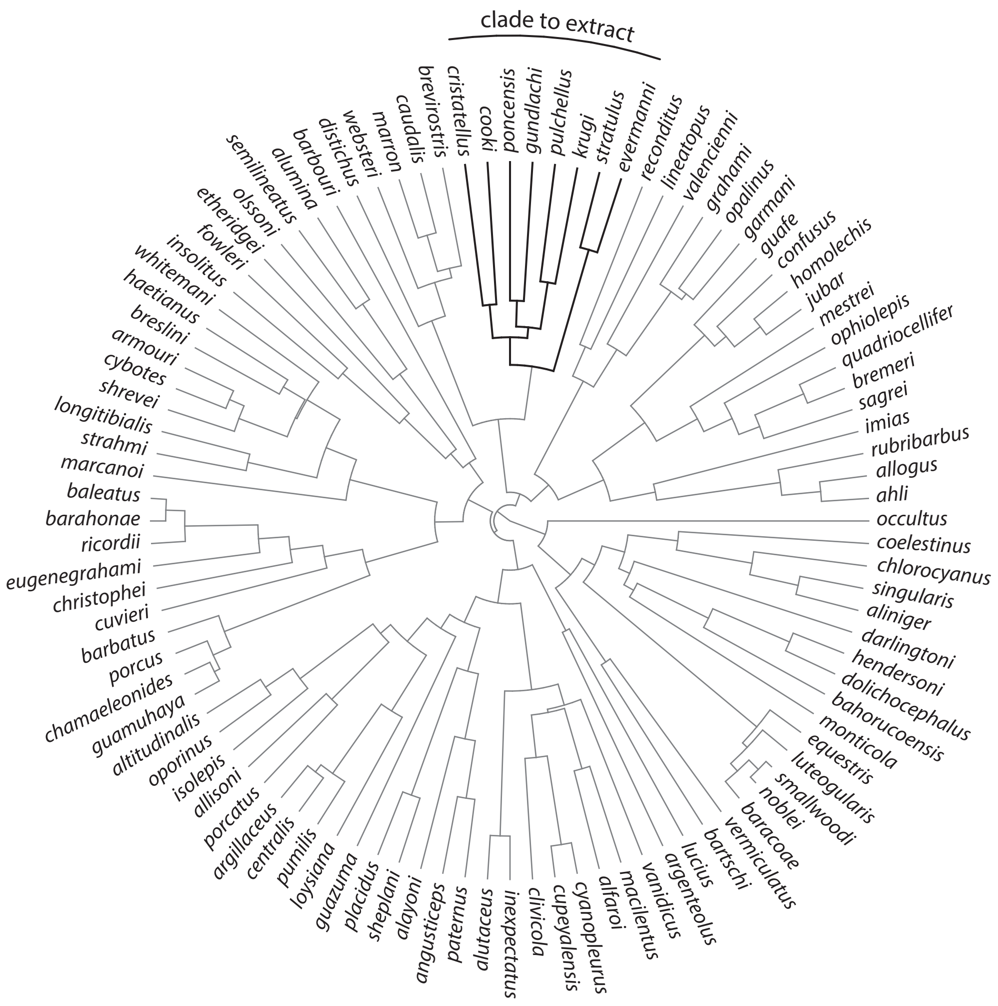

Figure 1.7
Tree of *Anolis* lizards. Here we marked the part of the tree we plan to extract by mapping this clade onto our `"phylo"` object with the *phytools* function `paintSubTree` and then by drawing the tree using `plot.simmap`.

```
pr.clade<-extract.clade(anolis.tree,node)
pr.clade
```

```
##
## Phylogenetic tree with 8 tips and 7 internal
nodes.
##
## Tip labels:
## evermanni, stratulus, krugi, pulchellus,
gundlachi, poncensis, ...
##
## Rooted; includes branch lengths.
```

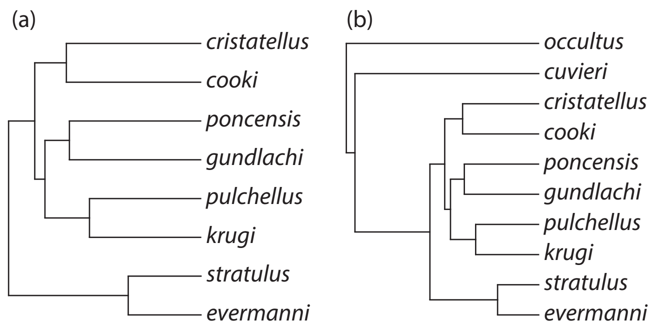

Figure 1.8
(a) Largest clade from Puerto Rico extracted from the full tree of Caribbean *Anolis* lizards using the *ape* function `extract.clade`. (b) Phylogeny containing all of the *Anolis* the species from Puerto Rico obtained from the full tree using the function `keep.tip`.

Likewise, just as we extracted the clade, we can also perform the converse operation—which would be to prune everything in the tree except for these species.

To do this, we will use a *different ape* function called `keep.tip`.

```
pr.tree<-keep.tip(anolis.tree,pr.species)
pr.tree
```

```
##
## Phylogenetic tree with 10 tips and 9 internal
nodes.
##
## Tip labels:
## evermanni, stratulus, krugi, pulchellus,
gundlachi, poncensis, ...
##
## Rooted; includes branch lengths.
```

Here are the two resultant phylogenies (figure 1.8).

```
par(mfrow=c(1,2))
plotTree(pr.clade,ftype="i",mar=c(1.1,1.1,3.1,1.1),
    cex=1.1)
mtext("(a)",line=0,adj=0)
plotTree(pr.tree,ftype="i",mar=c(1.1,1.1,3.1,1.1),
    cex=1.1)
mtext("(b)",line=0,adj=0)
```

1.7.3 Interactive tree manipulation using collapseTree

Finally, sometimes it's fun to prune our tree *interactively*—by clicking on nodes or tips of the phylogeny after it has been plotted. In R, this can be done using the *ape* function `drop.tip`, which has an interactive mode, or by using the animated *phytools* function `collapseTree` as follows.

```
anolis.pruned<-collapseTree(anolis.tree)
```

Obviously, this cannot be demonstrated on the pages of a book, but please try it out! As you click on or near nodes of the tree, you should see clades collapse or reexpand.[40] When you're done, just right-click and select *stop*.

1.8 Multiple trees in a single object

It's often useful to store multiple phylogenies in a single object. This could be true, for instance, when we have a set of trees in a posterior sample from Bayesian phylogeny inference, if we're working with a bootstrap distribution of phylogenies, or when we want to replicate a simulation analysis across a large number of trees.

1.8.1 The `"multiPhylo"` object

In R, multiple phylogenetic trees are usually stored in the form of an object of class `"multiPhylo"`.

This sounds fancy, but it is really nothing more than a list of objects of class `"phylo"` but with the class attribute `"multiPhylo"` assigned!

Many, but not all, functions in *ape*, *phytools*, and other R packages are "vectorized" so that they can be applied to both `"phylo"` and `"multiPhylo"` objects. For instance:

```
anolis.trees<-c(anolis.tree,anolis.noPR,pr.clade,
    pr.tree)
print(anolis.trees,details=TRUE)
```

```
## 4 phylogenetic trees
## tree 1 : 100 tips
## tree 2 : 90 tips
## tree 3 : 8 tips
## tree 4 : 10 tips
```

```
write.tree(anolis.trees,file="example.trees")
```

Here, we first combined all of our individual trees into a single `"multiPhylo"` object using the function (and S3 method) `c` (short for *combine*).

Next, we printed a summary of our object. Meanwhile, we also turned on the `print` method optional argument `details` so that we could see a bit more information about each tree in the object—in this case, the number of terminal taxa ("tips") in each tree.

Finally, we wrote all the trees to a single text file using the *ape* function `write.tree`.

This output file is merely just a simple text file, with each of our phylogenies written in Newick format onto separate lines:

[40]The animation works better in some R GUIs than others!

```
cat(readLines("example.trees"),sep="\n")
```

```
## (((((((ahli:0.131,allogus:0.131):0 ...
## (((((((ahli:0.131,allogus:0.131):0 ...
## ((evermanni:0.214,stratulus:0.214): ...
## (((evermanni:0.214,stratulus:0.214 ...
```

1.9 Managing trees and comparative data

Throughout this book, we'll often be called upon to manage not only phylogenies but also comparative phenotypic trait data for species.

To see an example of how to do this with real data, let's use two different data files from the book webpage: `anole.data.csv` and `ecomorph.csv` (Mahler et al. 2010).

We'll combine the data of these files with the phylogeny from our *Anolis* tree file (`Anolis.tre`) that we read into R earlier in the chapter.

1.9.1 The CSV file format

Both of our two data files (`anole.data.csv` and `ecomorph.csv`) are written in a common data file format called *CSV*[41] format.

R can read data in lots of different formats; however, CSV format is pretty reliable and widely used. Precisely as you might expect, CSV format is a simple text format for tabular data, but in which the elements in different rows are separated by a hard return, while the elements in different columns within a row are separated by the comma character: `,`.

Let's read our CSV files into R using the function `read.csv` as follows:

```
anole.data<-read.csv(file="anole.data.csv",row.names=1,
    header=TRUE)
ecomorph<-read.csv(file="ecomorph.csv",row.names=1,
    header=TRUE,stringsAsFactors=TRUE)
```

Calls to `read.csv`, like those we've executed here, generate data frames in R.[42]

The argument `row.names=1` tells R to look for row names in the first column of our data file, while `header=TRUE` tells R that the first row of our data file contains the column or variable names.

An astute reader might also notice that in our second `read.csv` call, we were careful to set the argument `stringsAsFactors` to `TRUE`. This is to ensure that the discrete character trait contained in this file was read into R as a multilevel *factor* rather than as a simple set of character strings.[43]

Although we find the CSV format to be a very reliable way to store tabular data, one complication is that in South America and in some parts of continental Europe, the comma (`,`) is used

[41] CSV stands for *comma-separated values*.

[42] A data frame looks like a matrix, but it's technically a *list* arranged in a tabular way such that all the columns of the data frame are vectors with the same number of elements. We'll encounter and work with a lot of data frames in this book!

[43] Prior to R version 4.0, `stringsAsFactors` defaulted to `TRUE`, and this would not have been necessary.

as a decimal separator in place of the period (`.`). As such, it would be impractical to demarcate columns in tabular data using commas. Frequently, then, in these places, the columns of a CSV-formatted text file will have been demarcated using the semicolon character (;).

This is no problem at all for R as we can modify the separator and decimal characters using the `read.csv` function arguments `sep=";"` and `dec=","`, respectively, *or* simply by substituting the function `read.csv2` (which uses these arguments by default).

As an aside, one phenomenon that we have often observed in teaching R phylogenetics is that for many users, their spreadsheet software will be set to open CSV format files automatically by default. As a consequence, students often mistakenly open CSV files in their spreadsheet program and then proceed to resave them in a different format instead of as a genuine CSV file. This should obviously be avoided.[44]

Now that we've read our data into R, let's proceed and use the function `head` to inspect the first few rows of the data frames that we've created and then `dim` to review the dimensions[45] of our two data frames.

```
head(anole.data)
```

```
##              SVL      HL     HLL     FLL     LAM
## ahli     4.03913 2.88266 3.96202 3.34498 2.86620
## alayoni  3.81570 2.70212 3.27950 2.80245 3.07527
## alfaroi  3.52665 2.37816 3.30542 2.48366 2.73387
## aliniger 4.03656 2.89884 3.64623 3.15908 3.15677
## allisoni 4.37539 3.35896 3.96069 3.44620 3.23921
## allogus  4.04014 2.86103 3.94018 3.33829 2.80827
##               TL
## ahli     4.50400
## alayoni  4.07265
## alfaroi  4.41601
## aliniger 4.54173
## allisoni 5.05911
## allogus  4.52189
```

```
dim(anole.data)
```

```
## [1] 100   6
```

```
head(ecomorph)
```

```
##             ecomorph
## ahli              TG
## allogus           TG
## rubribarbus       TG
## imias             TG
## sagrei            TG
## bremeri           TG
```

[44]Some spreadsheet software files can be read directly by R, but this is less reliable and as such we don't really recommend it.

[45]That is, the number of rows and columns, respectively.

```
dim(ecomorph)
```

```
## [1] 82  1
```

Doing this, we can see that our first data frame (`anole.data`) has 100 rows and contains six different numeric variables, with the names `SVL` (snout-to-vent length, incidentally—on a log scale), `HL` (head length), `HLL` (hindlimb length), and so on. Our data frame `ecomorph`, by contrast, has only 82 rows and contains one factor variable (also denominated `ecomorph`). The row names of both data frames contain the taxon labels: in this case, the specific epithets of species of lizard in the genus *Anolis*, just as in our tree.

1.9.2 Comparing a character data set and tree

Although it seems likely that the first data set has the same set of species as our 100-taxon *Anolis* tree from earlier in the exercise, we can (and should!) verify this.

Let's do so using the *geiger* function `name.check`.[46]

```
library(geiger)
name.check(anolis.tree,anole.data)
```

```
## [1] "OK"
```

This result (`"OK"`) tells us that the taxon names in the phylogeny exactly match those of the data frame.

`name.check` is useful not only for identifying incongruencies between the phylogeny and data but also instances in which a taxon label may have been misspelled, mistranscribed, or misread by R in either our data set or the tree.

In the case of `ecomorph`, however, there are obviously fewer observations in the data than in the tree. That suggests that there are at least *some* differences between the data and phylogeny. Let's use `name.check` again to see how they differ.

```
chk<-name.check(anolis.tree,ecomorph)
chk
```

```
## $tree_not_data
##  [1] "argenteolus"    "argillaceus"
##  [3] "barbatus"       "barbouri"
##  [5] "bartschi"       "centralis"
##  [7] "chamaeleonides" "christophei"
##  [9] "etheridgei"     "eugenegrahami"
## [11] "fowleri"        "guamuhaya"
## [13] "lucius"         "monticola"
## [15] "porcus"         "pumilis"
## [17] "reconditus"     "vermiculatus"
```

[46]We could have also used the *geiger* function `comparative.data`, which serves some of the same purposes but works a little differently than `name.check`.

```
##
## $data_not_tree
## character(0)
```

Now we can see that when there are differences between our data and our tree, `name.check` returns a handy list indicating which taxa are in the tree but not the data, as well as vice versa.

For examples with larger discrepancies between data and tree, we can also print an abbreviated summary of our result as follows:

```
summary(chk)
```

```
## 18 taxa are present in the tree but not the data:
##     argenteolus,
##     argillaceus,
##     barbatus,
##     barbouri,
##     bartschi,
##     centralis,
##     ....
##
## To see complete list of mis-matched taxa, print object.
```

1.9.3 Pruning a tree to match your data set, and vice versa

Now, precisely as we learned earlier in the chapter, let's go ahead and prune[47] all the taxa that are present in our phylogeny, but not in our `ecomorph` data frame. This can be done using *ape*'s `drop.tip` function as follows:

```
ecomorph.tree<-drop.tip(anolis.tree,chk$tree_not_data)
ecomorph.tree
```

```
##
## Phylogenetic tree with 82 tips and 81 internal nodes.
##
## Tip labels:
##   ahli, allogus, rubribarbus, imias, sagrei, ...
##
## Rooted; includes branch lengths.
```

We can similarly subsample our data to include only those taxa present in a phylogeny.

Let's do it for our data frame `anole.data` so that it contains only data for the species in our new, pruned phylogeny that we've denominated `ecomorph.tree`.[48]

[47] That is, remove from the phylogeny.

[48] This trick subsamples the data frame to include only rows whose names match `ecomorph.tree$tip.label`—the taxon labels of our tree. One odd behavior of R is that if our data frame has only one column, this operation will return a vector rather than a data frame with one column! This can

```r
ecomorph.data<-anole.data[ecomorph.tree$tip.label,]
head(ecomorph.data)
```

```
##                 SVL      HL     HLL     FLL
## ahli        4.03913 2.88266 3.96202 3.34498
## allogus     4.04014 2.86103 3.94018 3.33829
## rubribarbus 4.07847 2.89425 3.96135 3.35641
## imias       4.09969 2.85293 3.98565 3.41402
## sagrei      4.06716 2.83515 3.85786 3.24267
## bremeri     4.11337 2.86044 3.90039 3.30585
##                 LAM      TL
## ahli        2.86620 4.50400
## allogus     2.80827 4.52189
## rubribarbus 2.86751 4.56108
## imias       2.94375 4.65242
## sagrei      2.91872 4.77603
## bremeri     2.97009 4.72996
```

Our new trait data frame (`ecomorph.data`) should now match our pruned phylogeny exactly—but let's make sure, once again using the function `name.check`:

```r
name.check(ecomorph.tree,ecomorph.data)
```

```
## [1] "OK"
```

This result (`"OK"`) tells us that `name.check` now thinks that our tree and data match exactly!

1.10 A simple comparative analysis: Phylogenetic principal components analysis

Now that our *Anolis* lizard tree and data sets match, let's go ahead and do a very simple analysis called a "phylogenetic principal components analysis" or phylogenetic PCA (Revell 2009) using our morphological character data.

A phylogenetic PCA is exactly the same as a regular PCA except that we're going to take the nonindependence of species into account when we compute the covariances (or correlations) between different traits.

Whereas in regular (nonphylogenetic) PCA, principal components are *orthogonal*,[49] in phylogenetic PCA, components are evolutionarily orthogonal, meaning that the evolutionary correlations[50] between principal components are all zero. Likewise, whereas principal components describe successive orthogonal dimensions of maximum variance in the original multidimensional trait space, *phylogenetic* principal components correspond to successive evolutionarily orthogonal dimensions of maximum evolutionary variance.

be circumvented by using the argument `drop=FALSE` as follows: `ecomorph.data <- anole.data [ecomorph.tree$tip.label„drop=FALSE]`. Isn't that weird?

[49]That is to say, *uncorrelated*.

[50]The evolutionary correlation will be discussed in much greater detail in chapters 2 and 3.

The interpretation of the first phylogenetic principal component is thus that it is the axis of greatest, multivariate *evolution*[51] of our traits. Subsequent axes are successive orthogonal dimensions of maximum evolution.

To undertake a phylogenetic principal component analysis in R, we can use the function `phyl.pca` in the *phytools* package as follows.[52]

```
ecomorph.pca<-phyl.pca(ecomorph.tree,ecomorph.data)
ecomorph.pca
```

```
## Phylogenetic pca
## Standard deviations:
##        PC1        PC2        PC3        PC4
## 0.81375257 0.22561158 0.12277034 0.10577996
##        PC5        PC6
## 0.04926765 0.03692593
## Loads:
##            PC1         PC2         PC3
## SVL -0.9712073  0.16073225  0.01979472
## HL  -0.9644970  0.16959751 -0.01199377
## HLL -0.9814007 -0.02674374  0.10309671
## FLL -0.9712156  0.17590524  0.10692548
## LAM -0.7809539  0.37434869 -0.47406978
## TL  -0.9013706 -0.42546037 -0.07612345
##             PC4         PC5          PC6
## SVL  0.14785037 -0.06199108 -0.069477241
## HL   0.17994467  0.08065005  0.044203206
## HLL -0.13799438  0.06907952 -0.041160294
## FLL -0.09104262 -0.06097041  0.048562708
## LAM -0.15858923  0.00217263 -0.008754817
## TL   0.01713199 -0.01755709  0.010858471
```

```
par(mar=c(4.1,4.1,2.1,1.1),las=1) ## set margins
plot(ecomorph.pca,main="")
```

From this printout, we can see that phylogenetic PC1 loads strongly, and *negatively*, for all of the traits in our data set. This principal component represents evolutionary variation in overall size. Remember that the *sign* of each principal component is arbitrary, so let's flip it. This is easy enough to do in R as follows.

```
ecomorph.pca$Evec[,1]<--ecomorph.pca$Evec[,1]
ecomorph.pca$L[,1]<--ecomorph.pca$L[,1]
ecomorph.pca$S<-scores(ecomorph.pca,
    newdata=ecomorph.data)
```

[51] Under our chosen evolutionary model: more in chapters 4 and 5.

[52] The plotting argument `las=1`, which we can also often set using `par(las=1)` and we use in many places throughout the book, merely sets the axis tick labels to plot horizontally rather than parallel to the axis (`las=0`, the default in R) or vertically (`las=2`).

(continued...)

Index