R for Pharmaceutical Sciences

Devin Pastoor

Contents

Welcome

This book is a sum of pouring over dozens of books, countless blogs, and many projects as I developed my own R skills and toolbox. While there are extensive resources available to learn R, two patterns I noticed were: a) most material was very general, forcing a constant 'translation' into datasets or problems faced when dealing with problems specific to pharmaceutical sciences (and in my case, specifically, pharmacometrics) and b) a lot of the scientific examples are quite outdated or take the 'long way around' to get to a solution. This is my attempt to modernize and inform by bringing together the core topics I believe can make someone a successful and productive user of R.

0.1 How to read this book

When writing this book I have made a conscious decision to introduce topics in a pragmatic way to keep those new to programming engaged. In essence, I want people to get 'up and running' as quickly as possible to actually doing things like data visualization, data manipulation, etc. Hence, during the first read-through there may be some sections or pieces of code that feel like 'magic', or may not fully make sense. Bear with me, hopefully all those will get addressed in later sections such as 'Core R' and 'Thinking in R'. This way, by the time you are introduced to some topic like 'what attributes are maintained when adding a vector to a data frame)', there will be a foundation of examples that can help provide additional context to the topics and reduce the feeling of trying to digest all the topics in the abstract.

That said, for more experienced programmers, coming from other languages such as python, javascript, c/c++, java, etc, there may be appreciation in getting up to speed to connect concepts they already know how to perform in one language to how R behaves. If you fall into that category, I would suggest perusing through 'Core R' and 'Thinking in R' either before or in line with the rest of the book as it will help provide a deeper connection to the data structures and functionality available in R to your prior experience(s).

Even more importantly, it does not make sense for me to re-explain the wheel when Hadley has done such a fantastic job, so I defer to explanations of the finer details of how R works, especially 'under the hood', to his book advanced R. This book should be 'required' reading to better appreciate how to master R.

Chapter 1

Reproducible Research

This section covers the conceptual framework behind reproducible research.

1.1 Importance of Reproducible Research

1.2 “Best” Practices in Reproducible Research

Minimal set of guidelines that should be followed to reduce the likelihood of other people being unable to reproduce your results given your document and data.

1. Avoid absolute paths
2. If necessary to set absolute path or read in datasets from other locations, do so in a single, clear location. Minimize the need for people to hunt for locations they need to change paths/pointers.
3. Keep each project in a single directory
4. For any simulation or number generation `set.seed(<number>)` to maintain reproducibility
5. Use R vanilla to avoid differences with `.Rdata` or `.Rprofile`
6. Include `sessionInfo()` with `cache = FALSE` so other people can check if difficulties are due to environment or version differences
7. Before generating a final report, clear your environment (and `cache()` if the full analysis could be completed in a reasonable amount of time) before knitting to make sure the document compiles from scratch properly.

1.3 Reproducible Research Tools

1.4 In-line code

One major goal of reproducible research is a final report should not include numbers derived from the data. Instead, numbers should be printed as part of the code evaluation.

For example, I would not write: In the Theophylline dataset there are 12 unique subjects. Instead, I can use in-line code evaluation to write:

```
There are 'r length(unique(Theoph$Subject))' individuals in the dataset
```

using the backtick character “’ (located to the left of the 1 on a US keyboard) instead of quote as shown above.

The above evaluates to:

There are 12 individuals in the dataset

This can be extended to handle output from models, summary statistics, etc.

Karl Broman also provides a nice knitr in a nutshell tutorial

Chapter 2

Rmarkdown

This documentation will eventually grow to a more complete resource of some "better" practices and examples for how to design and maintain an Rmd lab notebook.

2.1 Good Practices

- When working 'live' on chunks, use a clean R session `rm(list = ls())` so you won't get anything different when you knit (eg. make sure no external code was run that wouldn't run when you knit)
- attach `sessionInfo()` at the bottom of every document!
- better yet `devtools::session_info()` is a new(ish) function that gives more relevant information.
- If you have multiple files or additional complexities using a Makefile may be easier long term. In simple terms, a makefile is a script that re-builds your project so you can just source that script rather than manually clicking knit, etc.
- use chunk labels if possible

2.2 Some 'gotchas' for knitting

- when knitted, each chunk is evaluated based on a working directory of the current file location of the Rmd document. * This directory is reset after each chunk! So no setting in a higher level chunk and forgetting

2.3 Extracting R Code

To *tangle* (extract program code), the function `purl()` will compile all R-code to a single .R file.

```
library(knitr)
purl("your-file.Rmd")
# results in your-file.R in the same directory
```

2.4 Chunk Labels

Think of chunk labels as unique id's in a document. While they are used mainly for geration of external files, naming allows you to reference them elsewhere in your document. Automatically generated figures are also based on chunk-label names.

`{r chunk_name, <additional options>}`

2.5 Global Options

global options can be modified at any point in your document and will affect all chunks below.

The syntax is `opts_chunk$set(<options-you-want,...>)`

2.6 Digits of Output

- Control with `options(scipen = <#>, digits = <#>)` * `scipen` - controls when reported as scientific notation * digits = # digits to report

2.7 Showing/Hiding Output Options

- `echo` - can take a TRUE/FALSE argument for whether to display the code as well as the output (default, TRUE) or just the output (FALSE) **or** can specify certain lines you would like to display
  - `echo=1:2` would display lines 1 and 2 only
  - note: line numbers are based on *expressions* rather than completed lines
    - see here for more details
- `results`
  - `asis` - for when your output is already 'processed', eg when a function already gives you html or latex output. Tells knitr to not treat the code as markdown to be further processed but pass it directly on to the final output.
  - `hide` - like the opposite of `echo`, does not display output. Good if you want to show code, but not print the output.
- `warning/error/message` - whether to display warning/error/message(s).
- `split`
- `include` - whether to include the code chunk in your final document

2.8 Figures

- Alignment - `fig.align = default=center/left/right`
- Path - `fig.path`
- height/width
- `fig.height`
- `fig.width`
- `out.height`, `out.width`
- `fig.retina` ## Caching
- `cache = TRUE`

Do have some nice granular control options however

- update if version changes `version = R.version.string`
- check to see if input file changes `<file>_name=file.info('<file>.csv')$mtime` and re-read data if newer
- check if other chunk updates `dependson='<chunk-name>'` * can also take integer chunk names `dependson = -1` would set dependency for chunk above

2.9 Cross-Referencing

```
{r my-theme}
theme(axis.text.x = element_text(size = 18),
)
```

Then in later chunk

```
qplot(conc, Time, data = Theoph, color = Subject) + <<my-theme>>
```

TODO: flesh out and add additional material from knitr book

2.10 Adding Tables

Knitr has a built in function `kable` that allows for easy creation of tables.

```r
library(knitr)
kable(head(Theoph))
```

Subject	Wt	Dose	Time	conc
1	79.6	4.02	0.00	0.74
1	79.6	4.02	0.25	2.84
1	79.6	4.02	0.57	6.57
1	79.6	4.02	1.12	10.50
1	79.6	4.02	2.02	9.66
1	79.6	4.02	3.82	8.58

It is worth checking out the documentation for kable via `?kable`

By default, the output is a markdown table, which makes printing to the console or evaluating the knitted markdown easy. `kable` also allows direct output into latex, html, pandoc, and rst via the `format` argument

One other highly useful argument is `digits`, which passes all values in numeric columns through the `round()` function before printing them out. This prevents analysis results to print all calculated digits.

```r
AUC_df <- data.frame(ID = 1:5, AUC = runif(5, 10, 100))
kable(AUC_df)
```

ID	AUC
1	17.3
2	85.1
3	64.1
4	24.1
5	10.7

```r
kable(AUC_df, digits = 2)
```

ID	AUC
1	17.3
2	85.1
3	64.1
4	24.1

ID	AUC
5	10.7

QUESTION: can digits be specified for certain columns only?

Chapter 3

Intro to Data Manipulation

All of you have most likely used every single element of this module, but now we will reassess these operators and behaviors to get a deeper understanding - thus allowing us to wield them more effectively.

3.1 Subsetting

The three subsetting (or subscripting) operators available in R are `[`, `[[`, and `$`. There are also some functions such as `subset`. Each has different behaviors and caveats attached that are important when deciding which to use for the intended task.

Subscript	Effect
Positive Numeric Vector	selects items in those indices
Negative Numeric Vector	selects all but those indices
Character Vector	selects items with those names
Logical Vector	selects all TRUE (and NA) items
Missing	selects all missing

You can easily see how each of these works given a simple vector

```r
x <- c(1, 5, NA, 8, 10)
names(x) <- c("a", "b", "c", "d", "e")
x[1]
#> a
#> 1
x[-1]
#>  b  c  d  e
#>  5 NA  8 10
x[c(1:3)]
#>  a  b  c
#>  1  5 NA
x[-c(1:3)]
#>  d  e
#>  8 10
x[is.na(x)]
#>  c
#> NA
```

```
x[!is.na(x)]
#>  a  b  d  e
#>  1  5  8 10
x["b"]
#> b
#> 5
x[c("b", "c")]
#>  b  c
#>  5 NA

# Logical subsetting returns values that you give as true
x[c(TRUE, FALSE, TRUE, FALSE, FALSE)]
#>  a  c
#>  1 NA
# but don't forget about the recycling rules!
x[c(TRUE, FALSE)]
#>  a  c  e
#>  1 NA 10

# if you call a specific index more than once it will be returned more than once
x[c(2, 2)]
#> b b
#> 5 5
```

By default, `[` will simplify the results to the lowest possible dimensionality. That is, it will reduce any higher dimensionality object to a list or vector. This is because if you select a subset, R will coerce the result to the appropriate dimensionality. We will give an example of this momentarily. To stop this behavior you can use the `drop = FALSE` option.

For higher dimensionality objects, rows and columns are subset individually and can be combined in a single call

```
Theoph[c(1:10), c("Time", "conc")]
#>     Time  conc
#> 1   0.00  0.74
#> 2   0.25  2.84
#> 3   0.57  6.57
#> 4   1.12 10.50
#> 5   2.02  9.66
#> 6   3.82  8.58
#> 7   5.10  8.36
#> 8   7.03  7.47
#> 9   9.05  6.89
#> 10 12.12  5.94
```

`[[` and `$` allow us to take out components of the list.

Likewise, given data frames are lists of column, `[[` can be used to extract a column from data frames.

`[[` is similar to `[`, however, it only returns a single value. `$` is shorthand for `[[` but is only available for character subsetting

There are two additional important distinctions between `$` and `[[`

1. `$` can not be used for column names stored as a variable:

```r
id <- "Subject"
Theoph$id
Theoph[[id]]
```

2. `$` allows for partial matching, though this is not advised given code completion engines in this day and age

```r
names(Theoph)
# $ allows for partial matching
head(Theoph$Sub)
head(Theoph$"Subj")
head(Theoph$Subject)

# '[[' does not
head(Theoph[["Subj"]])
```

Using `$` can lead to unintended consequences if you're looking to grab a column of a certain name that isn't there but a partial match is - especially if this is nested in a function where it isn't clear immediately

`[` and `[[` are both useful for different tasks. In a general sense you use them to accomplish the following:

	Simplifying	Preserving
Vector	`x[[1]]`	`x[1]`
List	`x[[1]]`	`x[1]`
Factor	`x[1:4, drop = T]`	`x[1:4]`
Array	`x[1, ]`, `x[, 1]`	`x[1, , drop = F]`, `x[, 1, drop = F]`
Data frame	`x[, 1]`, `x[[1]]`	`x[, 1, drop = F]`, `x[1]`

There are benefits for each - simplying is often beneficial when you are looking for a result. Preserving is often beneficial in the programming context when you want to keep the results and structure the same.

An easy way to think about it:

> "If list `x` is a train carrying objects, then `x[[5]]` is the object in car 5; `x[4:6]` is a train of cars 4-6." — [@RLangTip](http://twitter.com/#!/RLangTip/status/118339256388304896)

One thing to note: S3 and S4 objects can override the standard behavior of `[` and `[[` so they behave differently for different types of objects. This can be useful for controlling simplification vs preservation behavior.

3.1.1 Try it yourself

- create a vector, list, and dataframe.
- Extract elements using [, [[, and $
- What are the type and attributes that remain for the extracted piece of information
- Quickly brainstorm a couple situations that these could be important to remember for more complex tasks

3.2 Logical Subsetting

One of the most common ways to subset rows is to use **logical subsetting**.

Let's take a look

```
Theoph[Theoph$Subject ==1,]
#>    Subject   Wt Dose  Time  conc
#> 1        1 79.6 4.02  0.00  0.74
#> 2        1 79.6 4.02  0.25  2.84
#> 3        1 79.6 4.02  0.57  6.57
#> 4        1 79.6 4.02  1.12 10.50
#> 5        1 79.6 4.02  2.02  9.66
#> 6        1 79.6 4.02  3.82  8.58
#> 7        1 79.6 4.02  5.10  8.36
#> 8        1 79.6 4.02  7.03  7.47
#> 9        1 79.6 4.02  9.05  6.89
#> 10       1 79.6 4.02 12.12  5.94
#> 11       1 79.6 4.02 24.37  3.28
```

We just subset the Theoph dataframe to only give us back the rows in which Subject equals 1. How does R go about doing this? Logical subsetting!

Notice what we get when we simply ask for `Theoph$Subject == 1`

```
Theoph$Subject ==1
#>   [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#>  [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [45] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [67] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [78] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [89] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [100] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [111] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [122] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

It doesn't give us back the values for the rows where subject equals one, rather, it gives us back a vector of `TRUE` or `FALSE` values.

So, in reality, we are using the logical subsetting rules to extract the rows of the dataframe that come back `TRUE` from our logical query.

We can do this 'by hand' to show whats going on

```
subj <- ifelse(Theoph[["Subject"]] ==1, TRUE, FALSE)
subj
#>   [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#>  [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [45] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [67] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [78] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#>  [89] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [100] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [111] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
#> [122] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
Theoph[subj,]
```

```
#>    Subject   Wt Dose  Time  conc
#> 1        1 79.6 4.02  0.00  0.74
#> 2        1 79.6 4.02  0.25  2.84
#> 3        1 79.6 4.02  0.57  6.57
#> 4        1 79.6 4.02  1.12 10.50
#> 5        1 79.6 4.02  2.02  9.66
#> 6        1 79.6 4.02  3.82  8.58
#> 7        1 79.6 4.02  5.10  8.36
#> 8        1 79.6 4.02  7.03  7.47
#> 9        1 79.6 4.02  9.05  6.89
#> 10       1 79.6 4.02 12.12  5.94
#> 11       1 79.6 4.02 24.37  3.28
```

Logical subsetting is at the core of many of R's operations. Any time you're matching, or checking with `is.*` you are using logical subsetting to test the condition you're looking for then returning the `TRUE` results

3.3 Common Subsetting Situations and Some Useful Functions

Now that you've gotten your feet wet with the basics of subsetting, lets check-in with some of the commonly used operators that give us some enhanced subsetting functionality.

Note: These all take advantage of logical subsetting:

Take a moment to prod through to documentation for the other operations. Note for things operations like `%in%` or `&` to query help you need to add a single quote around it like so `?'%in%'`

This is a good chance for us to take a deeper look @ both how these functions work and how to read the documentation

pause to look @ %in%, is.na, and which documentation

- `%in%`
- `is.na`
- `!`
- `duplicated`
- `unique`
- `&`
- `|`
- `any`
- `all`
- `which`

It can be useful to make yourself a brief 'cheat sheet' of some of the common operations you use to reference when you're thinking about what you are trying to do what you want your output to be.

For example:

- `%in% - compares values from the input vector with a table vector and returns T/F. * input/output are coerced to vectors and then type-matched before comparison. Factors, lists converted to character vectors! * Never returns NA, making it useful forif`‘ conditions
  - Can be slow for long lists and best avoided for complex cases

There is an interesting nugget of information in the documentation - that the input is coerced to a vector then type-matched.

So what is going on in these two situations?

```
ISM <- c(TRUE, FALSE, TRUE, FALSE)
ISMnums <- c(1, 0, 1, 0)
SUBJ <- c(1, 2, 3, 4)
ISM %in% SUBJ
#> [1]  TRUE FALSE  TRUE FALSE
which(ISM %in% SUBJ)
#> [1] 1 3
ISMnums %in% SUBJ
#> [1]  TRUE FALSE  TRUE FALSE
which(ISMnums %in% SUBJ)
#> [1] 1 3

ISM <- c(TRUE, FALSE, TRUE, FALSE)
SUBJ <- factor(c(1, 2, 3, 4))
ISM %in% SUBJ
#> [1] FALSE FALSE FALSE FALSE
which(ISM %in% SUBJ)
#> integer(0)
ISMnums %in% SUBJ
#> [1]  TRUE FALSE  TRUE FALSE
which(ISMnums %in% SUBJ)
#> [1] 1 3
```

hint: look @ ID numbers

The moral of the story, is make sure you know what is going on under the hood - sometimes you can get the 'right' answer for the wrong reasons.

3.3.1 Pop quiz question

What's wrong with this subsetting command? `dosingdf <- df[unique(df$ID),]`

3.4 Data Manipulation

Now that we can slice and dice our data how we want - let's examine how we can actually manipulate the data

Our goal for this section is to be able to: - rename columns systematically - reorder and rearrange rows and columns to our needs - create new columns

3.5 renaming columns

Quick renaming of columns can be easily accomplished using the dplyr `rename` command (originally from plyr) with the structure:

`dataframe <- rename(dataframe, c("oldcolname1" = "newcolname1", ...))`

```
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
names(Theoph)
#> [1] "Subject" "Wt"      "Dose"    "Time"    "conc"
Theoph <- rename(Theoph, ID = Subject)
names(Theoph)
#> [1] "ID"   "Wt"   "Dose" "Time" "conc"
```

This can also be done directly without PKPDmisc, however is a bit more verbose

```
names(Theoph)[names(Theoph)=="Subject"] <- "ID"
```

Pause for a second - given what we've learned about subsetting - what is going on based on the way we've constructed the renaming.

Answer: `names(Theoph)` creates a vector of names - the `names(Theoph) == "Subject"` logically subsets the vector to identify which index matches the query. `<- "ID"` is to assign a new value to that index location(s).

Column names can be directly accessed using the `colnames` function (or for dataframes or lists simply `names`), and you can rename them all directly by giving it a vector of names.

```
colnames(Theoph) <- c("hello", "there")
colnames(Theoph)
#> [1] "hello" "there" NA      NA      NA
rm(Theoph)
```

As you can see, this can be dangerous due to not giving the proper length vector (remember R's recycling rule!), likewise, if the order of columns changes unexpectedly, your vector could rename columns incorrectly.

There are a couple things directly accessing all the colnames can be useful for though.

For example, capitalization of columns can often be inconsistent and frustrating. This can be quickly fixed by converting all columns to uppercase or lowercase using `toupper()` and `tolower()`

```
names(Theoph)
#> [1] "Subject" "Wt"      "Dose"    "Time"    "conc"
names(Theoph) <- toupper(names(Theoph))
names(Theoph)
#> [1] "SUBJECT" "WT"      "DOSE"    "TIME"    "CONC"
rm(Theoph)
```

3.6 reordering rows/columns

When reordering columns in a dataframe you can actually think of it as creating a new dataframe in which the columns get created in the way you order them.

```
newTheoph <- Theoph[c("Subject", "Time", "conc", "Dose", "Wt")]
head(Theoph)
#>   Subject   Wt Dose Time  conc
#> 1       1 79.6 4.02 0.00  0.74
#> 2       1 79.6 4.02 0.25  2.84
#> 3       1 79.6 4.02 0.57  6.57
#> 4       1 79.6 4.02 1.12 10.50
#> 5       1 79.6 4.02 2.02  9.66
#> 6       1 79.6 4.02 3.82  8.58
head(newTheoph)
#>   Subject Time  conc Dose   Wt
#> 1       1 0.00  0.74 4.02 79.6
#> 2       1 0.25  2.84 4.02 79.6
#> 3       1 0.57  6.57 4.02 79.6
#> 4       1 1.12 10.50 4.02 79.6
#> 5       1 2.02  9.66 4.02 79.6
#> 6       1 3.82  8.58 4.02 79.6
```

Most of the time since you don't want to create a new data frame every time you reorder, you can simply overwrite the old data frame. `Theoph <- Theoph[c("Subject", "Time", "conc", "Dose", "Wt")]`

Just like all other subsetting we've gone over, we can also organize by index

`Theoph <- Theoph[c(1, 4, 5, 3, 2)]`

I suggest against it unless you have good reason. (ie you know your code will always be a specific structure) Even then, while faster to type than named indices, it makes legibility more difficult, and modification down the line more tedious trying to keep track.

3.7 Keys

Similar to columns, rows also have names. As you slice and dice and reorder a dataset it can get pretty ugly, so if the need arises, row names can be rest by `rownames(df) <- NULL`

Rows can always be referred by their name. They are also structurally distinguishable by their content.

A **key** is the set of columns that can uniquely distinguish every row. Different datasets can have keys of varying complexity (number of columns)

A basic dosing dataset key may be as simple as ID, however for a cross-over clinical trial a dataset may be keyed on ID, time, and cohort.

The general relationship between **key columns** and other columns is:

> Key columns represent unique objects (persons, groups, sites, etc) and the other columns should characterize these objects

For example, a person might be the key column for different concentration, time, wt, etc measurements, thus if you are 'extracting' information you'd likely want it based on the key column, so for example by max concentration by individual)

R does not explicitly recognize keys - it is up to you to keep track. Keys become increasingly important when delving into advanced data manipulation. Using dplyr `group_by` makes working with keys much more straightforward than in base R.

Chapter 4

Core R

The objectives of this section is to take a number of concepts that you use frequently but may have not questioned the 'why' that R behaves that way.

The overarching theme in this documentation is to "concisely" present the concept in a way to help you to think "in terms of R" rather than trying to memorize patterns. This will be supplemented by examples to help reinforce the concepts and help you on your journey to become an R-ninja.

4.1 Data Structures

R has a number of ways of storing information. The quick way to visualize the possibilities is as such:

	Homogeneous	Heterogeneous
1d	Atomic vector	List
2d	Matrix	Data frame
nd	Array	

Homogeneous - all elements must be of the same type (mode)

Heterogeneous - elements can be of different type (mode)

A **Type/Mode** indicates how R stores the information in memory

- numeric
- double
- integer
- logical
- character
- list of pointers
- function

The term **Mode** and **Type** are virtually interchangeable - from here out we will use the term **Type** as it is more commonly discussed. ('mode' was inherited mostly from S/S-plus nomenclature/definition)

4.2 Vectors

Data structures in R can be boiled down to a Vector. The most basic vector is an atomic vector.

Vectors have 4 key components:

- *contents* - the information
- *type* - what type of information it stores
- *length* - how long it is
- *attributes* - additional meta data

These components can be easily accessed as such:

```
# example vector
numeric_vector <- c(1, 2, 3)

# access contents by calling the vector by name
numeric_vector
#> [1] 1 2 3

# see vector type
typeof(numeric_vector)
#> [1] "double"

# check length
length(numeric_vector)
#> [1] 3

# see additional attributes
attributes(numeric_vector)
#> NULL
```

There are two useful functions for handling data structures: `is.*` and `as.*`

- `is.*` is a testing function that returns TRUE or FALSE
- `as.*` is a coercion function - it attempts to convert the input to the requested data structure

As an example with vectors:

```
numeric_vector <- c(1,2,3)
typeof(numeric_vector)
#> [1] "double"
is.numeric(numeric_vector) # note is.numeric will return TRUE for both doubles and integers
#> [1] TRUE
is.list(numeric_vector)
#> [1] FALSE

#coerce to list
numeric_vector <- as.list(numeric_vector)
typeof(numeric_vector)
#> [1] "list"
```

tidbit - `is.null` determines whether an Object is empty (has no content). `NULL` is often used to represent objects of zero length and is returned by expressions and functions whose value is undefined. Keep `is.null` filed away for when we get to function writing, as it is an excellent way to control behavior under certain circumstances.

4.3 Coercion

Coercion is a tricky topic in R, that will rear its head throughout one's R career. While it would be extremely difficult to cover all cases in a robust nature, at least understanding a bit of what is going on under the hood can help set expectations, and when odd behavior is occuring think back to For homogeneous vectors, if you attempt to combine elements of different types, it will pick the class of the first element and coerce all others to that type

```
c("hello", 1, FALSE)
#> [1] "hello" "1"     "FALSE"
```

Note: If a logical vector is coerced into a numeric TRUE becomes 1 and FALSE becomes 0 - this can be used to 'count' the number of TRUE/FALSES easily using sum()

- vectors can be indexed by position: v[1] refers to the first element of the vector
- vectors can be indexed by multiple positions using c() - v[c(1,2)] will return a sub-vector the first and second element of 'v'

4.3.1 Coercion chain

To help defensively prepare yourself for handling coersion, it is helpful to understand the heirarchy of coersion:

```
fac_test <- factor(c("test1", "test2"))
fac_test
#> [1] test1 test2
#> Levels: test1 test2
str(c(fac_test, 1, TRUE, "1" ))
#>  chr [1:5] "1" "2" "1" "TRUE" "1"
str(c(as.factor("test"), 1, TRUE  ))
#>  num [1:3] 1 1 1
str(c(as.factor("test"), TRUE  )) ## numerical representation of factor
#>  int [1:2] 1 1
```

Whichever is HIGHEST, all others will be converted to match:

character <- numeric <- factor <- boolean

Notice, one other perculiarity of R, in that the factor test1/test2 are converted to values 1 and 2. This behavior will be discussed in more detail later.

4.4 Attributes

Attributes are additional metadata about an object. The most common 3 attributes are: * `names()` - character vector of element names * `dim()` - the structure of the object * `class()` - used to implement an object system (described later)

You can give a vector names in three ways:

- During creation: `x <- c(a = 1, b = 2, c = 3)`
- By modifying a vector in place: `x <- 1:3; names(x) <- c("a", "b", "c")`
- By creating a modified vector: `x <- setNames(1:3, c("a", "b", "c"))`

`dim()` passes additional structural information to the vector. Higher level data structures are simply atomic vectors with the addition of the dim() attribute.

This is an important characteristic to remember as it can help conceptualize the implications of what will happen if you are trying to coerce a data structure into a different type.

As vectors are coerced into higher order structures (eg. matrices and arrays) R handles this by default in a column-wise manner.

```
a <- matrix(1:6)
a
#>      [,1]
#> [1,]    1
#> [2,]    2
#> [3,]    3
#> [4,]    4
#> [5,]    5
#> [6,]    6
```

An element in a matrix can be indexed exactly like an atomic vector

```
a <- matrix(1:6)
a[2]
#> [1] 2
a[4]
#> [1] 4
```

In addition, R can refer to the dimensionality via:

DS[row, column]

```
a <- matrix(1:6)
# give first row
a[1,]
#> [1] 1
# give first column
a[,1]
#> [1] 1 2 3 4 5 6
```

Beyond the 3 common attributes, additional attributes can be assigned; however it is important to note that when a vector is modified most attributes beyond the 3 listed above are lost. This rears its head frequently with custom types - for example the `haven` package, which reads in sas files and will add attributes such as `label` that are present in SAS, can be lost during convertions due to how R removes attributes.

```
conc <- c(9.62, 3.1, 2, 0.3)
attr(conc, "units") <- "ug/mL"
conc
#> [1] 9.62 3.10 2.00 0.30
#> attr(,"units")
#> [1] "ug/mL"
attributes(conc)
#> $units
#> [1] "ug/mL"
conc <- as.data.frame(conc)
conc
#>   conc
#> 1 9.62
#> 2 3.10
#> 3 2.00
#> 4 0.30
attributes(conc)
```

```
#> $names
#> [1] "conc"
#>
#> $row.names
#> [1] 1 2 3 4
#>
#> $class
#> [1] "data.frame"
```

In a similar vein, for the package `dplyr`, which will become a key part of one's analysis toolkit, columns or vectors with additional attributes actually cause errors as dplyr is conservative, and rather than accidentally stripping key attributes before calculations, dplyr just errors out, so you must manually remove the additional attributes.

Attributes can be removed by setting them to `NULL`, so for the above example, we can strip the `units` attribute by setting it to NULL

```
attr(conc, "units") <- NULL
```

4.5 Lists

Lists offer the ability to combine objects of different types. They also separate themselves from atomic vectors as they have the capability of **recursion** - Lists can contain nested lists.

```
list(list(1:3), list("hello", "there"), TRUE)
#> [[1]]
#> [[1]][[1]]
#> [1] 1 2 3
#>
#>
#> [[2]]
#> [[2]][[1]]
#> [1] "hello"
#>
#> [[2]][[2]]
#> [1] "there"
#>
#>
#> [[3]]
#> [1] TRUE
```

To combine multiple lists into one large list, c() will coerce all lists to vectors and vectors to individual elements then combine them.

```
str(list(list(1:3), list("hello", "there"), TRUE, 1:3))
#> List of 4
#>  $ :List of 1
#>   ..$ : int [1:3] 1 2 3
#>  $ :List of 2
#>   ..$ : chr "hello"
#>   ..$ : chr "there"
#>  $ : logi TRUE
#>  $ : int [1:3] 1 2 3
str(c(list(1:3), list("hello", "there"), TRUE, 1:3))
```

```
#> List of 7
#>  $ : int [1:3] 1 2 3
#>  $ : chr "hello"
#>  $ : chr "there"
#>  $ : logi TRUE
#>  $ : int 1
#>  $ : int 2
#>  $ : int 3
```

Note: As shown in the examples, the structure of an object can be shown with `str`

4.6 Data Frames

The most commonly used method for storing data is the data frame. At its core, a data frame is a list of equal-length vectors. You can think of it as having the same properties as both a matrix and a list. (This means you can use the properties of both for things like indexing and subsetting)

A couple useful commands for dataframe attributes:

- `names()`
- `colnames()`
- `rownames()`
- `length()`
- `nrow()`
- `ncol()`

You can easily see how the properties of both 1-dimensional structures (list) and 2-dimensional structures (matrix) come into play.

```
# can initialize with named vectors
df <- data.frame(time = 1:6, conc = c(9.1, 8.5, 7.3, 4.2, 3.8, 2.5), race = "male")
## list-like
# length
length(df)
#> [1] 3
# subset for individual element by index or name like a vector
df[1]
#>   time
#> 1    1
#> 2    2
#> 3    3
#> 4    4
#> 5    5
#> 6    6
df["time"]
#>   time
#> 1    1
#> 2    2
#> 3    3
#> 4    4
#> 5    5
#> 6    6

## array-like
```

```
# subset by dimension (remember - ds[row, column])
df[1,]
#>   time conc race
#> 1    1  9.1 male
colnames(df)
#> [1] "time" "conc" "race"
nrow(df)
#> [1] 6
```

4.7 Creation vs Coercion

You can create new data structure with `datastructure()` - ie `data.frame()` As mentioned previously, you can coerce an object to a different data structure with `as.*` - ie: `as.data.frame()`

An additional note when coercing to data frames:

- a vector will yield a one-column data frame
- a list will yield one column for each element; it's an error if they're not all the same length
- a matrix will yield a data frame with the same number of columns

4.8 Combining Data Structures/Objects

There are a number of ways to combine multiple objects. The simplest are `c()`, `cbind()`, and `rbind()`

```
time <- 1:6
conc <- c(9.1, 8.5, 7.3, 4.2, 3.8, 2.5)
c(time, conc)
#>  [1] 1.0 2.0 3.0 4.0 5.0 6.0 9.1 8.5 7.3 4.2 3.8 2.5
rbind(time,conc)
#>      [,1] [,2] [,3] [,4] [,5] [,6]
#> time  1.0  2.0  3.0  4.0  5.0  6.0
#> conc  9.1  8.5  7.3  4.2  3.8  2.5
cbind(time,conc)
#>      time conc
#> [1,]    1  9.1
#> [2,]    2  8.5
#> [3,]    3  7.3
#> [4,]    4  4.2
#> [5,]    5  3.8
#> [6,]    6  2.5
```

While `cbind()` may seem like an easy option to quickly create df's be careful of coercion:

```
class(cbind(time, conc))
#> [1] "matrix"
```

```
class(data.frame(time, conc))
#> [1] "data.frame"
```

```
time <- 1:6
conc <- c("9.1", 8.5, 7.3, 4.2, 3.8, 2.5)
cbind(time,conc)
#>      time conc
#> [1,] "1"  "9.1"
#> [2,] "2"  "8.5"
#> [3,] "3"  "7.3"
#> [4,] "4"  "4.2"
#> [5,] "5"  "3.8"
#> [6,] "6"  "2.5"
```

That seems easy to catch, but what about something like this:

```
as.data.frame(cbind(time, conc))
#>   time conc
#> 1    1  9.1
#> 2    2  8.5
#> 3    3  7.3
#> 4    4  4.2
#> 5    5  3.8
#> 6    6  2.5
```

It prints out to the console looking like numbers but…

```
str(as.data.frame(cbind(time, conc)))
#> 'data.frame':    6 obs. of  2 variables:
#>  $ time: Factor w/ 6 levels "1","2","3","4",..: 1 2 3 4 5 6
#>  $ conc: Factor w/ 6 levels "2.5","3.8","4.2",..: 6 5 4 3 2 1
```

They are actually coerced to a matrix of character factors

This is because cbind will create a matrix unless one of the objects is already a data frame.

The best way to avoid this is to be careful when deciding if you want to *coerce* (as.*) or initialize a new data frame.

A good habit is to just to use data.frame() directly unless you have good reason for the coercion.

```
# coerce objects together into a data frame
str(as.data.frame(cbind(time, conc)))
#> 'data.frame':    6 obs. of  2 variables:
#>  $ time: Factor w/ 6 levels "1","2","3","4",..: 1 2 3 4 5 6
#>  $ conc: Factor w/ 6 levels "2.5","3.8","4.2",..: 6 5 4 3 2 1
# instead create new data frame
str(data.frame(time, conc))
#> 'data.frame':    6 obs. of  2 variables:
#>  $ time: int  1 2 3 4 5 6
#>  $ conc: Factor w/ 6 levels "2.5","3.8","4.2",..: 6 5 4 3 2 1
```

Likewise, the safest way to use cbind() is to ensure all objects are of the same type or already a higher level data structure such as a dataframe.

4.9 One last combining catch

That is, what happens if vectors being combined to are of *unequal length*?

R handles this through something called a *Recycling Rule*.

When R performs operations, it does so element-by-element. When combining multiple vecotrs it does so in pairs. When R reaches the end of the shorter vector it starts over from the first element and keeps filling to the length of the longer vector.

This can be convenient when you want to add a new column with a single value:

```
id <- 1:6
race <- "caucasian"
data.frame(id, race)
#>   id      race
#> 1  1 caucasian
#> 2  2 caucasian
#> 3  3 caucasian
#> 4  4 caucasian
#> 5  5 caucasian
#> 6  6 caucasian
```

But can be dangerous and lead to unintended behavior:

```
id <- 1:6
race <- c("caucasian", "black")
data.frame(id, race)
#>   id      race
#> 1  1 caucasian
#> 2  2     black
#> 3  3 caucasian
#> 4  4     black
#> 5  5 caucasian
#> 6  6     black
```

4.10 Factors

Factors in R are a tricky beast and I haven't spent as much time as I'd like writing up this section so I will give the cliff notes.

- unordered factors cannot be sorted
- to convert a numeric factor back to a numeric use the command `as.numeric(as.character(factor))`
- the as.numeric(as.character(...)) idiom is so prevalent, a convenient wrapper in `PKPDmisc` is provided: `as_numeric()`
- factors have a couple key arguments when dealing with them `levels` - are the values the factor takes `labels` - are an optional value of labels that can be used to name the factors

This will be hopefully a good example for us to examine.

```
Theoph3 <- Theoph[with(Theoph, as_numeric(Subject) < 4),]
within(Theoph3, {id <- factor(Subject,
                                levels = c(1, 2, 3),
                                labels = c("John", "Mary", "Joe")
                                )
                 subnum <- as.numeric(Subject)
                 subcharnum <- as.numeric(as.character(Subject))
                 id2 <- factor(subnum,
                                levels = c(1, 2, 3),
                                labels = c("John", "Mary", "Joe")
                                )
                 id3 <- factor(subnum,
                                levels = c(11, 6, 5),
                                labels = c("John", "Mary", "Joe")
                                )
```

```
            }

)
#>    Subject   Wt Dose  Time  conc  id3  id2 subcharnum subnum   id
#> 1        1 79.6 4.02  0.00  0.74 John <NA>          1     11 John
#> 2        1 79.6 4.02  0.25  2.84 John <NA>          1     11 John
#> 3        1 79.6 4.02  0.57  6.57 John <NA>          1     11 John
#> 4        1 79.6 4.02  1.12 10.50 John <NA>          1     11 John
#> 5        1 79.6 4.02  2.02  9.66 John <NA>          1     11 John
#> 6        1 79.6 4.02  3.82  8.58 John <NA>          1     11 John
#> 7        1 79.6 4.02  5.10  8.36 John <NA>          1     11 John
#> 8        1 79.6 4.02  7.03  7.47 John <NA>          1     11 John
#> 9        1 79.6 4.02  9.05  6.89 John <NA>          1     11 John
#> 10       1 79.6 4.02 12.12  5.94 John <NA>          1     11 John
#> 11       1 79.6 4.02 24.37  3.28 John <NA>          1     11 John
#> 12       2 72.4 4.40  0.00  0.00 Mary <NA>          2      6 Mary
#> 13       2 72.4 4.40  0.27  1.72 Mary <NA>          2      6 Mary
#> 14       2 72.4 4.40  0.52  7.91 Mary <NA>          2      6 Mary
#> 15       2 72.4 4.40  1.00  8.31 Mary <NA>          2      6 Mary
#> 16       2 72.4 4.40  1.92  8.33 Mary <NA>          2      6 Mary
#> 17       2 72.4 4.40  3.50  6.85 Mary <NA>          2      6 Mary
#> 18       2 72.4 4.40  5.02  6.08 Mary <NA>          2      6 Mary
#> 19       2 72.4 4.40  7.03  5.40 Mary <NA>          2      6 Mary
#> 20       2 72.4 4.40  9.00  4.55 Mary <NA>          2      6 Mary
#> 21       2 72.4 4.40 12.00  3.01 Mary <NA>          2      6 Mary
#> 22       2 72.4 4.40 24.30  0.90 Mary <NA>          2      6 Mary
#> 23       3 70.5 4.53  0.00  0.00  Joe <NA>          3      5  Joe
#> 24       3 70.5 4.53  0.27  4.40  Joe <NA>          3      5  Joe
#> 25       3 70.5 4.53  0.58  6.90  Joe <NA>          3      5  Joe
#> 26       3 70.5 4.53  1.02  8.20  Joe <NA>          3      5  Joe
#> 27       3 70.5 4.53  2.02  7.80  Joe <NA>          3      5  Joe
#> 28       3 70.5 4.53  3.62  7.50  Joe <NA>          3      5  Joe
#> 29       3 70.5 4.53  5.08  6.20  Joe <NA>          3      5  Joe
#> 30       3 70.5 4.53  7.07  5.30  Joe <NA>          3      5  Joe
#> 31       3 70.5 4.53  9.00  4.90  Joe <NA>          3      5  Joe
#> 32       3 70.5 4.53 12.15  3.70  Joe <NA>          3      5  Joe
#> 33       3 70.5 4.53 24.17  1.05  Joe <NA>          3      5  Joe
```

There are a number of ways of controlling and examining factor levels

```
## generate data
x = factor(sample(letters[1:5],100, replace=TRUE))

print(levels(x))  ## This will show the levels of x are "Levels: a b c d e"
#> [1] "a" "b" "c" "d" "e"

## To reorder the levels:
## note, if x is not a factor use levels(factor(x))
x = factor(x,levels(x)[c(4,5,1:3)])

print(levels(x))  ## Now "Levels: d e a b c"
#> [1] "d" "e" "a" "b" "c"
```

4.10.0.1 Your task

- Using Theoph dataset, relevel the `Subject` column so the factor levels correspond to the labels

Chapter 5

Writing Functions

This section will be our bread and butter as functions provide the means for us to start to harness R's power to reduce duplication of code and increase our efficiency.

Functions in R are known as "first class objects" - that is, they can be treated like other R objects.

They can be:

- created without a name
- assigned to variables
- stored in lists
- returned from functions
- passed as parameters to other functions

Essentially, you can do anything with a function that you can with a vector.

In R, a function is defined with the following syntax:

`function(parameters) {body}`

- `function` is a reserved word to initialize creation.
- **parameters** are sets of formal parameter names that will be defined in the function body.
- **Formal parameters** are parameters included in the function definition
- The **body** is simply the code that the function will execute

A function can be written in one line as shown above, however, to encapsulate multiple lines brackets `{}` must be used.

A multi-line function could look as such:

```
function(parameters) {
    some code
    some more code
    even more code
}
```

To create a function for future use we must assign it to an object (a variable). For example, we can a simple addition function to examine some features

```
add_fun <- function(x, y) {
    x + y
}
```

This is a function declaration. We have created a function and given it a name. We can use it by calling it by name and passing some parameters that it requires.

```r
add_fun(1, 5)
#> [1] 6
```

There are numer of important behaviors going on 'behind-the-scenes' in even this simple function call.

5.0.1 Default Behaviors

Formal parameters can be soley user defined, they can also have a default value/behavior.

Defaults can be assigned to a parameter with `=`

Let's update our function to default to `y = 5`

```r
add_fun2 <- function(x, y = 5) x + y
```

When a default behavior defined, if no object or value is passed to that parameter, the default value is used.

```r
add_fun2(6)
#> [1] 11
```

But you can override the default behavior by simply passing in some value

```r
add_fun2(6, 3)
#> [1] 9
```

If no default is defined, the function will halt and give you an error requesting what to do with the missing parameter value

```r
add_fun2(y = 3)
#> Error in add_fun2(y = 3): argument "x" is missing, with no default
```

When you have multiple parameters - how does a function know which one to use for the various parameters?

5.0.2 Parameter Matching

Like all things programming - R has specific rules for how it handles parameter matching for functions.

Here is a basic overview:

- parameters can be matched positionally or by name
- you can mix positional and named matching
  - when an parameter is matched by name it is "removed" from the parameter list - the remaining parameters are matched by order
- parameters can be partially matched

The overall order of operations for parameter matching:

1) Check exact match for named parameter
2) Check for partial match for named parameter
3) Check for positional match
4) Any remaining unmatched formal parameters are "taken up" by `...`

Caveat(s)

- Any parameters *after* `...` are only matched exactly
- Tags partially matching multiple parameters will result in an error

5.0.3 Passing on parameters

`...` parameter indicates that parameters may be passed on to other (internally called) functions

`...` can be used when extending another function where you don't want to copy all the parameters from the original function.

```
f <- function(x,...) {
    print(x)
    summary(...)
}

f("It worked! The summary is:", runif(1000, 0, 100), digits=2)
#> [1] "It worked! The summary is:"
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>       0      24      52      51      76     100
```

As you can see, all parameters after the first - which was given to `x` - where passed to `summary`.

5.0.4 Return Values

In R after a function completes its code it will return a resulting value.

```
f <- function(x) x + 1
f(2)
#> [1] 3
```

`f(2)` returns the result of `x + 1`, which in this case is 3.

By default, **R returns the last evaluated expression**. You can also formally declare what you'd like R to return using `return()`

```
f <- function(x) return(x + 1)
```

This can be helpful for legibility when dealing with more complex functions where multiple outcomes are possible. It can also help you "escape" a function early by returning a result as soon as one is relevant

```
num_sign <- function(x) {
if (!is.numeric(x)) return("NaN")
if (x > 0) return("positive")
if (x <0) return("negative")
return("Don't know - is it zero?")
}

num_sign(1)
#> [1] "positive"
num_sign(-1)
#> [1] "negative"
num_sign("hello")
#> [1] "NaN"
num_sign(0)
#> [1] "Don't know - is it zero?"

num_sign2 <- function(x) {
if (!is.numeric(x)) "NaN"
if (x > 0) "positive"
if (x <0) "negative"
"Don't know - is it zero?"
}

num_sign2(1)
#> [1] "Don't know - is it zero?"
num_sign2(-1)
#> [1] "Don't know - is it zero?"
num_sign2("hello")
#> [1] "Don't know - is it zero?"
num_sign2(0)
#> [1] "Don't know - is it zero?"
```

One other nomenclature change to note in the above example, is if an `if` statement only contains a single statement on a single line, parentheses are not required.

R can only return a single result from a function

To return multiple objects you can combine them into a list or other structure

```
PK_info <- function(){
    id <- 1:10
    id
    doses <- c(1, 5, 10)
    doses
    time <- seq(0, 10, 1)
    time
}

PK_info2 <- function(){
    id <- 1:10
    doses <- c(1, 5, 10)
    time <- seq(0, 10, 1)
    list("id" = id, "doses" = doses, "time" = time)
}

PK_info() # only returns time
#>  [1]  0  1  2  3  4  5  6  7  8  9 10
PK_info2() # returns everything as a list
#> $id
#>  [1]  1  2  3  4  5  6  7  8  9 10
#> 
#> $doses
#> [1]  1  5 10
#> 
#> $time
#>  [1]  0  1  2  3  4  5  6  7  8  9 10
```

5.1 Types of Functions

3 specific types of functions that you may frequently run-into and/or utilize yourself are:

- Anonymous functions - functions that don't have a name
- Closures - functions written by other functions (will not discuss further)

- Lists of Functions - storing multiple functions in a list

5.1.1 Anonymous Functions

In R, there is no special syntax for creating functions. Functions, like most things in R, are objects themselves. When you create a function, you are simply assigning a name to the object you are creating. By this behavior you can even create a function and assign it many names.

Sometimes, however, we don't want or need to spend the time assigning a name. You've most likely run across this when reading code that uses commands such as the `apply` family, `do.call`, or with `plyr`.

Something along the lines of:

```
lapply(df, function(x) length(unique(x)))
```

This lapply command could be rewritten with a named function

```
len_unique <- function(x) length(unique(x))
lapply(df, len_unique)
```

However, that is unnecessarily verbose for a one-time function, and can also introduce unnecessary clutter into your environment(s)

Just like other functions, anonymous function have **formals** (parameters), a **body** and are tied to a **parent environment**

5.1.2 Lists of Functions

One way to store sets of functions is to put them in lists. Instead of a single function returning a list of results, you can actually store the functions themselves in a list for later re-use.

```
means <- list(
normal = function(x) mean(x),
geometric = function(x) ...,
harmonic = function(x) ...,
    )
```

To then call a function you can simply extract it from the list

`means$harmonic(data)` or `means[["geometric"]](data)`

While it might seem awkward there are some situations where lists of functions provide convenience. It also offers a degree of modularity that reduces dependencies.

5.1.2.1 Assignment

- Using a dataset of your choosing write 2-3 functions to create exploratory plots
- Add those functions to an `exploratory_plots` list.
- Using `lapply` quickly call all the functions on:
  - the whole dataset
  - a subset

Chapter 6

Functional Programming

Functions provide a host of benefits to the user. The allow for efficient automation of repetitive tasks, bundling or common operations and a host of other possibilities.

One additional, and equally important, opporunity they offer is to **reduce errors**.

As touched upon in *Pragmatic Programming*, the **DRY** (DON'T REPEAT YOURSELF) is well suited to functions.

A Motivating Example

When dealing with a new dataset there are two frequent issues that arise with concentration data - BQL values, and how to handle them.

Given the hypothetical situation where you are given 3 similarly-structured datasets how you could handle replacing the phrase `LLOQ < 10` simply with NA?

The copy-paste process may be something along the lines of:

```
df1$conc[df1$CONC == 'LLOQ < 10'] <- NA
df2$conc[df2$CONC == 'LLOQ < 10'] <- NA
df3$conc[df3$CONC == 'LLOQ < 10'] <- NA
df4$conc[df4$CONC == 'LLOQ < 10'] <- NA
df5$conc[df4$CONC == 'LLOQ < 10'] <- NA
```

Quick - did anyone spot any issue with the code?

Let's write a function to help automate this, as well as reduce potential for mistakes such as the above

```
BQL_NA <- function(x) {
    x[x[["CONC"]] == 'LLOQ < 10'] <- NA
    x
}

BQL_NA(df1)
BQL_NA(df2)
BQL_NA(df3)
BQL_NA(df4)
BQL_MA(df5)
```

Closer, but again, we're repeating ourselves.

We touched on `lapply`, lets combine our dataframes into a list a

SIDE TRICK `df[] <- lapply(df, our_fun)` - using `df[]` will give us back a dataframe instead of a list from `df <- lapply(df, our_fun)`

```
df_list <- list(df1, df2, df3, df4, df5)
df_NOBQL <- lapply(df_list, BQL_NA)
```

6.0.1 Assignment

- extend BQL_NA to allow us to pass different character strings for what how the LLOQ was defined
- BONUS: likewise, extend it to:
  - handle any column 'conc' regardless of capitalization
  - handle any concentration column name (conc, concentration, DV, ...)

Chapter 7

Control Structures

Control structures offer the ability to control how your code executes and what is returned. R was designed to abstract away many of the basic use-cases so you can get a long way without ever explicitly using some of these constructs, however, both the concepts and techniques are vital to appreciate as more complex problems arise.

7.1 IF statements

`if` statements are the most basic control structure, and have the structure and syntax:

```
if (<condition>) {
## do something
}
```

One basic use case is when doing clinical trial simulations where many reps are run, it can be useful to only occasionally print status updates. In pseudocode the objective would be to have code that looked like

```
for (rep in 1:50) {
  #every 10 reps print a status update
  if(rep %% 10 == 0) {
    # %% is the modulus operator, a nice way to detect every 10th rep
    print(paste("starting rep", rep))
  }
  # run some more code
}
#> [1] "starting rep 10"
#> [1] "starting rep 20"
#> [1] "starting rep 30"
#> [1] "starting rep 40"
#> [1] "starting rep 50"
```

Another common pattern is to handle certain scenarios in a function, again using print (since it is easy to demonstrate) for whether a function prints out a message before returning the values.

```
chatty_sum <- function(a, b, verbose=FALSE) {
  if(verbose) {
    print(paste("calculating the sum of a:",a, "and b:", b))
  }
  return(a+b)
}
chatty_sum(1, 3)
#> [1] 4

chatty_sum(2, 4, verbose=TRUE)
#> [1] "calculating the sum of a: 2 and b: 4"
#> [1] 6
```

The `chatty_sum` function also illustrates another concept about `if` statements, **they are inherently truthy**, which means that if the condition you are evaluating naturally resolves to be `TRUE` or `FALSE`, you do not need to explicitly evaluate it. In this case, an example is worth a thousand words:

Correct way

```
condition <- TRUE

if (condition) {

}
```

INCORRECT way

```
condition <- TRUE

if(condition ==TRUE) {
# do stuff
}
```

It is key to understand the inherent truthiness of **if** statements to further extend their usefulness. For example, if you need to execute the code inside the if statement when the tested condition is **FALSE** you can do-so using the `!` (often pronounced as 'bang') operator, which means *not*, so we are saying we want to run the code when it is **not FALSE**, which is equivalent to **TRUE**

```
good_condition <- FALSE
if(!good_condition) {
  print("bad news!")
}
#> [1] "bad news!"
```

7.2 ELSE

if statements are powerful to testing single conditions, however in may situations it is valuable to to also do something different if the condition is NOT met.

In **select** situations you may want to cover each of these scenarios via multiple if statements

```
if (<condition1>) ## do something
if (<condition2>) ## do something
if (<condition3>) ## do something
```

However this is generally frowned upon as it makes the code harder to make sense of, and easier to have unexpected conditions when a later condition is also TRUE, thereby overwriting a the results of a earlier TRUE condition.

Instead, the `else` block offers a separate chunk of code which will only run if the if condition is FALSE.

```
if (<condition>) {
## do something
} else {
## do something else
}
```

This can be further expanded to handle multiple conditions, however the value of this technique is only the code in the block with the *first* block that evaluates to TRUE will be run, and if no blocks are TRUE, then the code in the else block will run.

```
if(<condition>) {
    ## do stuff
} else if (<condititition2>) {
    ## do other stuff
    } else {
    ## do something for any other condition
}
```

7.3 ifelse

Full if/else blocks can be tedious to write out, and do not always play well with R's vectorization principles. Hence, a 'shortcut' block, `ifelse` is also available. `ifelse` is most likely going to be your mostly frequently used control statement used during the data analysis process so make sure it is well understood!

`ifelse` blocks are designed with the format `ifelse(<test condition>, <if yes>, <if no>)`

```
gender <- c("Male", "Male", "Female")

ismale <- ifelse(gender == "Male", 1, 0)

gender
#> [1] "Male"   "Male"   "Female"
ismale
#> [1] 1 1 0
```

In the above block, the test is saying, go through the **gender** vector, and for each element, test if that value is equal to "Male", then return a value of 1, and if it is **anything other than Male**, return 0. Since this is performed for each element separately, it will return a vector the same length as the tested vector.

You can even nest `ifelse` statements

```
race <- c("white", "black", "hispanic", "asian", "alien")

race_num <- ifelse(race == "white", 1,
                   ifelse(race == "black", 2,
                          ifelse(race == "hispanic", 3, 4)))
race
#> [1] "white"    "black"    "hispanic" "asian"    "alien"
race_num
#> [1] 1 2 3 4 4
```

Notice that in this technique if none if the `TRUE` conditions are met, it will be assigned the value `4`. This is a good practice to make sure you catch all possible values. The last false value should always be used to handle 'all other conditions'.

For example

```
race <- c("white", "black", "hispanic")

## bad way
race_num <- ifelse(race =="white", 1,
       ifelse(race == "black", 2,
              3)) # expect 3 to be hispanics
race_num
#> [1] 1 2 3
```

Works as expected, however if you accidentally miss a condition or other conditions are present

```
race <- c("white", "black", "hispanic", "asian", "alien")
race_num <- ifelse(race =="white", 1,
       ifelse(race == "black", 2, 3))
race_num
#> [1] 1 2 3 3 3
```

You can silently get unexpected assigned values.

One technique that can be used is to use the final condition as `-99` or some other very obvious flag to check that you captured all conditions

```
race <- c("white", "black", "hispanic", "asian", "alien")
race_num <- ifelse(race =="white", 1,
       ifelse(race == "black", 2,
              ifelse(race == "hispanic", 3, -99)))
race_num
#> [1]   1   2   3 -99 -99

if(any(race_num < 0)) {
  print("missed a condition")
} else {
  print("handled all")
}
#> [1] "missed a condition"
```

And it even makes it easy to subset out the conditions missed to find and correct them

```
race[race_num == -99]
#> [1] "asian" "alien"
```

see that missed alien and asian

```
race <- c("white", "black", "hispanic", "asian", "alien")
race_num <- ifelse(race =="white", 1,
       ifelse(race == "black", 2,
              ifelse(race == "hispanic", 3,
                     ifelse(race == "asian", 4,
                            ifelse(race == "alien", 5, -99)
                            )
                     )
              )
       )
race_num
#> [1] 1 2 3 4 5
```

```
if(any(race_num < 0)) {
  print("missed a condition")
} else {
  print("handled all")
}
#> [1] "handled all"
```

7.4 Loops

- `for` looping structure

Loops can be constructed based on a specified vector length or by specific indices

```
for (i in 1:5) {
    store_results[i] <- do_something()
}
```

While you may be more familiar with the construct:

```
for(i in 1:length(x)) {
     results[i] <- do_something(x[i])
}
```

This is actually a “bad” habit that can run you into trouble with objects of length(0)

`seq_along` is a ‘safer’ option that has the exact same effect if you’re starting from the first indice, with the added benefit of failing more gracefully.

- `next` - skip iteration of loop

`next` can be used to skip iterations in a loop, so as soon as a `next` is seen, the for loop moves to the next iteration and will ignore any code remaining from the existing iteration.

This is useful if you are checking a condition at the beginning and if that condition is met go on so you don’t run extra unnecessary code.

7.5 more For loop information

Given a for loop where certain elements you do not want anything to happen, the `next` keyword allows one to immediately go back to the top of the loop and start with the next indice.

```
for(i in 1:10) {
    if (i < 3) next
    print(i)
}
#> [1] 3
#> [1] 4
#> [1] 5
#> [1] 6
#> [1] 7
#> [1] 8
#> [1] 9
#> [1] 10
```

This is convenient if there is additional code below that you do not want R to run/evaluate given certain conditions are met, without having to resort to complex or nested `if` statments.

`break` break the execution of a loop. Unlike next, this will actually halt the loop completely and procede on to any later code after the loop.

```
## some code

for (i in 1:10) {
  start <- i
  print(paste0("before break, rep: " ,  i))
  if (i == 5) {
    break
  }
  print(paste0("after break, rep: " ,  i))
  finished <- i
}
#> [1] "before break, rep: 1"
#> [1] "after break, rep: 1"
#> [1] "before break, rep: 2"
#> [1] "after break, rep: 2"
#> [1] "before break, rep: 3"
#> [1] "after break, rep: 3"
#> [1] "before break, rep: 4"
#> [1] "after break, rep: 4"
#> [1] "before break, rep: 5"
```

In the above example, we see a conditional check where if `i == 5` break out of the loop. As the print statements show, only 4 values are printed for `after break`, as the last rep that executed code to the end of the loop was `4`, however you can see that the fifth replicate did start, and executed code up until the `break` statement.

Finally, in the context of a function, a for loop (or any other code) can be prematurely completed by using a `return` statement.

`return` - exit function

```
number_is_present <- function(nums, test_number) {
  for (i in seq_along(nums)) {
    if(i == test_number) {
      print("number found!")
      return(TRUE)
    }
  }
  print("completed scanning, number not found!")
  return(FALSE)
}

number_is_present(1:10, 5)
#> [1] "number found!"
#> [1] TRUE

number_is_present(1:10, 12)
#> [1] "completed scanning, number not found!"
#> [1] FALSE
```

In the example above, it makes sense to not continue scanning more numbers if we already know that the number is present, so by returning as soon as the number is detected, we prevent the need to continue running the function to completion.

7.6 While loops

- `while` - execute loop *while* tested condition is true. Often used if you need to evaluate until a certain condition is met. **danger**: can potentially result in infinite loops if not written properly or if the tested condition is never met.

```
count <- 0
while(count < 10) {
    print(count)
    count <- count + 1
}
#> [1] 0
#> [1] 1
#> [1] 2
#> [1] 3
#> [1] 4
#> [1] 5
#> [1] 6
#> [1] 7
#> [1] 8
#> [1] 9
```

While loops are frankly not used often for data analysis tasks. In most cases, a for-loop, the in-built apply functions, or using vectorization is be preferable. While loops can be valuable for instances when you are unsure about when the final criteria required will be satisfied. For example, given a resampling function that you must resample from a mathematical distribution, however want to apply some other physiological constraints such that some samples might need to be resampled again before being stored.

A simple example of this is, given simulations from a truncated normal distribution, where only values greater than 0.1 should be kept.

A couple different ways this could be mentally managed

```
# while loop inside a for loop
result <- c()
for (i in 1:10) {
  sample <- 0
  while (sample < 0.1) {
    ## will always fire the first time per rep since at first sample will be 0
    print(paste0("sample less than 0.1, resampling on rep", i))
    sample <- rnorm(1, mean = 0.5, sd = 2)
  }
  result <- c(result, sample)
}
#> [1] "sample less than 0.1, resampling on rep1"
#> [1] "sample less than 0.1, resampling on rep1"
#> [1] "sample less than 0.1, resampling on rep2"
#> [1] "sample less than 0.1, resampling on rep2"
#> [1] "sample less than 0.1, resampling on rep3"
#> [1] "sample less than 0.1, resampling on rep4"
#> [1] "sample less than 0.1, resampling on rep5"
#> [1] "sample less than 0.1, resampling on rep5"
#> [1] "sample less than 0.1, resampling on rep5"
#> [1] "sample less than 0.1, resampling on rep5"
#> [1] "sample less than 0.1, resampling on rep5"
#> [1] "sample less than 0.1, resampling on rep5"
#> [1] "sample less than 0.1, resampling on rep6"
#> [1] "sample less than 0.1, resampling on rep7"
#> [1] "sample less than 0.1, resampling on rep7"
#> [1] "sample less than 0.1, resampling on rep8"
#> [1] "sample less than 0.1, resampling on rep8"
#> [1] "sample less than 0.1, resampling on rep8"
#> [1] "sample less than 0.1, resampling on rep9"
#> [1] "sample less than 0.1, resampling on rep10"
#> [1] "sample less than 0.1, resampling on rep10"

# in the end get our nice set of 10 results
result
#>  [1] 1.011 0.489 1.743 2.797 1.758 4.630 1.525 0.395 1.586 1.436
```

The problem with this result is it reallocates a new vector each time a new sample is concatenated, so for large numbers of samples will be quite slow. Instead, we can also manage via vectorization. Since we are going to do all the calculations at once, we need to change our logic in how we are resampling.

```
result <- rnorm(10, 0.5, 2)
while(any(result < 0.1)) {
  print("at least 1 result still below 0.1, resampling those values")
  # find which indices correspond to values less than 0.1
  to_low <- which(result < 0.1)
  # replace them with resampled values
  result[to_low] <- rnorm(length(to_low), 0.5, 2)
}
#> [1] "at least 1 result still below 0.1, resampling those values"

result
#>  [1] 1.226 0.853 1.976 4.277 0.305 0.987 0.468 3.747 0.724 2.371
```

```
looped_resampling <- function(total_num = 100) {
  result <- c()
  for (i in 1:total_num) {
    sample <- 0
    while (sample < 0.1) {
      ## will always fire the first time per rep since at first sample will be 0
      message(paste0("sample less than 0.1, resampling on rep", i))
      sample <- rnorm(1, mean = 0.5, sd = 2)
    }
    result <- c(result, sample)
  }
  result
}

vectorized_resampling <- function(total_num = 100) {
  result <- rnorm(10, 0.5, 2)
  while(any(result < 0.1)) {
    message("at least 1 result still below 0.1, resampling those values")
    # find which indices correspond to values less than 0.1
    to_low <- which(result < 0.1)
    # replace them with resampled values
    result[to_low] <- rnorm(length(to_low), 0.5, 2)
  }
  result
}
```

```
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
library(tibble)
#>
#> Attaching package: 'tibble'
#> The following objects are masked from 'package:dplyr':
#>
#>     as_data_frame, data_frame, data_frame_, frame_data, glimpse,
#>     knit_print.trunc_mat, tbl_df, tibble, trunc_mat, type_sum
res <- suppressMessages(microbenchmark::microbenchmark(
  looped_resampling(),
  vectorized_resampling(),
  times = 20L
))
res_df <- data_frame(expr = res$expr, timing = res$time)

res_df %>% group_by(expr) %>%
  summarize(min = min(timing),
            mean = mean(timing),
            max = max(timing))
#> Source: local data frame [2 x 4]
#>
#>                      expr      min     mean      max
#>                    <fctr>    <dbl>    <dbl>    <dbl>
#> 1     looped_resampling() 34743114 41443430 47413149
#> 2 vectorized_resampling()   231060   884050  1694449
ggplot2::autoplot(res)
```

7.7 Assignments

- within theoph add a new column called `ISEVEN` - for all subjucts with even ID numbers assign a value 1, for all odds assign value 0
- write a creatinine clearance calculator function - use it to calculate the CRCL for some simulated data in a vectorized manner

Chapter 8

Apply functions

There are a number of `apply` commands in the R family. Each has different way of handling data

Additional information can be found in threads on stackoverflow

Note - the apply-style commands are *not* particularly faster than a *well-written* loop. The benefit is their one-line nature. Read some more here

8.1 apply

Used to evaluate a function over the margins of a matrix/array. It can apply a function by dimension (row/col).

`apply` takes the following arguments: * `X` - input * `MARGIN` - to distinguish by row or column. `1` - row, `2` - column * `FUN` - function to be applied * `...` is for other arguments to be passed into fun

For example to apply the function `range` across all columns in `Theoph`:

```
apply(Theoph, 2, range)
#>      Subject Wt     Dose   Time    conc
#> [1,] "1"     "54.6" "3.10" " 0.00" " 0.00"
#> [2,] "9"     "86.4" "5.86" "24.65" "11.40"
str(apply(Theoph, 2, range))
#>  chr [1:2, 1:5] "1" "9" "54.6" "86.4" "3.10" "5.86" ...
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : NULL
#>   ..$ : chr [1:5] "Subject" "Wt" "Dose" "Time" ...
```

Notice: - the 'range' function does not have '()' around it. - The results are all characters - since `apply` goes over by dimension and needs a matrix/array, the whole data structure is coerced to the highest level. Since Subject is an ordered factor for `Theoph` the whole data-frame is coerced to a matrix of type `char` before the function is applied!

tidbit - apply is a good opportunity to use colMeans/colSums and rowMeans/rowSums - they are significantly faster, especially for large data structures.

8.2 lapply

Works to apply a function to each element in the list and *returns a list back* `lapply` takes the following arguments: * `X` - input * `FUN` - function to be applied * `...` is for other arguments to be passed into fun

```
lapply(Theoph, range)
#> $Subject
#> [1] 6 5
#> Levels: 6 < 7 < 8 < 11 < 3 < 2 < 4 < 9 < 12 < 10 < 1 < 5
#>
#> $Wt
#> [1] 54.6 86.4
#>
#> $Dose
#> [1] 3.10 5.86
#>
#> $Time
#> [1]  0.0 24.6
#>
#> $conc
#> [1]  0.0 11.4
```

Given that a dataframe is a list with dimensions - lapply can easily step throuh the columns and apply your function and returns a list.

8.3 sapply

sapply is similar to lapply, the biggest difference is it attempts to simplify the result.

```
lapply(Theoph, range)
#> $Subject
#> [1] 6 5
#> Levels: 6 < 7 < 8 < 11 < 3 < 2 < 4 < 9 < 12 < 10 < 1 < 5
#>
#> $Wt
#> [1] 54.6 86.4
#>
#> $Dose
#> [1] 3.10 5.86
#>
#> $Time
#> [1]  0.0 24.6
#>
#> $conc
#> [1]  0.0 11.4
sapply(Theoph, range)
#>      Subject   Wt Dose Time conc
#> [1,]       1 54.6 3.10  0.0  0.0
#> [2,]      12 86.4 5.86 24.6 11.4
str(lapply(Theoph, range))
#> List of 5
#>  $ Subject: Ord.factor w/ 12 levels "6"<"7"<"8"<"11"<..: 1 12
#>  $ Wt     : num [1:2] 54.6 86.4
#>  $ Dose   : num [1:2] 3.1 5.86
#>  $ Time   : num [1:2] 0 24.6
#>  $ conc   : num [1:2] 0 11.4
```

```
str(sapply(Theoph, range))
#>  num [1:2, 1:5] 1 12 54.6 86.4 3.1 ...
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : NULL
#>   ..$ : chr [1:5] "Subject" "Wt" "Dose" "Time" ...
class(lapply(Theoph, range))
#> [1] "list"
class(sapply(Theoph, range))
#> [1] "matrix"
typeof(lapply(Theoph, range))
#> [1] "list"
typeof(sapply(Theoph, range))
#> [1] "double"
```

Generally, the “rules” for `sapply` can be thought-of as: * Result is a list where every element is length 1 - returns a vector * Result list where every elemtn is same length - returns a matrix * Neither of above - returns list

8.4 vapply

preferred over sapply due to minor speed improvement and better consistency with return types. Can read more about it on stack overflow here

The way `vapply` works is you also specific implicitly the type of value you expect it to return.

`vapply(<values>, <function>, <returntype>)`

```
sapply(Theoph[c("Wt", "conc")], summary)
#>           Wt  conc
#> Min.    54.6  0.00
#> 1st Qu. 63.6  2.88
#> Median  70.5  5.28
#> Mean    69.6  4.96
#> 3rd Qu. 74.4  7.14
#> Max.    86.4 11.40

vapply(Theoph[c("Wt", "conc")], summary, c(Min. = 0, "1st Qu." = 0, Median = 0, mean = 0,  "3rd Qu." =
#>           Wt  conc
#> Min.    54.6  0.00
#> 1st Qu. 63.6  2.88
#> Median  70.5  5.28
#> mean    69.6  4.96
#> 3rd Qu. 74.4  7.14
#> Max.    86.4 11.40
```

But unlike sapply, which will give back whatever, vapply will only work if explicity get back what you ask for.

```
sapply(Theoph, summary)
vapply(Theoph, summary, c(Min. = 0, "1st Qu." = 0, Median = 0, mean = 0,  "3rd Qu." = 0, Max. = 0))
```

A common pattern if you just expect a character value, or logical value etc. is to use `character(1)`

```
sapply(Theoph[, c("Wt", "conc", "Time")], class)
#>        Wt      conc      Time
#> "numeric" "numeric" "numeric"
```

```
vapply(Theoph[, c("Wt", "conc", "Time")], class, character(1))
#>        Wt      conc      Time
#> "numeric" "numeric" "numeric"
```

Again, vapply will catch so wouldn't run into unexpected output. This can be key when you are testing conditions (eg if class == "") where the test conditions are very specific, so things like passing a 2 element vector for the class would cause the whole function to error. It is better to catch it early than for things to blow up unexpectedly.

```
sapply(Theoph, class)
vapply(Theoph, class, character(1))
```

8.5 tapply

Used to apply functions to subsets of a vector. I will briefly give some detail, though my person preference is to use `dplyr` commands with `dplyr::group_by` when I'm dealing with subsets. It is good to know, however, in case you are unable to install dplyr for whatever reason.

`tapply` takes the following arguments: * `X` - input * `INDEX` is a factor or list of factors * `FUN` - function to be applied * `...` is for other arguments to be passed into fun * `simplify` - logical for whether to simplify result or not

note: INDEX must be a factor or list of factors - if not any value passed in will be coerced to factor.

```
with(Theoph, tapply(conc, Wt, mean))
#> 54.6 58.2 60.5 64.6   65 70.5 72.4 72.7 79.6   80 86.4
#> 5.78 5.93 5.41 3.91 4.51 4.68 4.82 4.94 6.44 3.53 4.89
```

Without using `with`:

```
tapply(Theoph$conc, Theoph$Wt, mean)
#> 54.6 58.2 60.5 64.6   65 70.5 72.4 72.7 79.6   80 86.4
#> 5.78 5.93 5.41 3.91 4.51 4.68 4.82 4.94 6.44 3.53 4.89
```

Chapter 9

Thinking in R

Thinking in R is very important as your R-tasks increase in complexity or magnitude. Keep in mind that the rest of the workshop will build and call on these concepts heavily. This will also be your first foray into examining your approach to a problem and 'best practices', as there will almost always be a number of ways of getting to the result you want. The goal is to keep you in the 'better' to 'best' range as often as possible.

9.1 Memory Allocation

Before we jump into subsetting, we need to briefly touch upon how R treats data, and more importantly, how it *grows* it as you use it.

Lets look at three possible methods of generating a sequence:

```
library(microbenchmark)
n <- 10000
# c_grow
c_grow <- function() {
vec <- numeric(0)
for(i in 1:n) vec <- c(vec, i)
}

# subscript
ss_grow <- function() {
vec <- numeric(n)
for(i in 1:n) vec[i] <- i
}

# colon
colon_grow <- function() vec <- 1:n

microbenchmark(
 c_grow(),
  ss_grow(),
  colon_grow(),
  times=10L
)
```

Why is the colon operator so much faster?

One reason is that ':' is actually a function just like sum() or mean() or ggplot() - but it is written in `C` and written for *speed*.

Second, When you grow an object incrementally, it can lead to an issue called **fragmented memory** - when R starts to store the new elements and runs out of the currently allocated space it has to hunt for new space. Do this over and over and it wastes a lot of time.

One stark example is the use of rbind to continuously append to a dataframe.

There are two ways to avoid this trap.

1. Build a list of the pieces and combine in one go (using do.call - will be covered later)
2. preallocate the memory (will also be covered more later)

Eliminating the growth or re-indexing of objects is an easy way of dramatically increasing the speed of R code.

9.2 Vectorization

I'm sure you've heard the concept that `for` loops in R are slow. Well slow compared to what?

Let's consider how we could add together a vector of numbers:

```
lsum <- 0
for (i in 1:length(x)) lsum <- lsum + x[i]
```

or the easy R-way

```
 sum(x)
```

It may seem like a silly example to you because `sum(x)` is so intuitive to the average user - but someone had to write the function at some point.

Similar to the `:` example previously, these examples provide the same result but with dramatic differences in speed.

The magic that gives these functions is they have been written in C. They are still loops - that is unavoidable, but they've been optimized for speed.

The other benefit that is frequent by-product of vectorization is *legibility* - sum(x) is so easy to understand compared to that loop.

Likewise, `conc <- amt*volume` is both easier to read, and clearly expresses the relationship between variables, whether we are multiplying one amt and volume or a columns of amts and volumes.

9.3 Vectorization Tips

- Put as much outside of loops as possible. Ex: If you have a sequence you are applying over and over, create the sequence first and reuse it inside the loop.
- Make the number of iterations as small as possible.
- Don't feel the need to over-vectorize

We will come back to more 'optimization'-type issues later, but for now keep these principles in the back of your head as we go forward.

9.4 Indexing

As has been applied in previous examples, R has to have a way of referring to where in your object certain pieces of information are stored. Given that everything in R can be thought of as a vector, indexing allows us to easily query the position in which the vector an element is stored.

For example, to examine the 10th element of an atomic vector one simply can do `v[10]`

For more complex structures keep in mind this concept of an *index* for when you're thinking how to obtain or subset your data.

Chapter 10

Common Commands

This section is essentially a general compendium of key terms that are important in R

Much of this section is courtosy of Hadley Wickham - however it is slightly condensed for our uses.

If you have a conceptual idea in your mind about some task, often looking through the vocabulary for that section can help spark ideas.

For example, if you are trying to think of a way to subset to one observation per individual you could go to the data structures and see the `duplicated` option.

10.1 Vocabulary

An important part of being a competent R programmer is having a good working vocabulary. Below, I have listed the functions that I believe consistute such a vocabulary. I don't expect you to be intimately familiar with the details of every function, but you should at least be aware that they all exist.

I came up with this list by looking through all functions in `base`, `stats`, and `utils`, and extracting those that I think are most useful. The list also includes a few pointers to particularly important functions in other packages, and some of the more important options.

10.2 The basics

```
# The first functions to learn
?
str

# Important operators and assignment
%in%, match
=, <-, <<-
$, [, [[, head, tail, subset
with
assign, get

# Comparison
all.equal, identical
!=, ==, >, >=, <, <=
is.na, complete.cases
is.finite

# Basic math
*, +, -, /, ^, %%, %/%
abs, sign
acos, asin, atan, atan2
sin, cos, tan
ceiling, floor, round, trunc, signif
exp, log, log10, log2, sqrt

max, min, prod, sum
cummax, cummin, cumprod, cumsum, diff
pmax, pmin
range
mean, median, cor, sd, var
rle

# Functions
function
missing
on.exit
return, invisible

# Logical & sets
&, |, !, xor
all, any
intersect, union, setdiff, setequal
which

# Vectors and matrices
c, matrix
# automatic coercion rules character > numeric > logical
length, dim, ncol, nrow
cbind, rbind
names, colnames, rownames


# Making vectors
c
rep, rep_len
seq, seq_len, seq_along
rev
sample
choose, factorial, combn
(is/as).(character/numeric/logical/...)

# Lists & data.frames
list, unlist
data.frame, as.data.frame
split
expand.grid

# Control flow
if, &&, || (short circuiting)
```

```
for, while
next, break
switch
ifelse
```

10.3 Common data structures

```
# Date time
library(lubridate)

# Character manipulation
grep, agrep
gsub
strsplit
chartr
nchar
tolower, toupper
substr
paste
library(stringr)

# Factors
factor, levels, nlevels
reorder, relevel
cut, findInterval
interaction
options(stringsAsFactors = FALSE)

# Ordering and tabulating
duplicated, unique
merge
order, rank, quantile
sort
table, ftable
```

10.4 Statistics

```
# Linear models
fitted, predict, resid, rstandard
lm, glm
hat, influence.measures
logLik, df, deviance
formula, ~, I
anova, coef, confint, vcov
contrasts

# Miscellaneous tests
apropos("\\.test$")

# Random variables
(q, p, d, r) * (beta, binom, cauchy, chisq, exp, f, gamma, geom,
  hyper, lnorm, logis, multinom, nbinom, norm, pois, signrank, t,
  unif, weibull, wilcox, birthday, tukey)
```

10.5 Working with R

```
# Workspace
ls, exists, rm
getwd, setwd
q
source
install.packages, library, require

# Help
help, ?
help.search
apropos
RSiteSearch
citation
demo
example
vignette

# Debugging
traceback
browser
recover
options(error = )
stop, warning, message
tryCatch, try
```

10.6 I/O

```
# Output
print, cat
message, warning
dput
format
sink, capture.output

# Reading and writing data
data
count.fields
read.csv, write.csv,
read.delim, write.delim
read.fwf
readLines, writeLines
readRDS, saveRDS
load, save
library(foreign)

# Files and directories
```

```
dir
basename, dirname, tools::file_ext
file.path
path.expand, normalizePath
file.choose
file.copy, file.create, file.remove, file.rename, dir.create
file.exists, file.info
tempdir, tempfile
download.file, library(downloader)
```

Chapter 11

Input/Output

11.1 Fast I/O

To preface this initial section, if you are working in a locked version of R, due to company policy, you may skip to the built-in section below.

Two packages provide much of the general I/O funtionality necessary:

1) `readr`
2) `data.table`

In addition, `PKPDmisc` offers wrappers that provide tailored input/output:

1) read_nonmem()
2) write_nonmem()
3) read_phx() # for phx nlme datasets with a units column

11.2 Built-in I/O

There are two primary reasons to introduce the built-in I/O functions as well:

1) `data.table` or `readr` will not always be available on your/collegues computers
2) the built in functions are designed to be incredibly robust, therefore will often be able to handle edge cases that `readr` and `data.table::fread` fail on.

The principal way of reading/writing data in R `read.table` and `write.table`. Additionally, some "custom" functions with commonly used defaults are also available.

- `read.csv` is simply `read.table` with the default separator set to `,` and a couple other slight modifications

Another useful function is `source`. `source` allows us to read and execute R files automatically.

The `read.table` function has some important arguments we can go through for later reference (note all the arguments are obviously the same for `read.csv`)

- `file` - name of file
- `header` - indicates if file has header line
- `sep` - how columns are separated
- `colClasses` - character vector with class of each column
- `nrows` - number of rows
- `comment.char` - character string to indicate the comment character

- `stringsAsFactors` - how to handle strings (factor or simply strings)
- `skip` - how many lines to skip before starting to read Lets also look at a couple relevant defaults for `read.table` and `read.csv`

```
?read.table
```

- unless specificied path is relative to current working directory
- header = FALSE
- stringsAsFactors = TRUE
- R will skip lines that begin with a # (comment.char = "#")

```
?read.csv
```

- header = TRUE
- sep = ","
- quote = """
- dec = "."
- fill = TRUE
- comment.char = ""

11.3 Some tidbits

Assigning `colClasses` will significantly speed up the read.table process - this can be valuable for highly rich data.

A quick and dirty way of allowing R to detect and assign them for you is via:

```
initial <- read.table("data.txt", nrows = 50)
classes <- sapply(initial, class)
all_data <- read.table("data.txt", colclasses = classes)
```

11.4 Output

Many of the output settings are similar to input settings. Again, your best bet is to spend a little bit of time checking out the documentation.

A couple values that are relevant and frequently useful:

- `quote = FALSE` - will keep column names from being output with quotes around them
- `na` - allows you to declare the string to use for missing values in data
- `row.names = FALSE` - don't output additional (any usually unnecessary for us) row names column
- `append = TRUE` - rather than overwriting a file w/ that name - it appends the results to the end of the file

cookbook-R on writing text and output to files

11.4.1 Your task

- you are given a dataframe with a header and units column
- create list with the column names and units that you can reference as necessary during analysis
- are there other ways you could store units to access them later? attributes? What are the risks?
- BONUS: write a new function write_phx that has both the header column as well as a units column below to make importing to phoenix and setting units easier. Write it such that you would not lose the type of each column when reading back into R

Chapter 12

Custom I/O

As an extension of the basic I/O, it is worth mentioning one custom method for I/O, processing connections/strings directly. The value is sometimes data is either so messy, or very large but in a format that `readr` and `data.table` won't handle, so you must default to the default slow methods. This technique allows one to manually parse output and clean it up so it is useable by read_csv or another parser. For example, nonmem simulation tables give the following output when simulating with multiple NSUB

```
TABLE NO. 1
 REP ID TIME DV ...
  1  1  0.0  . ...
  1  1  0.25 0.5 ...
  ...
TABLE NO. 1
 REP ID TIME DV ...
  2  1  0.0  . ...
  2  1  0.25 0.3 ...
```

Chapter 13

Pragmatic Programming

Uwe Ligges maxim:

Computers are cheap, and thinking hurts

"Make your functions as simple as possible. Simple has many advantages: Simple functions are likely to be human efficient: they will be easy to understand and to modify. Simple functions are likely to be computer efficient. Simple functions are less likely to be buggy , and bugs will be easier to fix. (Perhaps ironically) simple functions may b e more general, thinking about the heart of the matter often broadens the application." (Burns 2011:33)

"We want our functions to be correct. Not all functions are correct. The results from specific calls can be put in to 4 categories: 1. Correct. 2. An error occurs that is clearly idenitifed. 3. An obscure error o ccurs. 4. An incorrect value is returned." (Burns 2011:35)

Solve the Problem Here are four steps for general problem solving (though of course we have programming in mind).

1. List the starting ingredients
2. State the desired results
3. Break the journey from step 1 to step 2 into subproblems
4. Put the subproblem solutions together

This is a recursive algorithm — we do the same four steps on each of the subproblems, and on their subproblems.

The hard part is step 3 (but sometimes step 2 is murky). This is another case of abstraction, of carving up reality. There may be multiple possible combinations of subproblems — your task is to create a reasonable combination. Avoid feeling that you must solve the whole thing in one go. That's the recipe for being overwhelmed and stuck.

Our natural inclination is to go from beginning to end when breaking up a problem. If that is not bearing fruit, try working backwards from the end. The more you practice breaking a problem into subproblems, the better you get at it. This is important enough to practice deliberately. This section should be surrounded by flashing lights. Breaking a problem into pieces seems to be the central block with programming for most people. Great acts are done by a series of small deeds

We always want to jump in and attack problems directly. However, the real problem is often not what we first think. This is especially true when we're solving someone else's problem. It is ever so common for someone to ask for some specific technical thing which turns out not to address their actual concern, or to be a poor approach to it. Before you dig into a problem, make sure you know what the problem is.

I count three possible outcomes:

- the real problem is easier than the original — you have saved time and trauma
- the real problem is harder than the original — but you've avoided working on the wrong one
- the original problem is the real problem — but now you can tackle it with confidence