Computational Biology for All!

Nathan L. BRouwer

2021-09-10

Contents

Preface

Part I

Introduction

Predface

In Book 1 I'll introduce why the role of computers in biology is important and lay out the scope of this book. I'll also walk through the basic steps of getting R and RStudio up and running on your computer, walk through a basic R session, and discuss the various ways we'll interact with R using **code** and **script files**. I'll also introduce **rmarkdown**, a way of integrating the R code of scripts with basic capabilities of word processing and web development. Scripts organized in rmarkdown are a handy way to organize and **annotate code**, store and disseminate your work, and structure your analyses and data presentations so that the computations behind them are fully **reproducible**.

Chapter 1

Welcome to computational biology for all!

Welcome to *Computational Biology for All!*.

This book will introduce you to key concepts of computational biology using the software R. It covers such topics as statistics, data science, bioinformatics, building phylogenetic trees, and building computer models of biological processes.

I will make only two assumptions in this book:

1. You are interested in biology and need a computer to answer a question
2. You've had some college-level biology or are willing to read some basic background information in the book, the appendices or the internet.

That's it. If you've forgotten how many amino acids are specified by the genetic code (or never learned; its 20), have never run computer code, or aren't sure what "computational biology" is no worries. We'll work through everything step by step, review often, and link to additional resources.

1.1 R and RStudio - your new best friends!

R is one of the major computer languages used by scientists. "R" refers both to the computer language itself, and to the base software which runs the computer code for us. Most people write and run their code in a special program that acts like a word processor for coding; these goes by the fancy name **Integrated Development Environments** or **IDEs**. In this book we'll use the popular IDE software called **RStudio**.

You should get access to the software combination of the software **R** and **RStudio(())** either on the cloud via an account with (**RStudio Cloud**)[https://rstudio.cloud/] or on your own hard drive. (Some institutions may have

their own special implementation of R and RStudio; ask your tech support about this.)

If your are brand-new to using R the easiest way to get started is using RStudio Cloud. Getting R and RStudio set up on your own computer isn't (usually) difficult, but RStudio Cloud even easier. For information on R, RStudio, and RStudio Cloud see the Appendices. There are also many videos on the internet walking you through these topics.

If I've already lost you a bit with any of this information don't worry - we'll cover more details in the following chapters, and the Appendices cover how to get start with R, RStudio, and RStudio step-by-step.

1.2 How to use this book - be an active learner!

While you can read this as a regular book, it is meant to be an **active learning text**. That means you'll get much more out of it if you are working through all the code step by step. There are two ways to do this:

1. Read the book like a book (printed, PDF, website) and type the code in RStudio.
2. Download the associated **Active Learning Notebooks** and work through them in RStudio.

The **Active Learning Notebooks** contain **ALL** of the text and code, and is the recommended way to experience the book.

1.3 Biological scope of this book

"Computational Biology" means different things to different people, and I'll discuss how it can be defined in a later chapter. In general, though, I'll apply a very broad definition and touch upon all aspects of biology, from biochemistry to ecology. My starting point will generally by the topics classically associated with computational biology: bioinformatics, genomics, and building phylogenetic trees. Moreover, when I cover other topics like population dynamics or community ecology I'll often use molecular-biology related examples, such as population growth of transposons in our genomes and community diversity of bacteria in our guts (the so-called gut microbiome, which is studied using molecular sequencing technologies). If you're primary interest is in ecology everything in this book will be applicable at the very least in terms of the techniques, and hopefully it will help everyone expand their idea of how ecological concepts can be applied.

Chapter 2

What is Computational Biology?

This is mostly notes.

2.1 Notes

2.1.1 NIH definition

See file NIH_def_bioinfo_2000.rmd for full text of the NIH document

NOTES:

- bioinformatics relevant to all life sciences (not just biology)
- Bioinf/comp bio both "rooted" in life science, comp sci, info sci and technologies
- Bioinformatics: application focused, access / understanding from data
- Comp bio: theory/discovery approaches

  Bioinformatics: Research, development, or application of computational tools and approaches for expanding the use of biological, medical, behavioral or health data, including those to acquire, store, organize, archive, analyze, or visualize such data.

  Computational Biology: The development and application of data-analytical and theoretical methods, mathematical modeling and computational simulation techniques to the study of biological, behavioral, and social systems

2.2 Luscombe et al 2001

N.M. Luscombe, D. Greenbaum, M. Gerstein. Bioinformatics definition committee. 2001. What is Bioinformatics? A Proposed Definition and Overview of the Field https://www.thieme-connect.com/products/ejournals/abstract/10.1055/s-0038-1634431

> "Our definition is as follows: Bioinformatics is conceptualizing biology in terms of macromolecules (in the sense of physical-chemistry) and then applying "informatics" techniques (derived from disciplines such as applied maths, computer science, and statistics) to understand and organize the information associated with these molecules, on a large-scale." Luscombe et al 2000

> "Analyses in bioinformatics predominantly focus on three types of large datasets available in molecular biology: macromolecular structures, genome sequences, and the results of functional genomics experiments (eg expression data). Additional information includes the text of scientific papers and "relationship data" from metabolic pathways, taxonomy trees, and protein-protein interaction networks." Luscombe et al 2000

2.3 Taprich et al 20121

Taprich et al. 2021. An instructional definition and assessment rubric for bioinformatics instruction. https://iubmb.onlinelibrary.wiley.com/doi/full/10.1002/bmb.21361

> "An interdisciplinary field that is concerned with the development and application of algorithms that analyze biological data to investigate the structure and function of biological polymers and their relationships to living systems."

2.4 Smith 2015 Frontiers in Genetics

Smith. 2015. Broadening the definition of a bioinformatician. Frontiers in Genetics https://www.frontiersin.org/articles/10.3389/fgene.2015.00258/full

> "On a given day, I spend much of my research time staring at nucleotide sequences on a computer screen and theorizing about the evolution of genomes; thus, I feel comfortable calling myself a bioinformatician, or at the very least a scientist who primarily uses bioinformatics for his research. If asked, most of my colleagues, mentors, and students would also define me as a bioinformatician. But there is one small catch: I don't know how to program computer software or curate databases, and I am even quite pathetic at writing UNIX

> commands, which according to some precludes me from having the title of bioinformatician." Smith (2015)
>
> "I imagine that many of the scientists reading this essay will consider me an imposter, an amateur who points, clicks, and stumbles his way through the complicated landscape of bioinformatics. … I believe that as a research community we need to broaden our definition of what it means to be a bioinformatician, not restricting it to only those who develop software or design and maintain data resources. … [T]he term bioinformatician should encompass the countless and ever growing number of scientists who use computers and bioinformatics programs to address fundamental questions in biology." Smith (2015)

2.5 Good readings / references

Koonin, EV. 20xx. Primer: Computational genomics. Current Biology - Magazine. R155-158…

2.6 Potential references

https://asistdl.onlinelibrary.wiley.com/doi/full/10.1002/asi.20133?casa_token=iDFXKee8e1gAAAAA%3AD7vtxphVMi2MHDEIvWi7Y3twQcKUvdI5Co3pF11-OgoTdk1dEHe8H5XKv67mnz1uyGyFRoMeqrbeY_Q

https://www.geneticsmr.com/sites/default/files/articles/year2017/vol16-1/pdf/gmr-16-01-gmr.16019645_0.pdf

https://ieeexplore.ieee.org/abstract/document/1678036?casa_token=H9KSAnKN-XIAAAAA:R_uzPueqLDOCWccNZgdRkaRxpOo3J2RSMpVG7ZLAebTVp0-tqGl0TKQderfudEOS2efgl9oWVw https://dl.acm.org/doi/abs/10.1145/1031120.1031122?casa_token=YBJzo0bknd8AAAAA:tzp2T3uGqb1MXRPtScpKXAyKqtmTORa6B_yWGA9dQ9Pj4UElgeH8LK0hVJ_2PaAF51I3oqY8N-EkA

2.7 Original info, from Eco Data Science project

[NOTE: This section is currently under development. The paper by Touchon & McCoy (2016) and its references lay out many of the reasons for the statistical focus of this book and relates to all biology, not just ecology.]

> "Ecological questions and data are becoming increasingly complex and as a result we are seeing the development and proliferation of sophisticated statistical approaches in the ecological literature. … It is

> no longer sufficient to only ask 'whether' or 'which' experimental manipulations significantly deviate from null expectations. Instead, we are moving toward parameter estimation and asking **'how much'** and in **'what direction'** ecological processes are affected by different mechanisms" (Touchon & McCoy 2016, Ecosphere, emphsis mine)

> "Spreadsheets are often used as the basis of data collection and education; but this is potentially problematic since spreadsheets typically do not promote good data management practices…. The features of spreadsheets that make them desirable for the average researcher, such as extensibility, use of formatting for organization, embedding charts, make them undesirable for preparing data for long-term archiving and reuse."(Strasser & Hampton 2012 Ecosphere)

2.8 What is data science?

> Data analysis include "procedures for analyzing data, techniques for interpreting the results…, ways of planning the gathering of data to make its analysis easier, more precise or more accurate, and all the machinery and results of (mathematical) statistics which apply to analyzing data." John Tukey, "The future of data analysis", Annals of Mathematical Statistics, 1962.

- People argue about what data science is
- What Tukey calls "data analysis" is now termed "data science" by many.

- Some define data science as closely allied with computer science and want its use most closely associated with things like "big data", data mining, machine learning, and artificial intelligence.
- Others, such as RStudio's Hadley Whickham (creator of ggplot2, dplyr, and most of the infrasture of the tidyverse of R package) define it more broadly to involve all aspects of the life cycle of data.
- (Wickham also defines a data scientists as "A data scientist is a statistician who is wearing a bow tie" https://twitter.com/hadleywickham/status/906146116412039169?lang=en)

2.9 Refereces

Strasser & Hampton 2012. The fractured lab notebook: undergraduates and ecological data management training in the United States. EcoSphere. https://esajournals.onlinelibrary.wiley.com/doi/abs/10.1890/ES12-00139.1

Touchon & McCoy 2017. The mismatch between current statistical practice and doctoral training in ecology. EcoSphere. https://esajournals.onlinelibrary.wile

y.com/doi/abs/10.1002/ecs2.1394

Tukey. 1962. The future of data analysis. Annals of Mathematical Statistics. https://www.jstor.org/stable/2237638

2.10 Bibliography

Relevant papers cited by Touchon & McOy 2017.

Barraquand, F., T. H. G. Ezard, P. S. Jørgensen, N. Zimmerman, S. Chamberlain, R. Salguero-Gómez, T. J. Curran, and T. Poisot. 2014. Lack of quantitative training among early-career ecologists: a survey of the problem and potential solutions. PeerJ 2:e285.

Butcher, J. A., J. E. Groce, C. M. Lituma, M. C. Cocimano, Y. Sánchez-Johnson, A. J. Campomizzi, T. L. Pope, K. S. Reyna, and A. C. S. Knipps. 2007. Persistent controversy in statistical approaches in wildlife sciences: a perspective of students. Journal of Wildlife Management 71:2142–2144

Ellison, A. M., and B. Dennis. 2009. Paths to statistical fluency for ecologists. Frontiers in Ecology and the Environment 8:362–370.

Germano, J. D. 2000. Ecology, statistics, and the art of misdiagnosis: the need for a paradigm shift. Environmental Reviews 7:167–190.

Quinn, J. F., and A. E. Dunham. 1983. On hypothesis testing in ecology and evolution. American Naturalist 122:602–617.

Chapter 3

How will this book teach computational biology?

> "The rise of computer programming, computational power, and modern statistical approaches may…" allow "…scientists to ask new questions and to extract more information from data than ever before." (Touchon & McCoy 2016, Ecosphere)

3.1 What you'll learn

3.1.1 General skills that you'll learn

- **Statistical computing** using R, RStudio, and rmarkdown
- **Data analysis**, from t-tests to mixed models in R
- **Data visualization**, with an emphasis on ggplot2
- **Data science**, from data management best practices to data cleaning with dplyr
- **Computational reproducibility**, from formatting scripts to using rmarkdown to write reproducible reports

3.1.2 Computational biology skills you'll learn

- Working with sequence data
- alignments
- Phylogenetics
- …

3.2 Teaching approach

- Always explore and visualize data
- Step-by-step instructions
- Frequently refreshing and review
- Comprehensive and self-contained
- Worked example
- "Modify this code" tasks
- Code completion tasks (key steps missing)
- Broken code tasks (fix non-functional code)
- Links to Jupyter notebooks (not yet implemented)
- "Active Learning Notebooks" - all content and code in .Rmd format

3.3 Requirements

- R
- RStudio
- External packages loaded via RStudio

3.4 What's not in this book / course

Chapter 4

What is R and why use it?

[these notes are from a lecture and have not been re-written much yet]

R is a powerful piece of software used for data science and data analysis. In this chapter I will briefly introduce the advantages of using R, why you might want to learn it, and also indicate some alternatives and adjuncts you could consider.

4.1 How do we typically use software in science?

Most scientists rely on both general and specialized pieces of software for various parts of their work. For data entry they likely use spreadsheet software Excel, though increasingly Google Sheets. For data analysis they might use one of many options, such as GraphPad Prism, Minitab, SAS, SPSS, or STATA. For making plots, many people will export their export their results back to Excel, while others use specialized software like SigmaPlot. Many scientists also use specialized programs; in ecology many researchers do GIS in ArcGIS or QGIS, mark-recapture analysis in Program MARK or Distance, use RAMAS or Vortex for population viability analysis, or build custom mathematical programs in MatLab or Python. If they do multivariate statistics like [ordination](https://en.wikipedia.org/wiki/Ordination_(statistics) the may use a specialized stats program like PC-ORD.

Since software can be expensive, some scientists will rely on Excel for all of their work. Excel can do many things, but it can't do everything all the specialized types of software can do. Moreover, its very limited in the range of statistics it can do and graphs it can make.

4.2 What does R do?

R is amazing because it has been explicitly developed to do several things very well, particularly statistics, math, making great-looking figures, and writing computer programs to automate these tasks. Additionally, R has been extended by developers to be able to be a powerful tool for data cleaning and organization, to be used as a GIS, and as an integrated word processor and website make for publishing work.

4.3 Why use R

In addition to is many capabilities, R has the advantage this it is

- free anyone, always
- used by statisticians to develop new statistical techniques, so new techniques often come out 1st in R
- used by almost all ecological statisticians to develop new techniques (mark recapture, distance sampling)

4.4 Who uses it?

R continues to increase in popularity. Among data scientists it is second only to Python. Among academics it has eclipsed SAS in many fields. It is also used by analyses in many large companies, such Facebook, and by journalists looking for stories in or reporting on large volumes of data

see http://blog.revolutionanalytics.com/2014/05/companies-using-r-in-2014.html for further discussion.

4.5 R and computational reproducibility

One factor potentially contributing to R's popularity, or at least a major bonus for using it, is ease of use for making analyses reproducible. All commands in R are typed out and the best way to do this is in a static **script file** from which you send commands to R to execute. This creates a record of your analyses. This feature is shared by other programs such as SAS and Stats, and other programming languages such as Matlab and Python. The advantage of R is that the script files are simply plain text files which anyone can open and - if they've downloaded R, which is free - they can run. Developers have also created numerous tools for creating **reproducible analysis workflows** and which allow R to be used in all data-related aspects of a project, from **data cleaning** to **formatting journal submissions**. What this means is that without become an expert programmer you can set up your work so that you can re-run all of your data cleaning, analyses, and graph building with a

single command in R. This makes what you've done auditable, transparent, and easy to re-use for future work.

4.6 Alternatives to R

R has many advantages, but it has one critical issue: the learning curve. R is a command-line driven analysis tool, which means you type out specific commands for almost everything single thing R does. Excel is pretty user friendly, and several stats programs similarly use point-and-click interfaces, such as SPSS, JMP, and Stata SAS also requires a lot of command writing, but is generally consider more user friendly than R.

Recently, two free point-and-click statistical analysis programs have been release that are built on R but require no programming. JASP ("Just another statistics program") has an emphasis on Bayesian statistics, particularly Bayesian hypothesis testing using Bayes factors (an approach increasing in popularity, especially in psychology, but which some Bayesians, like Andrew Gelman, disavow). While JASP is based on R, it does not currently allow access to the underlying R code.

Jamovi has a similar spirit as JASP (indeed, it was founded by developers who had worked on JASP) but is more transparent about the underlying R code being used to run the analysis.

Chapter 5

The layout of RStudio

To move forward with this book you need to get access to R. There are several ways to do this, and if you are participating in a class your instructor may have a favored way to do it.

5.1 RStudio (Desktop or Cloud)

There are two primary ways to use R relevant to this book:

1. **Create an account at RStudio Cloud.** This is the easiest way to get started; you get 15 hours per month free which is enough for you to get your bearings. Complete instructions for doing this are available in the appendices. Note that RStudio Cloud and RStudio are essentially identical and so all instructions related to RStudio are relevant to RStudio cloud.

2. **Download R AND RStudio desktop.** Key here is that you need TWO pieces of software, R and RStudio. Complete instructions are in the appendices.

These two methods are essentially equivalent. To jump in and get started I recommend RStudio cloud. See the appendices for step by step instructions.

5.2 R emulators

There are also two other ways of using R which we'll occasionally use. I mention them here briefly. They essentially emulate the R experience but eliminate the hairy details.

1. **Shiny Apps:** These are independent mini-programs that allow you to explore or carryout some aspect of R functionality without doing any coding. They typically live on the internet, but can also be brought up

from RStudio desktop. An example of a simple Shiny App on the internet is here: https://shiny.rstudio.com/gallery/faithful.html.

2. **learnr tutorials:** These are interactive tutorials where you do basic coding in your web browser. Here's an example: https://learnr-examples.shinyapps.io/ex-data-filter/

5.3 Other ways

Two other ways you may encounter R

1. **Use a Jupyter notebook:** Jupyter notebooks are an advanced R emulator. They are really cool, allowing you to essentially combine text (like a tutorial) with code that can be run within a web browser.

2. **Use regular old R:** When you download R an icon will appear on your desktop. You can access old-school R that way, but almost no one ever does. For a bit more information about this see the appendix.

5.4 RStudio at a glance

Now we'll get started with RStudio. We'll get to know what it looks like and configure it a bit for out needs.

If you are using RStudio desktop the RStudio logo looks like this:

Figure 5.1: The RStudio logo
(#fig:rstudio.logo)

Whether on the Cloud or Desktop, when you open up you'll be greeted by a fairly busy array of menus and things. Don't panic! A typical fresh starting point in RStudio/RStudio Cloud is shown in Figure 2.

When referring to RStudio (and equivalently RSTudio Cloud - this is the last time I'll mention this so hopefully get get the point), there are two terms that need to be understood. As shown in Figure 3, there is 1)the **console** section of RStudio and 2) the **script editor** or **source viewer**.

(A "cheat sheet" called the "RStudio IDE Cheat Sheet" details all of RStudio's many features and is available at https://www.rstudio.com/resources/cheatsheets/ . It very thorough, though a bit dense. I don't recommend it for beginners but you should remember that it exists.)

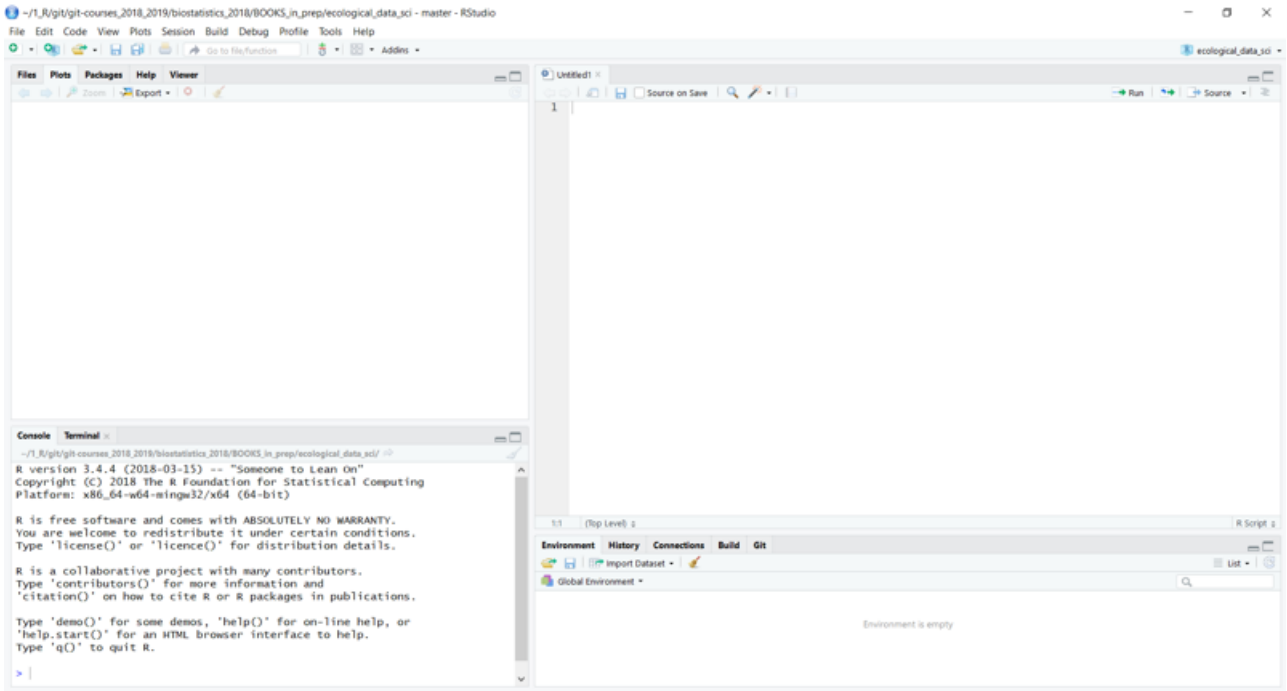


Figure 5.2: RStudio when first opened.
(#fig:rstudio.open)

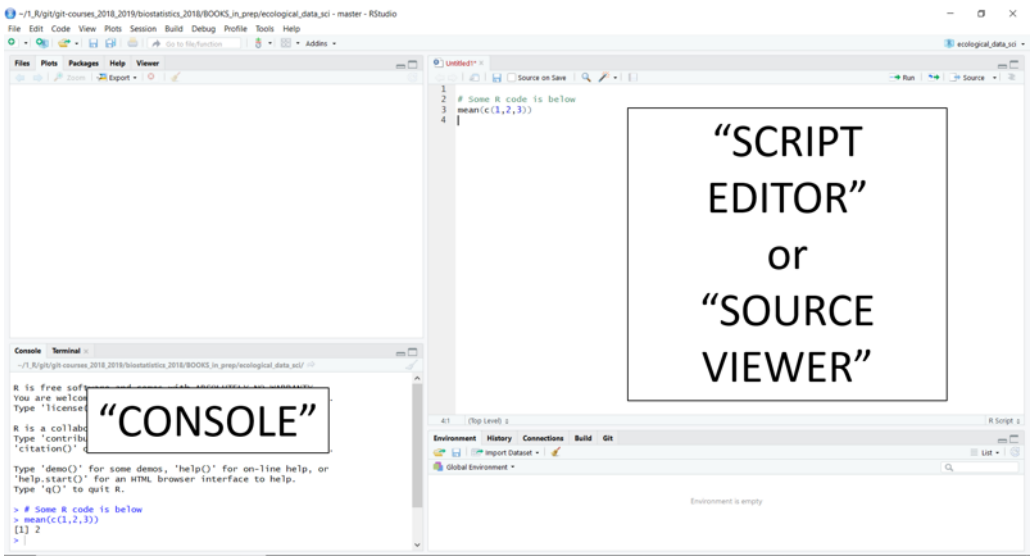


Figure 5.3: RStudio's console and script editor.
(#fig:rstudio.console)

5.4.1 The console versus the script editor

You can type and enter text into both the **console** and the **script editor** (also called the **source viewer**. The console, however, responds actively like a calculator, while the script editor works more like a text editor. Information can be passes unidirectionally from the script editor to the console, but not the other way.

5.4.1.1 The R console

The **console** in RStudio gives you **interactive programming** environment that is very similar to a scientific calculator. If you click your mouse inside the console and type `1 + 1` then press enter you will see the following type of output

```
1 + 1
```

```
## [1] 2
```

Note that right in front of where you typed `1+1` there is a `>` symbol. This is always in the R console and never needs to be typed. You may occasionally see it printed in books or on websites, but it doesn't ever need to be typed.

One thing to note about R is that it's not particular about spacing. All of the following things will yield the same results

```
1+1
1 + 1
1          +         1
1 +                                    1
```

Got it? Awesome! Now we're ready for some real data analysis.

(BOOK) Getting started with R

NEW text

In Book 2 we'll get started using R. We'll focus on jumping in with some simple examples that tell interesting scientific stories so we can get a sense for the power of R. Later, we'll dig deeper - bit by bit - not the details. Like any language R is a versatile tool for communication, but has conventions, quirks, idioms and dialects that new users have to become comfortable with. In general my goal in Book 2 will be to help you get a general sense of R, let you see a bit of the wild – and overwhelming – world of possibilities for how R is written, and outline the more specific aspects I'll use to help facilitate learning.

We'll have three specific goals:

1. Run some initial, simple commands in R to see how it works.
2. Take a broad tour of the wide world of R so see the many faces of R code you may encounter in the wild.
3. Highlight the particular way coding conventions and idioms of R I'll use.
4. Introduce briefly some of the different ways you can get data into R.

If you find yourself getting confused or overwhelmed don't panic A goal of this section is to prevent *future* confusion by giving you a sense for the many different ways R can be written and that you will eventually encounter on the internet or other books, and to contrast them with what you'll use in this book. Here, I've worked to keep things consistent and to either use the simplest methods to accomplish the goal or to carefully break down hard tasks or concepts so you can master them. Once you start typing into your search engine "R code …" you will get MANY different types of code which you may not yet be prepared to work with.

Chapter 6

Hello R! A first encounter with data visualization

6.1 Key Ideas

- **Commands**: R uses simple typed commands to do *everything.*
- **Data**: loading data that comes with R using the `data()` command.
- **Plotting**: Making simple plots with the `plot()` command.

6.2 Data in R

We'll start our exploration of R with a classic dataset from ecology and statistics showing one of the most striking patterns in wildlife biology: (**population cycles**)[https://en.wikipedia.org/wiki/Population_cycle]. Populations are always changing, whether it is declines in the number bacteria in our gut after we take an antibiotic or increases in raptor species after highly toxic pesticides were banned in the mid-twentieth century.

In stark contrast, a few unique animals populations show dramatic **oscillations**, with rapid increases soon followed by dramatic drops, as if the population were on a roller coaster. One such population are Canada lynx (*Lynx canadensis*) in Canada.

Lynx are (unfortunately) prized for their soft fur. In years when there are many lynx, trappers kill many lynx; in years when there are few lynx, trappers kill few (however, the number of lynx killed by humans does not contribute to the cycle). Records from fur industry have been compiled by ecologists for over a century to investigate what drives changes in lynx numbers. The table below shows a snapshot of these data, starting from the earliest available records in 1821.

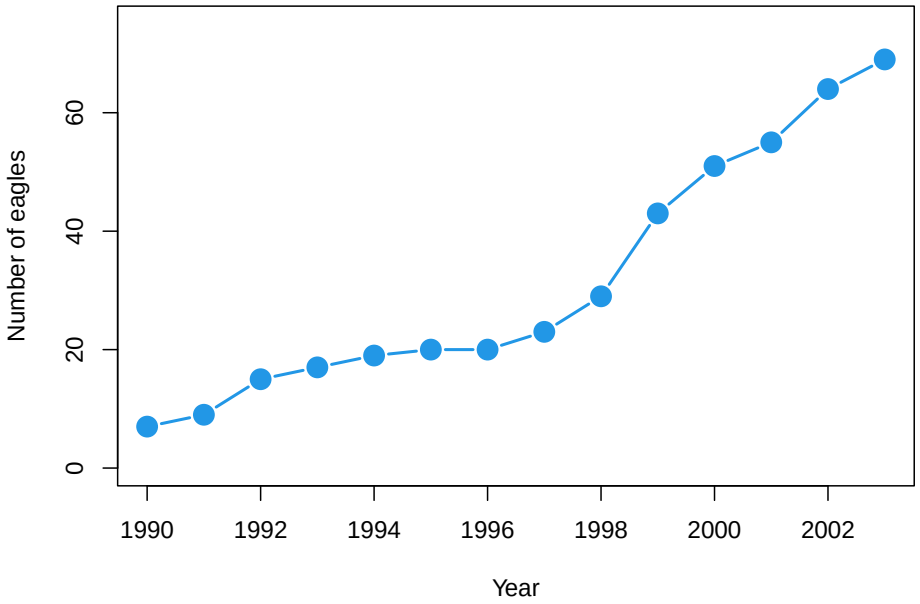


Figure 6.1: Growth of Eagle population in Pennsylvania, USA

Table 6.1: Number of lynx trapped in Canada

year	lnyx.population
1821	269
1822	321
1823	585
1824	871
1825	1475
1826	2821
1827	3928
1828	5943
1829	4950
1830	2577
1831	523
1832	98
1833	184
1834	279
1835	409

You've probably plotted data like this by hand using graph paper, point by point by locating the x-y coordinates. When plotting data like this, time (in this case "year") goes on the horizontal **x-axis**, and the changing variable (lynx.population) goes on the vertical **y-axis**. In a **spreadsheet**, you could highlight these columns and click on the "Make graph" icons to make the initial plot, then adjust things by clicking on parts of the plot you want to change.

Spreadsheets are said to operate under the principle of **"What You See Is What You Get"**, or **WYSIYGY**. They use a fully mouse-drive **Graphical User Interface (GUI)** where everything is done by pointing and clicking. Every time you make a plot you do these steps.

R is *very* different - you only see things when you want to see them and you do everything via typed commands. This is a large paradigm shift for most people, so we'll start very very slow.

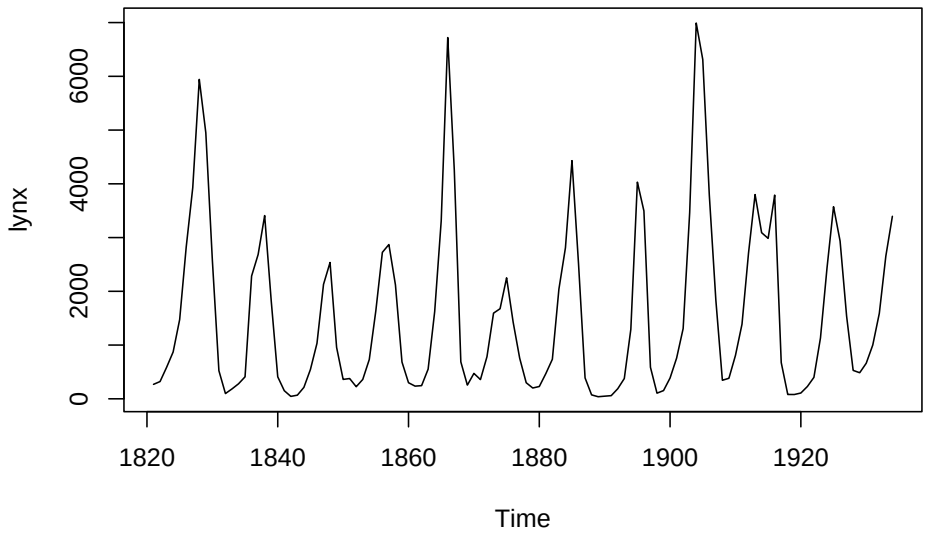
You might be wondering "*Ok, now what?*" because nothing apparently happened. What you've done, though is loaded the `lynx` data into R's active memory, where it will wait for you next command.

Loading your own data into programs such as R can be a pain, so I'll use techniques such as use the `data()` command to make things as smooth as possible. I'll also introduce other methods, such as loading small datasets using code and downloading them from the internet. I'll also provide a thorough overview in the Appendices.

6.4 Plotting simple datasets with `plot()`

Now in the console type `plot(lynx)` and press enter. You should see readout like what you see below …

```
plot(lynx)
```



… and this intriguing plot. The x-axis is time from the early 1800s to the early 1900s, and the y-axis is the number of lynx pelts. This pattern continues to today, and it causes have kept ecologists working hard for over 100 years. Moreover, statistical analyses of data such as these have similarly kept statisticians busy.

This is a simplified example, but that's the basics of working in R:

1. Load data
2. Use commands like `plot` tell R to do something with.

For most of this book we'll use data that's been mostly pre-packaged for you to work with and loaded using the `data()` command. Real data analyses require more steps, and later in the book we'll briefly cover them so you are familiar

with them when you see them elsewhere; further details can also be found in the Appendix.

6.5 Commands in R

The words "data" and "plot" in R represent **commands**; R associates specific code and therefore actions with these words To indicate commands in the text I'll always write it like this: `data()`. The parentheses are very important in R; forget one of them, and things won't work. After the word representing the command there is always a **parenthesis** `(`. Other things such as the name of a dataset go after the first parenthesis, and the command is completed with a matching parenthesis `)`. To emphasis that things using go within the parentheses I will often write commands like this `data(...)`, where the `...` in this case is the name of a dataset.

In some cases you can issue a command, like `data()`, and R does something only behind the scenes. Often, though, we'll elicit a reaction from R, either data will appear in the Console or a plot will be created.

6.6 The structure of commands in R

Our use of the `plot()` command was pretty standard; there were two pieces to it:

1. The command, `plot()`
2. The data, `lynx`

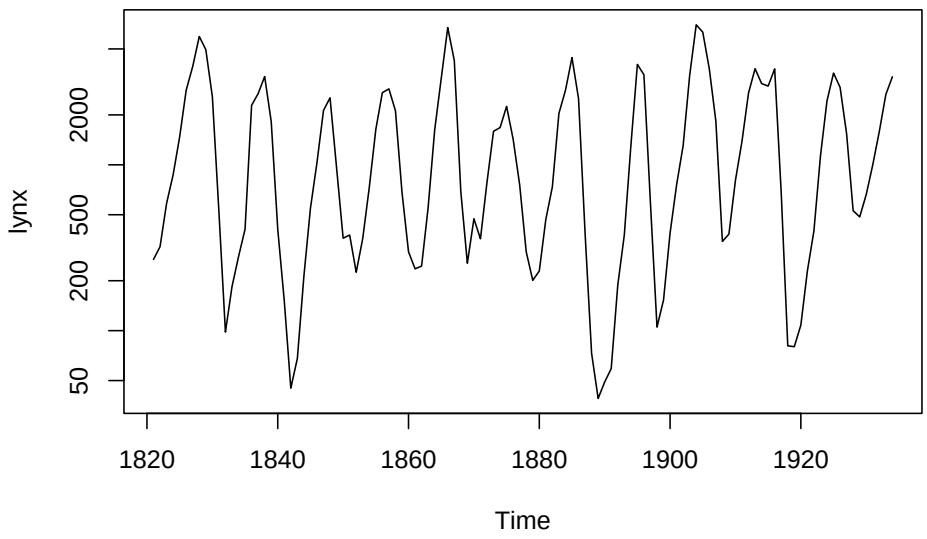
Data in R – and especially in computational biology – can take on many forms, which we'll cover as needed as throughout the book. All data is presented in R by an **object** stored behind the scenes in R's memory. The fact that data in R is usually resting out of view until we do something explicitly with it can take some getting used to, since usually we work with data printed out on a page or displayed in a spreadsheet.

Commands in R almost always include an object within them. Next we'll consider something else that go with commands : **arguments**.

6.7 Arguements in R

A common mathematical operation when doing data analysis is taking the **log** of something. (For now we won't worry about what the log is or why we use it' we'll come back to this little bit of math frequently though). We can tell R to plot the log of our lynx data by adding the argument `log = "y"` to the `plot(...)` command. This alters the graph a bit which, for some particular data analysis purposes, will come in handy (more on that later).

```
plot(lynx, log = "y")
```



In the code above `log` is the argument and `"y"` is the **value** assigned to the argument.

Arguments **always** have an equals sign with them, so I'll emphasize this by typically writing them as `argument.name = ...`. One tricky thing about arguments is that they can take on letters, words, or numbers, and sometimes there need to be quotation marks like `log = "y"`, but not always.

Since arguments are fairly tricky, they are a common source of errors, such as forgetting `=`, or putting the value of the argument in the wrong format.

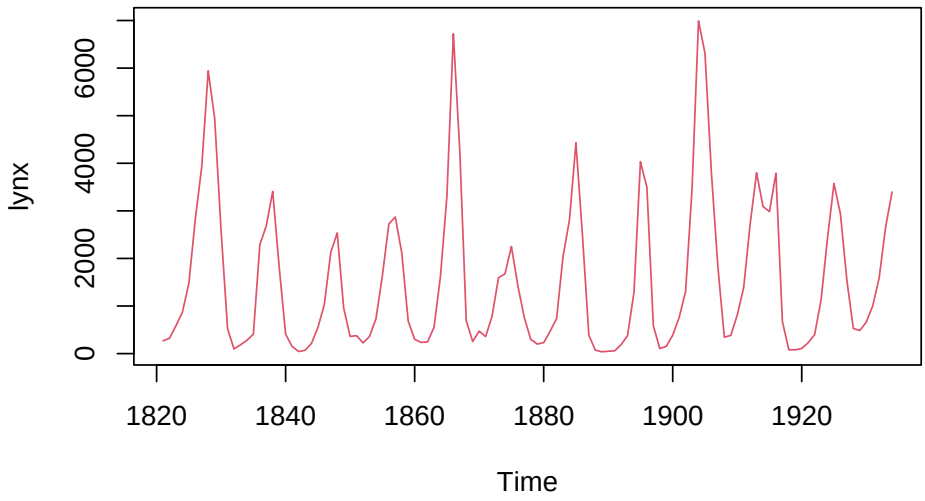
6.8 Arguements and more arguments

We now have three things going on

1. A **command**, `plot()`
2. A **data object**, `lynx`
3. An **argument**, `log = "y"`

Most functions in R have multiple arguments that can be invoked Try the following code `plot(lynx, col = 3)`. That is the `plot()` function with the argument `col = 3` added. What do you think `col = 2` means? Try different values like 4, 5, and 6.

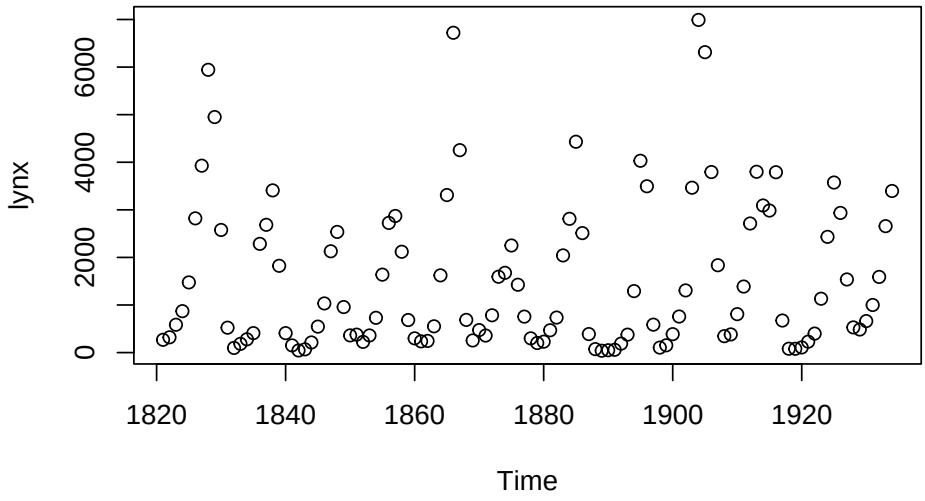
```
plot(lynx, col = 2)
```



Now try this: `plot(lynx, type = "p")`

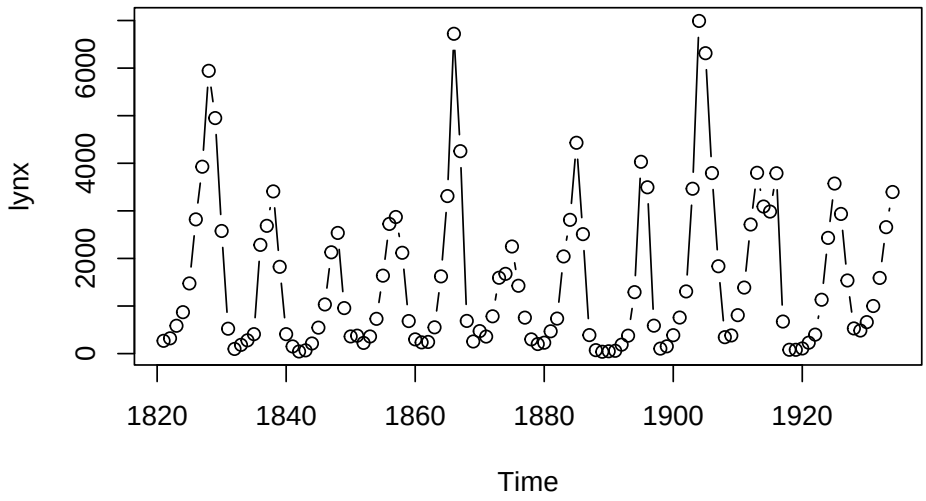
Note that there are quotation marks around the p.

```
plot(lynx, type = "p")
```



Now instead of "p" use "b", which stands for "both". What do you think the "both" is referring to?

```
plot(lynx, type = "b")
```

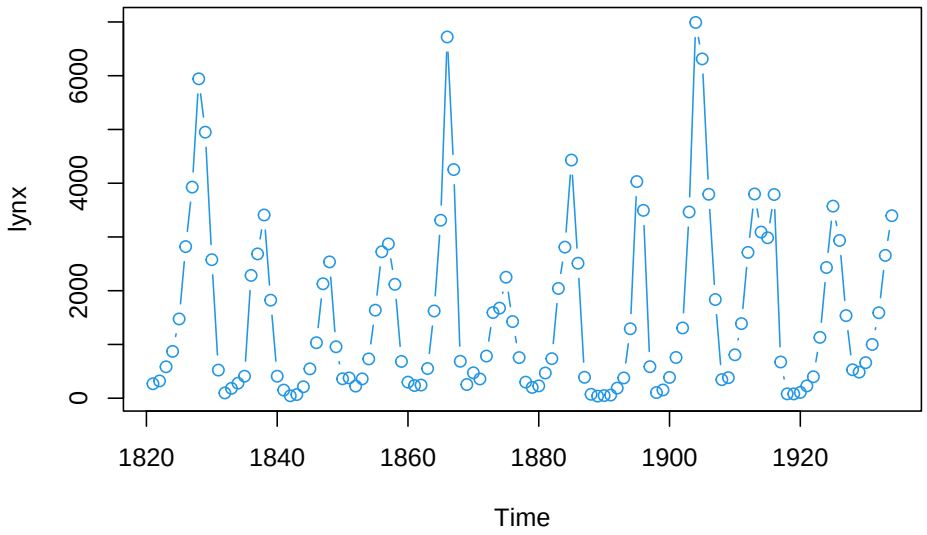



```
## TODO: add followup exercises
### R, biology
## TODO Add as optional? cover elsewhere

plot(log(lynx))
```

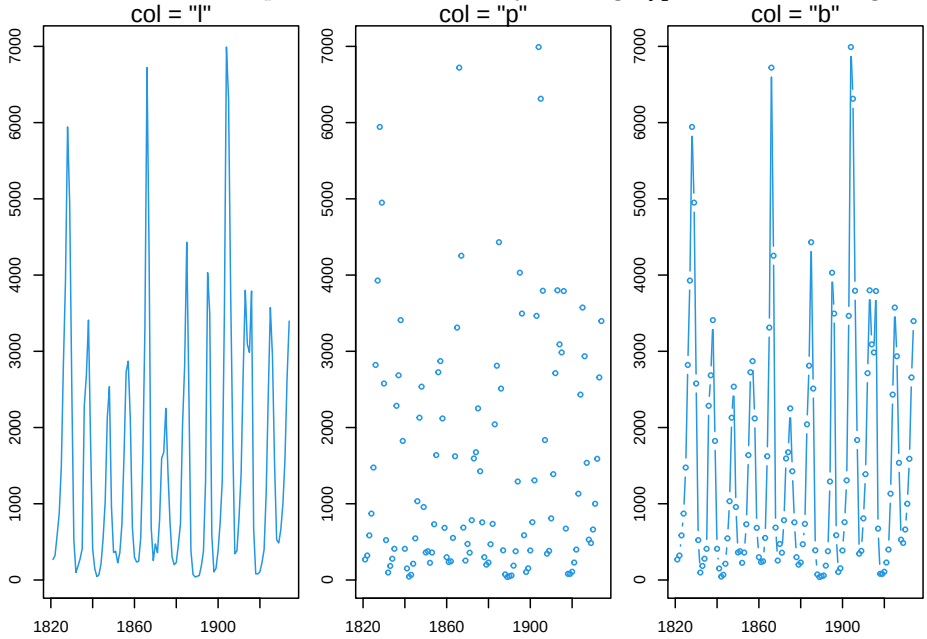
6.9 Multiple arguments at the same time

Functions can not only have multiple arguments, but they can **take on** multiple arguments at the same time. Let's feed two arguments to `plot()`, `col = ...` and a new one, `type = ...`.

```
plot(lynx, col = 4, type = "b")
```



Examine the series of plots below; what do you thing `type=...` is referring to?



6.10 R commands and line breaks

A cool thing about R is that it doesn't care about **line breaks** within a command, so I can do this if I want:

```
plot(lynx,
     col = 4,
     type = "b")
```

Or if for some reason this

```
plot(lynx,  col = 4,
     type = "b")
```

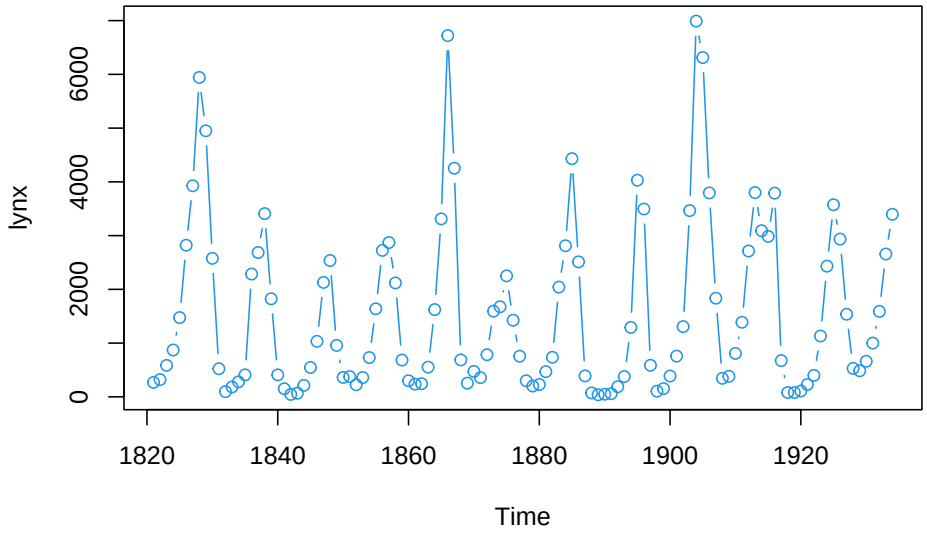
Or even this

```
plot(lynx,

     col = 4,


     type = "b")
```

6.11 Code comments

One thing that putting thing on multiple lines allows you do to is add **comments** to your code if you place a hashtag (aka pound symbol) in front of it.

```
plot(lynx,           # data object
     col = 4,        # color argument
     type = "b")  # type of graph argument
```



6.12 A note on plotting

One of the great things about R is that it can make really nice plots. You'll soon see that there are many ways to do the same basic thing in R, and this includes making plot. **Data visualization** is a key aspect of modern science, so its important to build up your skills, including knowing about the different ways plots are made in R. Don't worry though - we'll build all of this up step by step.

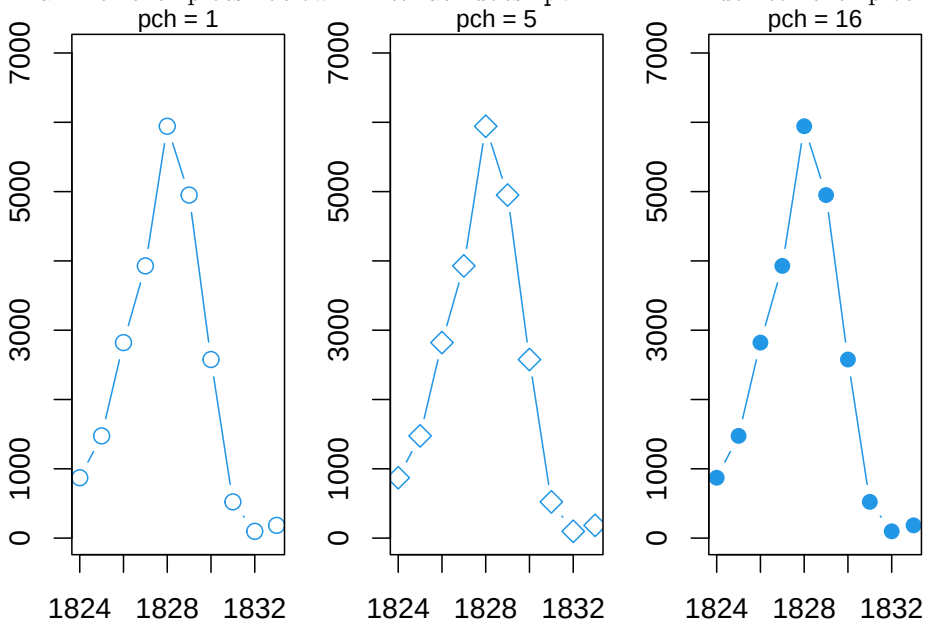
6.13 Now you try it

6.13.1 Easy tasks

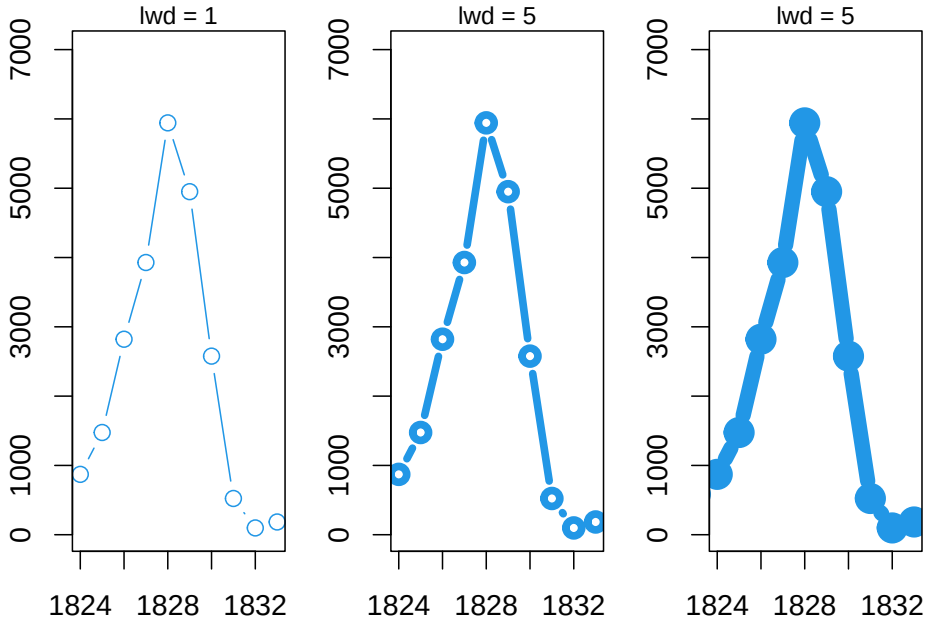
Fix each line of code below so the work

```
plot lynx
plot(lynx
```

Examine the plots below. What does `pch = ...` do to the plot?



Examine the plots below. What does `lwd = ...` do to the plot?



Fix each argument below so the code works.

```
plot(lynx, col  1)
plot(lynx, col = "1")
plot(lynx, type b)
plot(lynx, type = b)
plot(lynx, type "b")
plot(lynx type "b")
plot(lynx, type =  "b)
```

```
plot(lynx, col  1, type = "b)
plot(lynx, col = "1", type = b)
plot(lynx, pch = 1, type b)
plot(lynx, pch 1 type = b)
plot(lynx, lwd = "2" type "b")
plot(lynx pch = 2, type "b")
plot(lynx, lwd = 3 type =  "b)
```

Split the following code between multiple lines and add a comment between each of them to indicate what the argument does.

```
plot(lynx, type = "b", pch =  2,)
```

6.13.2 Intermediate tasks

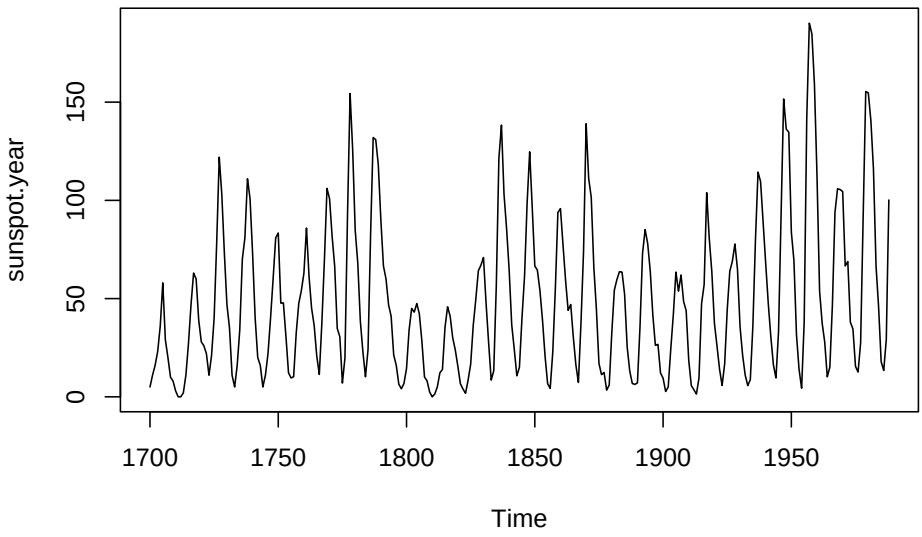
Here's a challenge: there is another dataset that comes with R called `sunspot.year` There was once a hypothesized link between the Canada lynx

and sunspots that we'll explore later; right now we'll just check it out. See if you can do the following things on your own in the R console.

Make a simple plot

1. Using the `data()` command, load `sunspot.year` data into R's active memory
2. Plot the data.

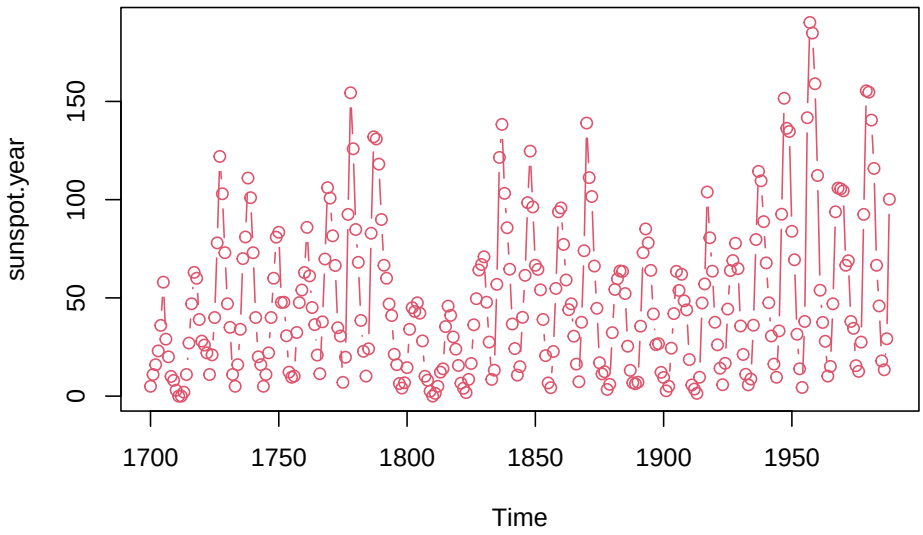
Your plot should look like this:



Modify your plot

1. Use the `col =` argument to make the color of the line different than black.
2. Use the `type =` argument to make the plot show points connected by a line.

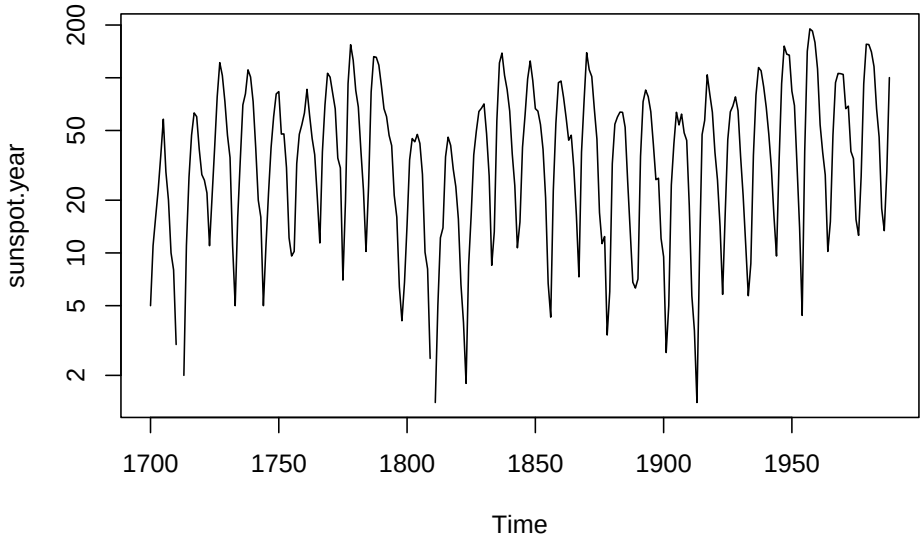
Your plot should look like this:



Plot on the log scale

Plot the sunspot data on the log scale. Note: you will get a warnig in red text - you can ignore this.

```
## Warning in xy.coords(x, NULL, log = log, setLab = FALSE): 3 y values <= 0
## omitted from logarithmic plot
```



Chapter 7

Downloading R packages (and their data)

In the previous chapter we used some data the comes with R. In this lesson we'll start expanding out from the **Base R distribution** and exploring the ways that external **packages** extend R.

7.1 Loading data from R packages

NEW text: When you install R you get *Base R**, which is the core set of functions, functionality, and some data sets. Base R is surrounded by a universe of extensions built by statistician, programmers, academics and businesses that use R for analyses. A lot of R's functionality is found in these packages, including data sets, special plotting functions, and statistical tools for the analysis of complex data. These have to be downloaded from the internet and installed. Most packages contain data in order to demonstrate what they do.

Most R packages you'll use are stored on the CRAN website where you download R (https://cran.r-project.org/). R and RStudio have functions and tools for downloading and managing packages that we'll briefly introduce in this exercise.

Another major platform for packages is called **Bioconductor**; we'll cover downloading packages from there later. Finally, many packages are hosted on a site called **GitHub**. This book relies heavily on an R package I've written and hosted on GitHub called `combio4all` (https://brouwern.github.io/compbio4all/) that contains the datasets used throughout the book, as well as some helpful R functions I've written. Many packages on CRAN also occur on GitHub, especially if programmers are actively developing, updating, and managing the package. We'll cover downloading packages from GitHub later.

7.1.1 Functions & Arguements

- `install.packages()`
  - Argument: `dependencies = TRUE`
- `library()`

7.2 OPTIONAL: What functions come with base R?

The following section is OPTIONAL

If for some reason you want to see *all* the functions that come with the distribution base R, type this into the console and press enter. (`ls` stands for "list" and is a function we'll use more later).

```
# this code chunk is OPTIONAL
ls("package:base")
```

As R has been developed there has also built up a cannon of tried and true packages that are downloaded automatically when you download R. If you want to see all of the packages that come with base R, do this. `library()` is a function you will use a lot.

```
# this code chunk is OPTIONAL
.libPaths("")
library()
```

One package that is part of this cannon is `MASS`, which stands for Modern Applied Statistics in S. "S" is the precursor to R, and MASS is the package that accompanies the book of the same name, which is one of the original books on S/R. (https://www.springer.com/us/book/9780387954578)

End OPTIONAL section

7.3 Load data from an external R package

Many packages have to be explicitly downloaded and installed in order to use their functions and datasets. Note that this is a **two-step process**:

1. Download package from internet with `install.packages(...)`
2. Explicitly tell R to load it with `library(...)`

7.3.1 Step 1: Downloading packages `install.packages(...)`

There are a number of ways to install packages. One of the easiest is to use the function `install.packages()`. Note that it might be better to call this "download.packages" since after you download it, you still have to load it! (Lots of confusion has resulted from calling this function "install.packages").

We'll download a package with lots of interesting biology data called `Stat2Data`. Note that in the **call** to `install.packages(...)`, the name of the package is in quotes.

```
install.packages("Stat2Data")
```

Often when you download a package you'll see a fair bit of red text. Usually there's nothing of interest here, but sometimes you need to read over it for hints about why something didn't work.

For example, when I downloaded this package I got this cryptic message in bright red text: `trying URL 'https://cran.rstudio.com/bin/macosx/contrib/4.0/Stat2Data_2.0.0.tgz'
Content type 'application/x-gzip' length 1177728 bytes (1.1 MB)
== downloaded 1.1 MB`

Followed by this in less insistent black: `The downloaded binary packages are in /var/folders/q8/gwjfr69n05vf4h15l6hdl4d8x1zk5v/T//RtmpeRHx2y/downloaded_packages`

This is all perfectly normal, so don't worry.

7.3.1.1 Protip: don't re-download packages all the time

You can think of R and RStudio like a computer operating systems, and packages like software you chose to download. R, RStudio and packages will need to be updated at times - indeed, the first step in diagnosing many problems with R is to update things. But updates probably only need to be done every six months or so at the most; I generally wait until things stop working.

Anytime a lesson introduces a new package I'll include code to do the download. You only need to do this once unless you start encountering a problems. A way to make this happen is to **comment out** the code by putting a hashtag in front of it like this

```
# install.packages("Stat2Data")
```

This preserves the code but tells R *"don't run this line."*

7.3.2 Step 2: Explicitly loading a package

The `install.packages()` function just downloads the package software to R; now you need to tell R explicitly *"I want to work with the package"*. This is

done using the `library()` function. (Its called library because another name for packages is **libraries**.).

```
library(Stat2Data)
```

As frequently is the case, R doesn't look like its doing much, but you've actually just installed a bunch of cool datasets.

In contrast to running `install.packages()`, `library()` need to be run **every time you use the code**. More specifically, every time you re-start R. What happens is when you shut down R, all the packages you loaded using `library()` are taken out of memory. Next time you use R you need to re-load them using `libary()`.

7.4 OPTIONAL: Seeing all of your installed packages

The following section is OPTIONAL

If for some reason you want to see everything you've downloaded, do this.

```
# This code is optional
installed.packages()
```

End OPTIONAL section

7.5 Downloading packages using RStudio

RStudio has a point-and-click interface to download packages. In the pane of the RStudio GUI that says "Files, Plots, Packages, Help, Viewer" click on "Packages". Below "Packages" it will then say "Install, Update, ..," Click on "Install." (There might be a lag during this process as RStudio gets info about your packages). In the pop up widow there will be a middle field "Packages" where you can type the name of the package you need. There's an auto-complete feature to help you in case you forget the name. Then click "install." Note that in the bottom right corner of the pop up is a checked box next to "Install dependencies." Leave that checked; more on that later.

This is a useful function, but I don't do this very often because I include any download information in my scripts.

7.6 Your turn

Fix this code

```
install.packages(MASS)
install.packages("car)
install.packages ggplot2
install.packages(cowplot
install.packagesggpubr
```

The package `carData` contains a time series dataset called USpop. Fix the following code to download and plot these data.

```
install.packages("carData")
plot(USPop)
```

The package `boot` contains a dataset called `ducks` on the behavior of crosses between two species of ducks: Mallards and Pintails. Fix the following code to download and plot these data.

```
install.packages("boot")
library(boot)
plot(ducks)
```

Chapter 8

Data in dataframes

An interesting dataset in `Stat2Data` is `SeaIce`. Load it with the `data()` command.

```
library(Stat2Data)
data(SeaIce)
```

`SeaIce` shows 37 years of the area of frozen ice in the arctic, from 1979 to 1993. The `lynx` data we worked with previously was in a special format that you'll probably rarely encounter ever again. It was nice to use, however, because it required very little code to plot.
SeaIce, however, is a typical R data object in the form of a **dataframe**. Dataframes are fundamental units of analysis in R. Most of the data you will load into R and work within R will be in a dataframe. The have the same basic structure as a spreadsheet, but R keeps them hidden in memory and you have to use commands to explore them.

8.1 Looking at dataframes with `View()`

To get a spreadsheet-like view of a dataframe you can use the `View` command

```
View(SeaIce)
```

This will bring up the data in a spreadsheet like viewer as a new tab in the script editor, similar to this.

Year	Extent	Area	t
1979	7.22	4.54	1
1980	7.86	4.83	2
1981	7.25	4.38	3
1982	7.45	4.38	4

Year	Extent	Area	t
1983	7.54	4.64	5
1984	7.11	4.04	6
1985	6.93	4.18	7
1986	7.55	4.67	8
1987	7.51	5.61	9
1988	7.53	5.32	10

Note: Unlike a spreadsheet you cannot edit the data when is called up using `View()`

Like a spreadsheet the data are organized in columns and rows. Each **column** represents a type of information:

- `Year`: when data were collected
- `Extent`: the amount of area within the ice-bound region
- `Area`: total area of ice, minus any non-ice area (land, melted water)
- `t`: time point, from 1 (1979) to 15 (1993)

You can think of `Extent` as similar to the size of a country, and `Area` as the actual amount of land in a country minus any lakes.

Each row represents a different year of data; row 1 is the `Extent` and `Area` for 1979, row to is the `Extent` and `Area` for 1980 and so on.

8.2 Looking at dataframes in the console

Another common way to examine data is simply type the name of the data in the console and press enter. This prints it out; however, if its a large data frame this may take up a LOT of room. (I'll just show an exert here).

```
SeaIce
```

```
##    Year Extent Area  t
## 1  1979   7.22 4.54  1
## 2  1980   7.86 4.83  2
## 3  1981   7.25 4.38  3
## 4  1982   7.45 4.38  4
## 5  1983   7.54 4.64  5
## 6  1984   7.11 4.04  6
## 7  1985   6.93 4.18  7
## 8  1986   7.55 4.67  8
## 9  1987   7.51 5.61  9
## 10 1988   7.53 5.32 10
## 11 1989   7.08 4.83 11
## 12 1990   6.27 4.51 12
```

```
## 13 1991   6.59 4.47 13
## 14 1992   7.59 5.38 14
## 15 1993   6.54 4.53 15
```

8.3 Examining part of a dataframe

Look at the top of the dataframe

```
head(SeaIce)
```

```
##   Year Extent Area t
## 1 1979   7.22 4.54 1
## 2 1980   7.86 4.83 2
## 3 1981   7.25 4.38 3
## 4 1982   7.45 4.38 4
## 5 1983   7.54 4.64 5
## 6 1984   7.11 4.04 6
```

Look at the bottom

```
tail(SeaIce)
```

```
##    Year Extent Area  t
## 32 2010   4.93 3.29 32
## 33 2011   4.63 3.18 33
## 34 2012   3.63 2.37 34
## 35 2013   5.35 3.75 35
## 36 2014   5.29 3.70 36
## 37 2015   4.68 3.37 37
```

8.4 Get information about dataframes

The following commands give you important information about a dataframe.

```
is(SeaIce)
dim(SeaIce)
names(SeaIce)
summary(SeaIce)
summary(SeaIce$Extent)
mean(SeaIce$Extent)
min(SeaIce$Extent)
max(SeaIce$Extent)
```

8.5 Accessing rows and columns of dataframes

First row

```
SeaIce[1, ]
```

```
##   Year Extent Area t
## 1 1979   7.22 4.54 1
```

First column

```
SeaIce[ , 1] # with brackets
```

```
##  [1] 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993
## [16] 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008
## [31] 2009 2010 2011 2012 2013 2014 2015
```

```
SeaIce$Year  # with $
```

```
##  [1] 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993
## [16] 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008
## [31] 2009 2010 2011 2012 2013 2014 2015
```

Chapter 9

Plotting data in dataframes

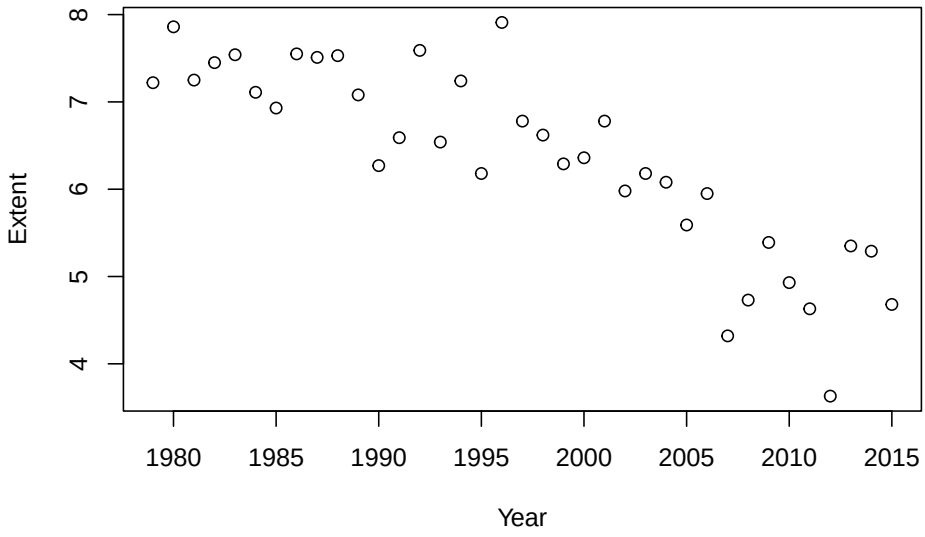
```
library(Stat2Data)
data(SeaIce)
```

Most data in R are organized into dataframes. Similar to when we plot data in a spreadsheet, to plot data from a dataframe we need to tell R exactly what we want on the **x-axis (horizontal)** and **y-axis (vertical)**.

Note: For reasons we don't have to get in to, the `lynx` data were a special case where we didn't have to define x and y

We can plot the Extent of artic sea ice again using the `plot()` command, and using a cool convention in R called **formula notation**. Formula notation uses the (**tilde**)[https://en.wikipedia.org/wiki/Tilde] symbol `~`. In math, `~` can have several meanings. In R, it means "relates to" , "versus", "depends on." So we can plot the relation between `Year` and seac `Extent` as a **`y` versus `x`** relation as `Extent ~ Year`. We also have to include the argument `data = SeaIce` so R knows where to get `Extent` and `Year`.

```
plot(Extent ~ Year, data = SeaIce)  # Note, both words capitalized.
```



9.1 Base-R graphics

When we use the `plot` command were using **Base R** graphics. As noted before there are several ways to make plots in R and you should be able to spot which one is which when looking at code. We'll cover some of the key features of Bare R graphics now. While different plotting methods have different commands and arguments, they all share a common feature: everything in a plot can be customized, and each element is customized with a command or arguement.

9.1.1 Type of points

`plot()` can draw dots or lines We make it use lines using the `type = "l"` argument (note that the l is in quotes)

```
plot(Extent ~ Year, type = "l", data = SeaIce)  # Note: l in quotes
```

As noted before, R doesn't mind if you split things on lines. To keep track of the things I"m doing to the plot I'll format things like this

```
plot(Extent ~ Year,  # relationship
     type = "l",     # type of plot; Note: l in quotes
     data = SeaIce)  # data
```

As we did with the `lynx` data we can combine points and lines with `type = "b"`. (Do you recall what "b" stands for?)

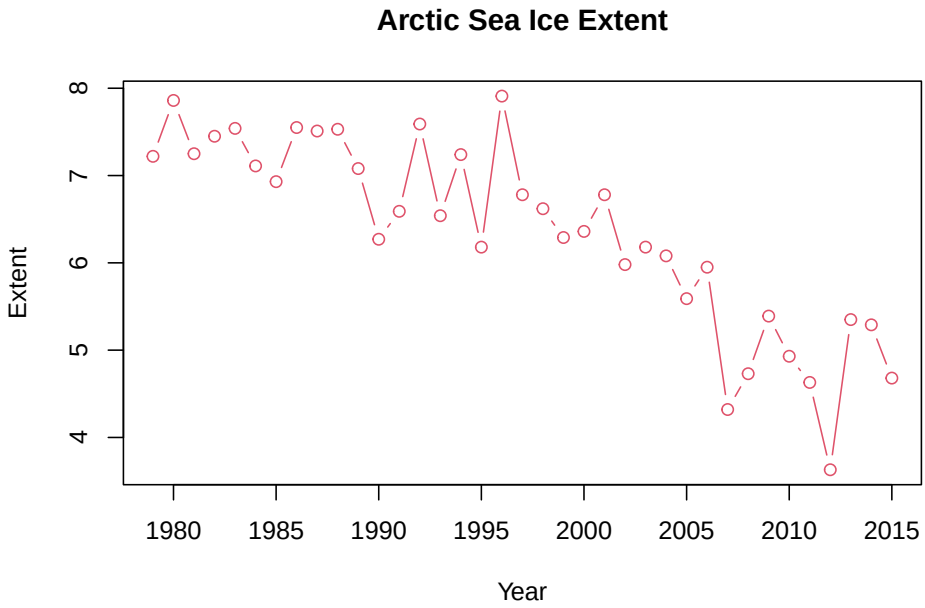
```
plot(Extent ~ Year,  # relationship
     type = "b",     # type of plot; Note: b in quotes
     data = SeaIce)  # data
```

We can adjust color with `col = ...`. Recall the this is just a number, not in quotes.

```
plot(Extent ~ Year,  # relationship
     type = "b",     # type of plot; Note: b in quotes
     col = 2,        # color; no quotes
     data = SeaIce)  # data
```

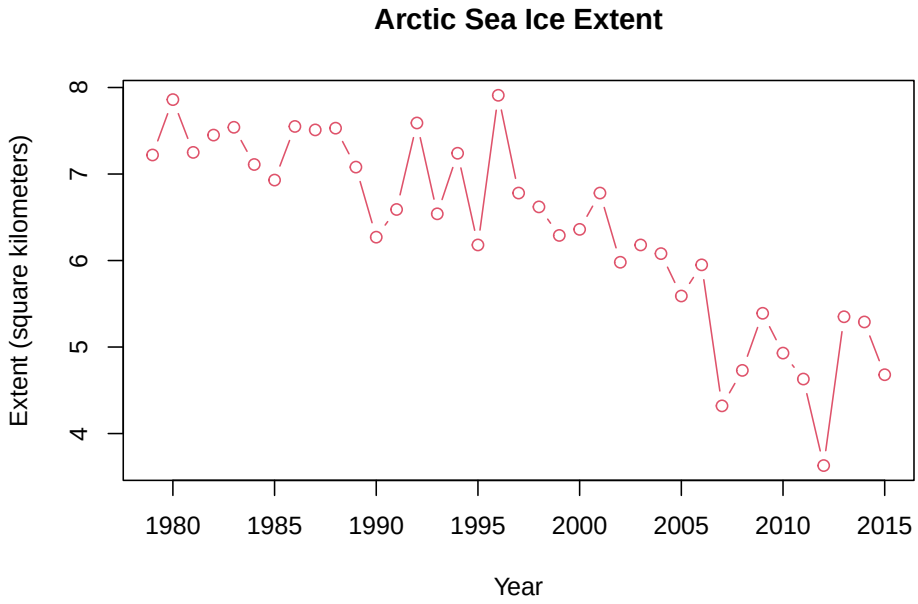
We can add a main title to with `main = ...`

```
plot(Extent ~ Year,  # relationship
     type = "b",     # type of plot; Note: b in quotes
     col = 2,        # color; no quotes
     main = "Arctic Sea Ice Extent", # main title, in quotes
     data = SeaIce)  # data
```



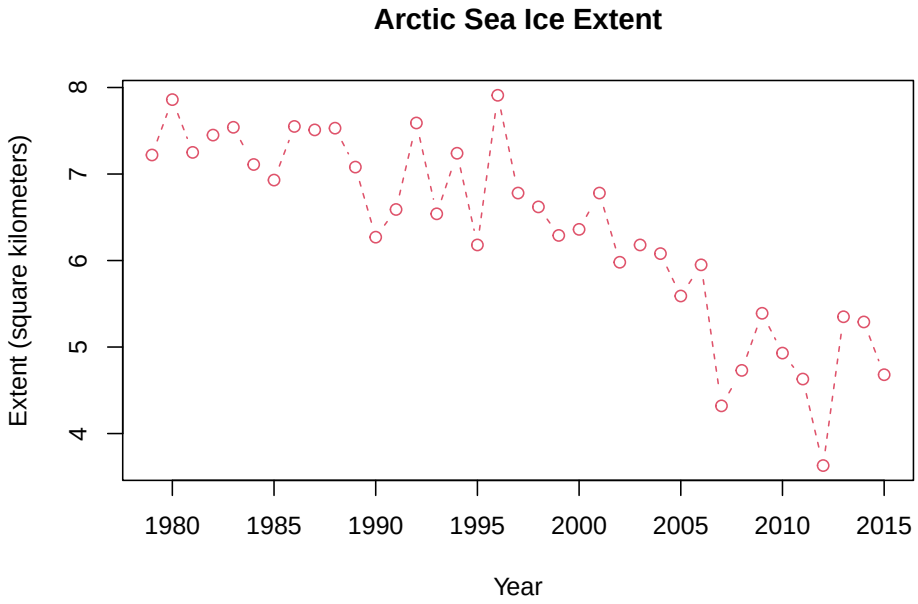
Its always good to include your units. `Extent` and `Area` are in square kilometers. We can say specifically what we want for the y-axis label using `ylab = ...`

```
plot(Extent ~ Year,  # relationship
     type = "b",     # type of plot; Note: b in quotes
     col = 2,        # color; no quotes
     main = "Arctic Sea Ice Extent", # main title, in quotes
     ylab = "Extent (square kilometers)",
     data = SeaIce)  # data
```



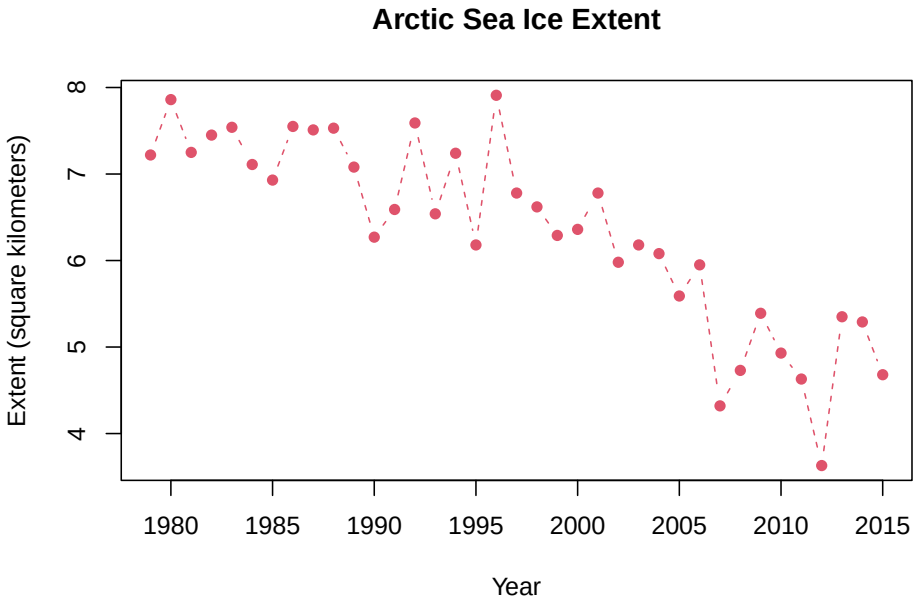
We can change the appearance of the line using `lty = ...`, which stands for "line type":

```
plot(Extent ~ Year,  # relationship
     type = "b",     # type of plot; Note: b in quotes
     col = 2,        # color; no quotes
     main = "Arctic Sea Ice Extent", # main title, in quotes
     ylab = "Extent (square kilometers)",
     lty = 2,        # line type; not quoted
     data = SeaIce)  # data
```



I'm not a fan of the open circles for plotting points; we can change those too using the argument `pch =`.

```
plot(Extent ~ Year,  # relationship
     type = "b",     # type of plot; Note: b in quotes
     col = 2,        # color; no quotes
     main = "Arctic Sea Ice Extent", # main title, in quotes
     ylab = "Extent (square kilometers)",
     lty = 2,        # line type; not quoted
     pch = 16,       # point type; not quoted
     data = SeaIce)  # data
```

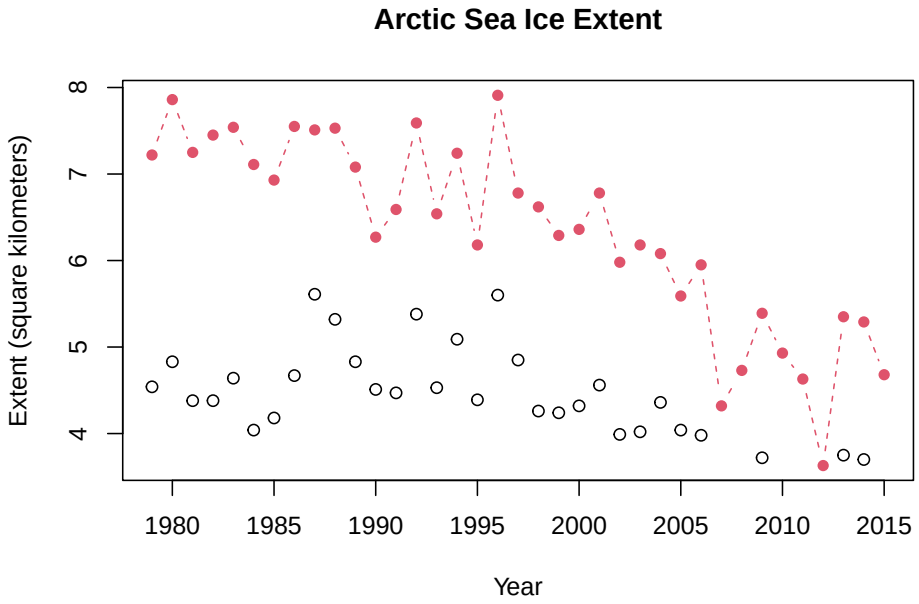


9.1.2 Plotting two columns of data

Often we want to represent two distinct things on our graph. In spreadsheets these are called separate **series** of data. When making a **time series plot** like this one we can add a new column of data using a species command called `points()` which works very similar to `plot()`.

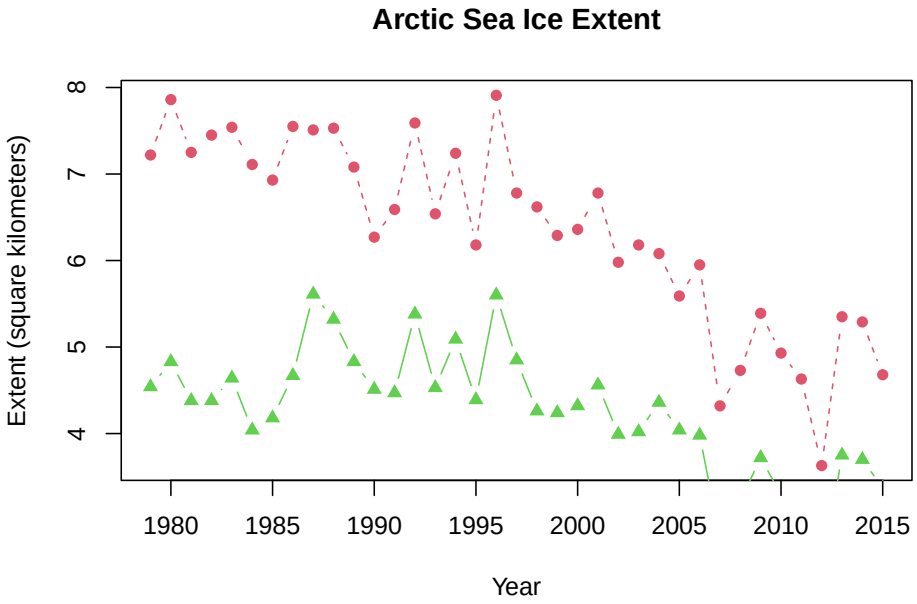
Note: The `points()` command only works if its it precede by a statement from the `plot()` command

```
# Main plot: Extent
plot(Extent ~ Year,  # relationship
     type = "b",     # type of plot; Note: b in quotes
     col = 2,        # color; no quotes
     main = "Arctic Sea Ice Extent", # main title, in quotes
     ylab = "Extent (square kilometers)",
     lty = 2,        # line type; not quoted
     pch = 16,       # point type; not quoted
     data = SeaIce)  # data

# Second column of data: Area
points(Area ~ Year, data = SeaIce)
```



I can now customize the sea ice `Area` part of the plot by add **arguments** to the `points()` statement.

```
# Main plot: Extent
plot(Extent ~ Year,  # relationship
     type = "b",     # type of plot; Note: b in quotes
     col = 2,        # color; no quotes
     main = "Arctic Sea Ice Extent", # main title, in quotes
     ylab = "Extent (square kilometers)",
     lty = 2,        # line type; not quoted
     pch = 16,       # point type; not quoted
     data = SeaIce)  # data

# Second column of data: Area
points(Area ~ Year,
       type = "b",
       col = 3,
       pch = 17,
       data = SeaIce)
```



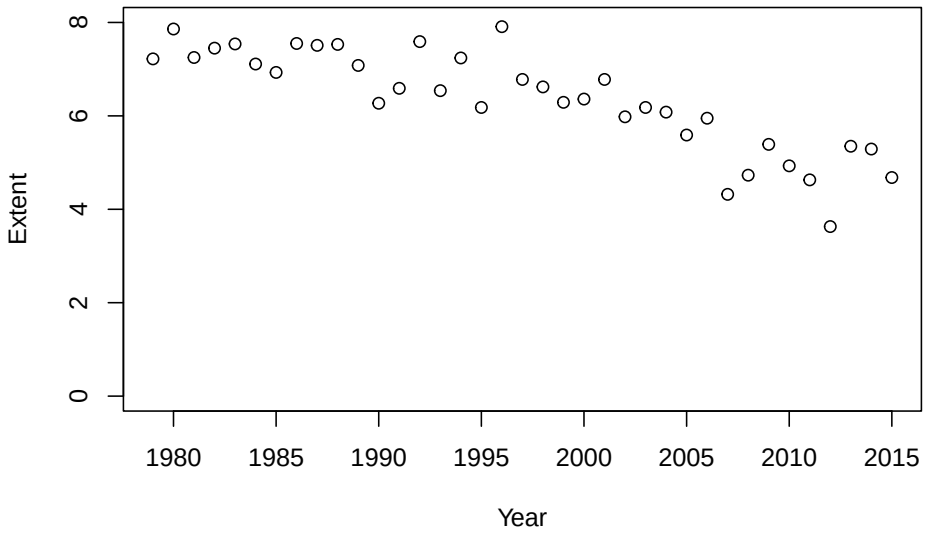
There's a problem with my graph though - can you spot it? It only really becomes apparent when you connect the dots with a line.

9.1.3 Changing the range of a plot axis

Let's go back to our original plot and forget about all the fancy arguments and adding `points()` for a little bit. The problem with the last graph we made is that some points are not showing - if you look in the lower right-hand part points around 2006-2008 and 2011-2013 are not showing up because the y-axis stops around 3.5. We can fix this by adding a new argument which sets the limits of the y-axis: `ylim = ...`. To do this correctly, we have to introduce a new function: `c()`. This is actually one of the most common functions in R. In the `c()` we need to tell R the lower and upper limits we want for the y-axis. Let's do 0 and 8, which will be coded as `c(0,8)`.

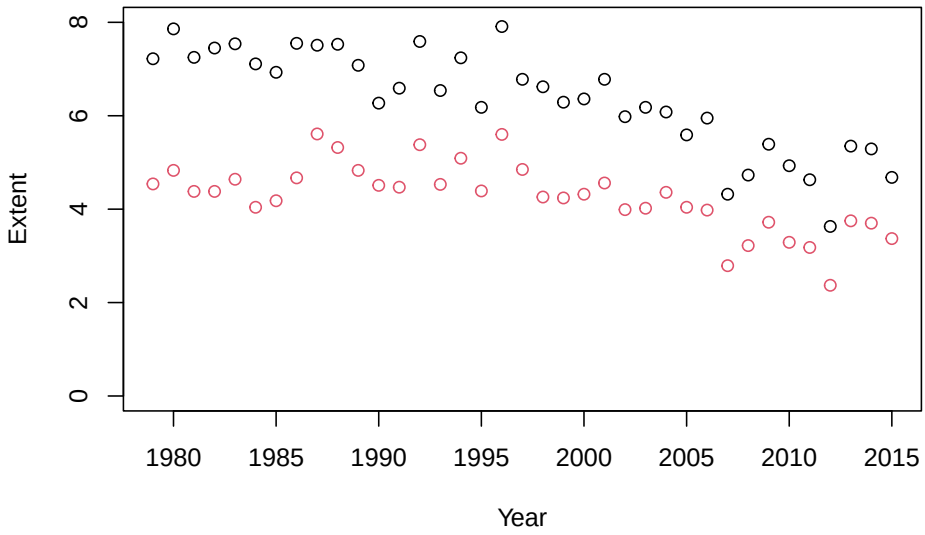
So, to set the y-axis limits we do this:

```
plot(Extent ~ Year,
     ylim = c(0,8), # the c(...) fucntion to set limits
     data = SeaIce)
```



Now we have a bunch more space at the bottom. We can add our `points()` back to see if this work:

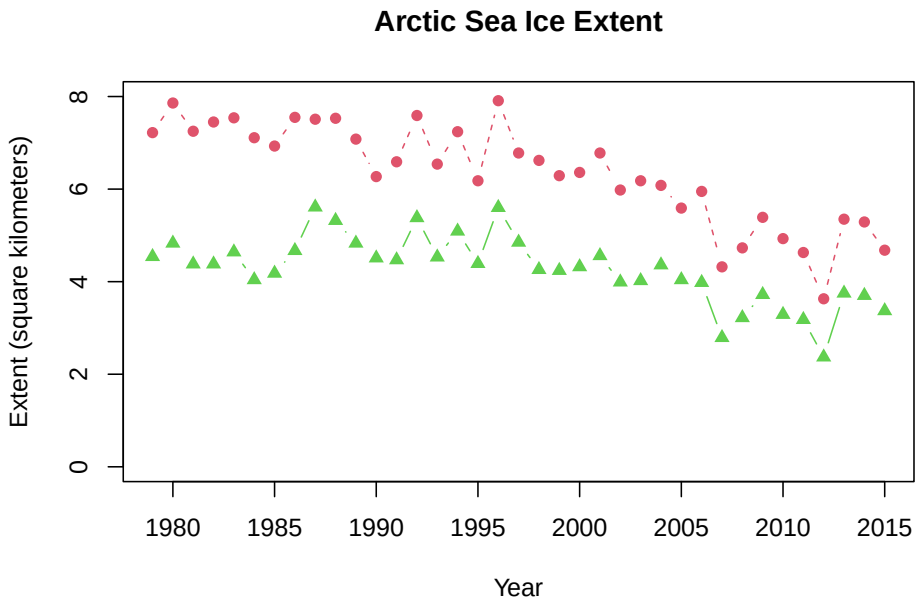
```
plot(Extent ~ Year,
     ylim = c(0,8), # the c(...) fucntion to set limits
     data = SeaIce)
points(Area ~ Year,
       data = SeaIce,
       col = 2)
```



Note: `ylim = ...` only goes in `plot()`, not `points()`. `View()`

Let's re-make our fancy plots now with `ylim = ...` set.

```
# Main plot: Extent
plot(Extent ~ Year,
     type = "b",
     col = 2,
     main = "Arctic Sea Ice Extent",
     ylab = "Extent (square kilometers)",
     lty = 2,
     pch = 16,
     ylim = c(0,8), # <#<== y-axis limits
     data = SeaIce)

# Second column of data: Area
points(Area ~ Year,
       type = "b",
       col = 3,
       pch = 17,
       data = SeaIce)
```



9.2 You try it

9.2.1 Fixer-uppers

Fix the code below so it works

```
plot(Extent , Year,
     data = SeaIce)
```

Fix the code below so it works

```
plot(Extent ~ Year,  # relationship
     type = b,      # type of plot; Note: b in quotes
     col = "2",        # color; no quotes
     data = SeaIce)  # data
```

9.2.2 Intermediate

Make a plot with `Area` on the x-axis and `Extent` on the y-axis.

```
## Write the code below
```

9.2.3 Advanced

Based on the code above, make a plot of the SeaIce data where `Area` appears within the `plot()` statement, and `Extent` is in `points()`.

```
# Write the code below:
```

Chapter 10

Build your own dataframe part I: Vectors

10.1 Dataframes are a type of data structure

For small datasets its often useful to build your own dataframes. Dataframes are **objects** in R; in particular they are a type of **data structure**. "Data structure" is just a fancy way of saying "holding or organizing data."

We'll work with the `SeaIce` data again.

```
library(Stat2Data)
data("SeaIce")
```

If we're curious about what kind of data is in an object we can use the `class()` command to see how is is classified.. Generally the first thing R spits out is the most important.

```
class(SeaIce)
```

```
## [1] "data.frame"
```

We see that `SeaIce is a`data.frame‘.

A related command is `is()`

```
is(SeaIce)
```

```
## [1] "data.frame" "list"       "oldClass"   "vector"
```

This command is more **verbose** and tells of several things, the most relevant of which is that `SeaIce` is a dataframe.

The `SeaIce` dataframe has several columns. We can pull up their names using the `names()` command.

```
names(SeaIce)
```

```
## [1] "Year"   "Extent" "Area"   "t"
```

There are times when we may want to examine just a single column. We can isolate a single column using a species notation which uses a dollar sign. To get the `Extent` column we do this

```
SeaIce$Extent
```

```
##  [1] 7.22 7.86 7.25 7.45 7.54 7.11 6.93 7.55 7.51 7.53 7.08 6.27 6.59 7.59 6.54
## [16] 7.24 6.18 7.91 6.78 6.62 6.29 6.36 6.78 5.98 6.18 6.08 5.59 5.95 4.32 4.73
## [31] 5.39 4.93 4.63 3.63 5.35 5.29 4.68
```

10.2 Dataframes are made from vectors

Running this we get just a stream of numbers. In R, such a stream of numbers is called a **vector**. Like dataframes, vectors are a type of data structure. We can check this with `is()`

```
is(SeaIce$Extent)
```

```
## [1] "numeric" "vector"
```

The first result is `numeric`, which indicates that there are numbers in this vector; the second result is vector.

All columns in a dataframe are are vectors:

```
is(SeaIce$Year)
```

```
## [1] "integer"             "double"              "numeric"
## [4] "vector"              "data.frameRowLabels"
```

```
is(SeaIce$Area)
```

```
## [1] "numeric" "vector"
```

10.3 Make your own vectors

We can make our own vectors using the `c()` command. (`c()` does MANY things in R, so we'll see it a lot).

Here is some more recent data on the Canda Lynx from 1984 to 2002 (Poole 2003, Table 3):

```
c(13445, 8625, 6853, 6953, 6574,
  8265, 9977, 7579, 11542, 7180,
```

```
  4713, 4907, 2819, 5171, 6873,
  6148, 8573, 9361, 11226)
```

```
##  [1] 13445  8625  6853  6953  6574  8265  9977  7579 11542  7180  4713  4907
## [13]  2819  5171  6873  6148  8573  9361 11226
```

We can save this to an R object using a very species function in R called the **assigment operator**

```
lynx.ca <- c(13445, 8625, 6853, 6953, 6574,
  8265, 9977, 7579, 11542, 7180,
  4713, 4907, 2819, 5171, 6873,
  6148, 8573, 9361, 11226)
```

We've now made a brand new object in R called `lynx.ca`. We can see what it is by just typing its name in the console...

```
lynx.ca
```

```
##  [1] 13445  8625  6853  6953  6574  8265  9977  7579 11542  7180  4713  4907
## [13]  2819  5171  6873  6148  8573  9361 11226
```

...and confirming what it is with `is()` and `class()`

```
is(lynx.ca)
```

```
## [1] "numeric" "vector"
```

```
class(lynx.ca)
```

```
## [1] "numeric"
```

The data are from 1984 to 2002. We can make a `year` vector like this

```
year <- c(1984, 1985, 1986, 1987,
          1988, 1989, 1990, 1991, 1992,
          1993, 1994, 1995, 1996, 1997,
          1998, 1999, 2000, 2001, 2002)
```

Its rather tedious to do that, so R has a trick. If we want a sequence of numbers we can use the `seq()` command, which has the arguments `from =...` and `to = ....`

```
year <- seq(from = 1984, to = 2002)
```

10.4 Checking the length of vectors

Typing in data directly into R is errorprone and whenever we do it we should check our work carefully. A first check should be that we entered in all the

numbers; we can do this by getting R to tell us the length of the vector. Can you guess what the command is called?

```
length(lynx.ca)
```

```
## [1] 19
```

Let's check that our year vector is the same length

```
length(year)
```

```
## [1] 19
```

10.5 Another example

Another famous example

Howlett and Majerus. 1987. The understanding of industrial melanism in the peppered moth (*Biston betularia*) .(Lepidoptera: Geometridae). 30: 31-44.

Table 2

```
year.ne <- c(1954, 1961, 1970, 1975, 1981, 1982, 1983, 1984, 1985)

moths.ne <- c(92.95, 94.78, 75, 64.7, 45.9, 50, 42.9, 42.1, 39.6)
```

(Raw data is slightly more complicated, but this works)

```
length(year.ne)
```

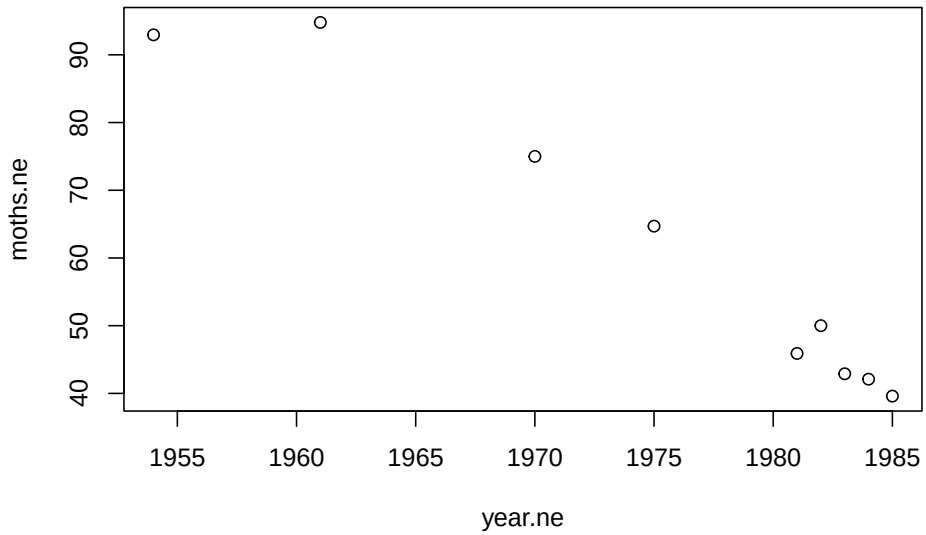
```
## [1] 9
```

```
length(moths.ne)
```

```
## [1] 9
```

```
moths <- data.frame(year.ne,
                    moths.ne)
```

```
plot(moths.ne ~ year.ne, data = moths)
```



Not how R doesn't need extra space for the years where there are no data

10.6 Try it yourself

The original lynx data that comes with R spans the year 1821–1934. Make a sequence of numbers called year.lynx using the `seq()` command.

```
1957:1964
```

```
## [1] 1957 1958 1959 1960 1961 1962 1963 1964
```

There was a hypothesis that sunspot number and/or size impact lynx populations. R has a dataset called sunspots that runs from 1749 to 1983. Make a vector called year year.sunspot using the `seq()` command.

```
1749:1983
```

```
##   [1] 1749 1750 1751 1752 1753 1754 1755 1756 1757 1758 1759 1760 1761 1762 1763
##  [16] 1764 1765 1766 1767 1768 1769 1770 1771 1772 1773 1774 1775 1776 1777 1778
##  [31] 1779 1780 1781 1782 1783 1784 1785 1786 1787 1788 1789 1790 1791 1792 1793
##  [46] 1794 1795 1796 1797 1798 1799 1800 1801 1802 1803 1804 1805 1806 1807 1808
##  [61] 1809 1810 1811 1812 1813 1814 1815 1816 1817 1818 1819 1820 1821 1822 1823
##  [76] 1824 1825 1826 1827 1828 1829 1830 1831 1832 1833 1834 1835 1836 1837 1838
##  [91] 1839 1840 1841 1842 1843 1844 1845 1846 1847 1848 1849 1850 1851 1852 1853
## [106] 1854 1855 1856 1857 1858 1859 1860 1861 1862 1863 1864 1865 1866 1867 1868
## [121] 1869 1870 1871 1872 1873 1874 1875 1876 1877 1878 1879 1880 1881 1882 1883
## [136] 1884 1885 1886 1887 1888 1889 1890 1891 1892 1893 1894 1895 1896 1897 1898
## [151] 1899 1900 1901 1902 1903 1904 1905 1906 1907 1908 1909 1910 1911 1912 1913
## [166] 1914 1915 1916 1917 1918 1919 1920 1921 1922 1923 1924 1925 1926 1927 1928
## [181] 1929 1930 1931 1932 1933 1934 1935 1936 1937 1938 1939 1940 1941 1942 1943
```

```
## [196] 1944 1945 1946 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958
## [211] 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971 1972 1973
## [226] 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983
```

10.7 A complicate dataframe to make

10.8 Build the dataframe

This code makes all of the columns and makes a dataframe

```
aa        <-c('A','C','D','E','F','G','H','I','K','L','M','N','P','Q','R','S','T','V',
MW.da     <-c(89,121,133,146,165,75,155,131,146,131,149,132,115,147,174,105,119,117,204
volume    <-c(67,86,91,109,135,48,118,124,135,124,124,96,90,
              114,148,73,93,105,163,141)
bulkiness <-c(11.5,13.46,11.68,13.57,19.8,3.4,13.69,21.4,
              15.71,21.4,16.25,12.28,17.43,
              14.45,14.28,9.47,15.77,21.57,21.67,18.03)
polarity  <-c(0,1.48,49.7,49.9,0.35,0,51.6,0.13,49.5,0.13,
              1.43,3.38,1.58,3.53,52,1.67,1.66,0.13,2.1,1.61)
isoelectric.pt <-c(6,5.07,2.77,3.22,5.48,5.97,7.59,6.02,9.74,5.98,
                   5.74,5.41,6.3,5.65,10.76,5.68,6.16,5.96,5.89,5.66)
hydrophobe.34 <-c(1.8,2.5,-3.5,-3.5,2.8,-0.4,-3.2,4.5,-3.9,3.8,1.9,
                  -3.5,-1.6,-3.5,-4.5,-0.8,-0.7,4.2,-0.9,-1.3)
hydrophobe.35 <-c(1.6,2,-9.2,-8.2,3.7,1,-3,3.1,-8.8,2.8,3.4,-4.8,
                  -0.2,-4.1,-12.3,0.6,1.2,2.6,1.9,-0.7)
saaH2O       <-c(113,140,151,183,218,85,194,182,211,180,204,158,
                 143,189,241,122,146,160,259,229)
faal.fold    <-c(0.74,0.91,0.62,0.62,0.88,0.72,0.78,0.88,0.52,
                 0.85,0.85,0.63,0.64,0.62,0.64,0.66,0.7,0.86,0.85,0.76)
polar.req    <-c(7,4.8,13,12.5,5,7.9,8.4,4.9,10.1,4.9,5.3,10,
                 6.6,8.6,9.1,7.5,6.6,5.6,5.2,5.4)
freq        <-c(7.8,1.1,5.19,6.72,4.39,6.77,2.03,6.95,6.32,
                10.15,2.28,4.37,4.26,3.45,5.23,6.46,5.12,7.01,1.09,3.3)
charge<-c('un','un','neg','neg','un','un','pos','un','pos',
          'un','un','un','un','un','pos','un','un','un','un','un')
hydropathy<-c('hydrophobic','hydrophobic','hydrophilic','hydrophilic','hydrophobic','ne
              'hydrophobic','hydrophobic','neutral')
volume.cat<-c('verysmall','small','small','medium',
              'verylarge','verysmall','medium','large','large',
              'large','large','small','small','medium','large','verysmall','small','med
polarity.cat<-c('nonpolar','nonpolar','polar','polar',
                'nonpolar','nonpolar','polar','nonpolar',
                'polar','nonpolar','nonpolar','polar','nonpolar','polar',
                'polar','polar','polar','nonpolar','nonpolar','polar')
chemical<-c('aliphatic','sulfur','acidic','acidic','aromatic',
```

```
            'aliphatic','basic','aliphatic','basic','aliphatic','sulfur',
            'amide','aliphatic','amide','basic','hydroxyl','hydroxyl',
            'aliphatic','aromatic','aromatic')

aa_dat <- data.frame(aa,MW.da,volume,bulkiness,
polarity,isoelectric.pt,hydrophobe.34,hydrophobe.35,
saaH2O,faal.fold, polar.req,freq,charge,hydropathy,volume.cat,polarity.cat,chemical)
```

Try plotting one numeric variable against another.

10.8.1 Factors

The amino acid data contains both **numeric** data and vectors of words that represent categories or groups of similar amino acids. When the dataframe first is created R doesn't do anything with it. For example, if we look at a numeric variable, we get this summary information

```
summary(aa_dat$MW.da)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    75.0   118.5   132.5   136.8   150.5   204.0
```

If we do the same summary command on a column of text we get this

```
summary(aa_dat$hydropathy)
```

```
##    Length     Class      Mode
##        20 character character
```

We can help R make sense of this columns usin the command factor().

```
aa_dat$hydropathy <- factor(aa_dat$hydropathy)
```

Note that on the right is the command

```
factor(aa_dat$hydropathy)
```

and on the left we are assigning the output of this function back onto the original column, thus telling R that the words in this column represent group or categories. (The term "factor" comes from statistics and isn't very helpful about what's going on.)

```
summary(aa_dat$hydropathy)
```

```
## hydrophilic hydrophobic     neutral
##           6           8           6
```

Note that if I do this

```
aa_dat <- factor(aa_dat$hydropathy)
```

I **overwrite** the entire aa_dat dataframe object with the data of just the aa_dat$hydropathy column.

10.9 References

Poole, Kim G. 2003. A review of the Canada Lynx, *Lynx canadensis* ,in Canada. Canadian [Field-Naturalist 117: 360-376.[(https://www.canadianfieldnaturalist.ca/index.php/cfn/article/view/738)] DOI: https://doi.org/10.22621/cfn.v117i3.738

Chapter 11

Build your own dataframe

Build a dataframe from the data below and practice making plots

Number of lynx

```
lynx.ca <- c(13445, 8625, 6853, 6953, 6574,
  8265, 9977, 7579, 11542, 7180,
  4713, 4907, 2819, 5171, 6873,
  6148, 8573, 9361, 11226)
```

Year of study

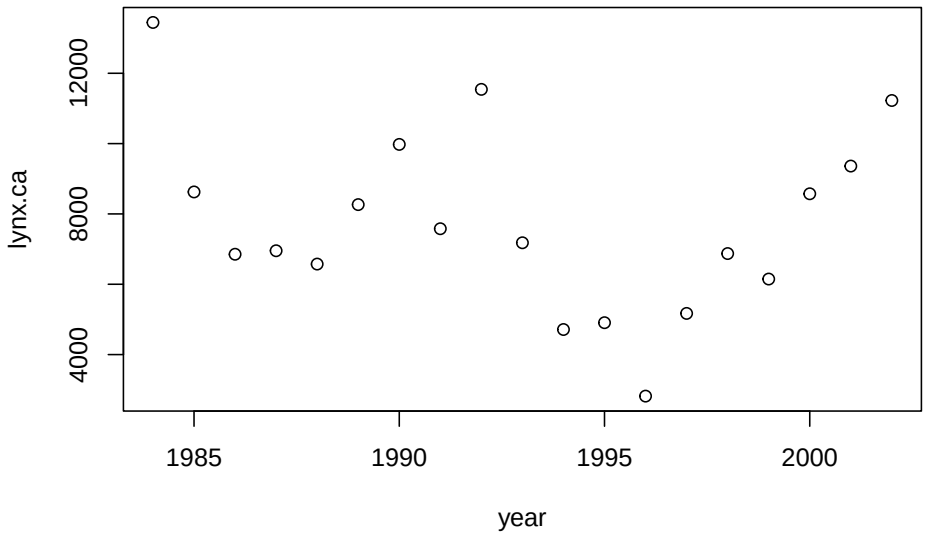
```
year <- c(1984, 1985, 1986, 1987,
          1988, 1989, 1990, 1991, 1992,
          1993, 1994, 1995, 1996, 1997,
          1998, 1999, 2000, 2001, 2002)
```

Make the dataframe.

```
lynx.new <- data.frame(lynx.ca,
                       year)
```

Build a basic plot

```
plot(lynx.ca ~ year, data = lynx.new)
```



As in the lynx dataset that comes with R, we see wild swings in abundance over short periods of time.

11.1 Your turn

Make this plot nice. Change the x axis label, y axis label, give it a main title, etc.

11.2 References

Poole, Kim G. 2003. A review of the Canada Lynx, *Lynx canadensis* ,in Canada. Canadian [Field-Naturalist 117: 360-376.[(https://www.canadianfieldnaturalist.ca/index.php/cfn/article/view/738)] DOI: https://doi.org/10.22621/cfn.v117i3.738