

R 与 tidyverse——数据分析入门

石天熠

2019-07-02

Contents

欢迎	5
1 R 和 RStudio 介绍和安装教程	9
1.1 什么是 R	9
1.2 安装 R 和 RStudio	9
1.3 为什么使用 R, R 与其他统计软件的比较	10
2 获取资源与帮助 (重要!)	11
2.1 论坛类 (解答实际操作中的问题)	11
2.2 Reference 类 (查找特定的 function/package 的用法, 就像查字典一样)	11
2.3 教程和书籍类 (用来系统地学习)	12
2.4 速查表 (Cheat sheets) (用来贴墙上)	12
3 RStudio 界面介绍, 基本操作, 和创建新项目	13
3.1 界面	14
3.2 基本操作	15
4 安装和使用 packages (包)	19
4.1 Package 是什么, 为什么使用它们?	19
4.2 如何安装 packages	19
4.3 如何使用 packages	20
4.4 其它	20
5 向量, 逻辑, 循环和函数	23
5.1 向量的概念, 操作和优越性	23
5.2 数据类型 (Data Types)	28
5.3 数学表达和运算	28
5.4 逻辑	32
5.5 判断和循环 (控制流程)	34
5.6 函数	36
5.7 简易的统计学计算	37
6 dataframe (数据框) 和 tibble	45
6.1 查看 dataframe/tibble 并了解它们的结构	45

6.2	tibble 的创建和基础操作	47
6.3	进阶内容: list, array 和 matrix	51
7	使用 ggplot 绘图	59
7.1	哲理	60
7.2	基础	60
7.3	进阶	60
7.4	附: Base R 中的作图	60
8	数据处理	61
8.1	把 “untidy data” 整成 “tidy data”	61
8.2	数据的导入和导出	61
9	与 Python 的联合使用	63
9.1	在 R 中使用 Python: <code>reticulate</code>	63
9.2	在 Python 中使用 R: <code>rpy</code>	63
9.3	Beaker Notebook	63
	Bibliography	65

欢迎

简介

本书为 R 和 tidyverse 的入门向教程。教学视频在 b 站 O。

使用说明

左上角的菜单可以选择收起/展开目录，搜索，和外观，字体调整。

如果你对某一段文字有修改意见，可以选择那段文字，并通过 Hypothesis 留言（选择“annotate”）。右上角可以展开显示公开的留言。

如果你熟悉Bookdown和 Github，可以在此提交 pull request.






Colophon

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

This work is licensed under a Creative Commons “Attribution-NonCommercial-NoDerivatives 4.0 International” license.



本作品采用知识共享署名-非商业性使用-禁止演绎 4.0 国际许可协议进行许可。

本书使用 R Markdown (<http://rmarkdown.rstudio.com/>) RStudio (<http://www.rstudio.com/ide/>) knitr (<http://yihui.name/knitr/>) 和 pandoc (<https://pandoc.org/>) Rmd html pdf

```
devtools::session_info(c("tidyverse"))
```

```
## - Session info -----  
## setting value  
## version R version 3.5.3 (2019-03-11)  
## os      macOS 10.15  
## system x86_64, darwin15.6.0
```

```
## ui          X11
## language    (EN)
## collate     en_GB.UTF-8
## ctype       en_GB.UTF-8
## tz          Asia/Shanghai
## date        2019-07-02
##
```

- Packages -----

## package	* version	date	lib	source
## askpass	1.1	2019-01-13	[2]	CRAN (R 3.5.2)
## assertthat	0.2.1	2019-03-21	[2]	CRAN (R 3.5.2)
## backports	1.1.4	2019-04-10	[2]	CRAN (R 3.5.2)
## base64enc	0.1-3	2015-07-28	[2]	CRAN (R 3.5.0)
## BH	1.69.0-1	2019-01-07	[2]	CRAN (R 3.5.2)
## broom	0.5.2	2019-04-07	[2]	CRAN (R 3.5.2)
## callr	3.2.0	2019-03-15	[2]	CRAN (R 3.5.1)
## cellranger	1.1.0	2016-07-27	[2]	CRAN (R 3.5.0)
## cli	1.1.0	2019-03-19	[2]	CRAN (R 3.5.2)
## clipr	0.6.0	2019-04-15	[2]	CRAN (R 3.5.2)
## colorspace	1.4-1	2019-03-18	[2]	CRAN (R 3.5.2)
## crayon	1.3.4	2017-09-16	[2]	CRAN (R 3.5.0)
## curl	3.3	2019-01-10	[2]	CRAN (R 3.5.2)
## DBI	1.0.0	2018-05-02	[2]	CRAN (R 3.5.0)
## dbplyr	1.4.2	2019-06-17	[2]	CRAN (R 3.5.3)
## digest	0.6.19	2019-05-20	[2]	CRAN (R 3.5.2)
## dplyr	* 0.8.2	2019-06-29	[2]	CRAN (R 3.5.2)
## ellipsis	0.2.0	2019-06-20	[2]	CRAN (R 3.5.2)
## evaluate	0.14	2019-05-28	[2]	CRAN (R 3.5.2)
## fansi	0.4.0	2018-10-05	[2]	CRAN (R 3.5.0)
## forcats	* 0.4.0	2019-02-17	[2]	CRAN (R 3.5.2)
## fs	1.3.1	2019-05-06	[2]	CRAN (R 3.5.2)
## generics	0.0.2	2018-11-29	[2]	CRAN (R 3.5.0)
## ggplot2	* 3.2.0	2019-06-16	[2]	CRAN (R 3.5.2)
## glue	1.3.1	2019-03-12	[2]	CRAN (R 3.5.2)
## gtable	0.3.0	2019-03-25	[2]	CRAN (R 3.5.2)
## haven	2.1.0	2019-02-19	[2]	CRAN (R 3.5.2)
## highr	0.8	2019-03-20	[2]	CRAN (R 3.5.2)
## hms	0.4.2	2018-03-10	[2]	CRAN (R 3.5.0)
## htmltools	0.3.6	2017-04-28	[2]	CRAN (R 3.5.0)
## httr	1.4.0	2018-12-11	[2]	CRAN (R 3.5.0)
## jsonlite	1.6	2018-12-07	[2]	CRAN (R 3.5.0)
## knitr	1.23	2019-05-18	[2]	CRAN (R 3.5.2)
## labeling	0.3	2014-08-23	[2]	CRAN (R 3.5.0)
## lattice	0.20-38	2018-11-04	[2]	CRAN (R 3.5.3)
## lazyeval	0.2.2	2019-03-15	[2]	CRAN (R 3.5.2)
## lubridate	1.7.4	2018-04-11	[2]	CRAN (R 3.5.0)

##	magrittr	1.5	2014-11-22	[2]	CRAN	(R 3.5.0)
##	markdown	1.0	2019-06-07	[2]	CRAN	(R 3.5.2)
##	MASS	7.3-51.4	2019-03-31	[2]	CRAN	(R 3.5.2)
##	Matrix	1.2-17	2019-03-22	[2]	CRAN	(R 3.5.2)
##	mgcv	1.8-28	2019-03-21	[1]	CRAN	(R 3.5.2)
##	mime	0.7	2019-06-11	[2]	CRAN	(R 3.5.2)
##	modelr	0.1.4	2019-02-18	[2]	CRAN	(R 3.5.2)
##	munsell	0.5.0	2018-06-12	[2]	CRAN	(R 3.5.0)
##	nlme	3.1-140	2019-05-12	[1]	CRAN	(R 3.5.2)
##	openssl	1.4	2019-05-31	[2]	CRAN	(R 3.5.2)
##	pillar	1.4.2	2019-06-29	[2]	CRAN	(R 3.5.2)
##	pkgconfig	2.0.2	2018-08-16	[2]	CRAN	(R 3.5.0)
##	plogr	0.2.0	2018-03-25	[2]	CRAN	(R 3.5.0)
##	plyr	1.8.4	2016-06-08	[2]	CRAN	(R 3.5.0)
##	prettyunits	1.0.2	2015-07-13	[2]	CRAN	(R 3.5.0)
##	processx	3.3.1	2019-05-08	[2]	CRAN	(R 3.5.2)
##	progress	1.2.2	2019-05-16	[2]	CRAN	(R 3.5.2)
##	ps	1.3.0	2018-12-21	[2]	CRAN	(R 3.5.0)
##	purrr	* 0.3.2	2019-03-15	[2]	CRAN	(R 3.5.2)
##	R6	2.4.0	2019-02-14	[2]	CRAN	(R 3.5.2)
##	RColorBrewer	1.1-2	2014-12-07	[1]	CRAN	(R 3.5.0)
##	Rcpp	1.0.1	2019-03-17	[2]	CRAN	(R 3.5.2)
##	readr	* 1.3.1	2018-12-21	[2]	CRAN	(R 3.5.0)
##	readxl	1.3.1	2019-03-13	[2]	CRAN	(R 3.5.2)
##	rematch	1.0.1	2016-04-21	[2]	CRAN	(R 3.5.0)
##	reprex	0.3.0	2019-05-16	[2]	CRAN	(R 3.5.2)
##	reshape2	1.4.3	2017-12-11	[2]	CRAN	(R 3.5.0)
##	rlang	0.4.0	2019-06-25	[2]	CRAN	(R 3.5.2)
##	rmarkdown	1.13	2019-05-22	[2]	CRAN	(R 3.5.2)
##	rstudioapi	0.10	2019-03-19	[2]	CRAN	(R 3.5.2)
##	rvest	0.3.4	2019-05-15	[2]	CRAN	(R 3.5.2)
##	scales	1.0.0	2018-08-09	[2]	CRAN	(R 3.5.0)
##	selectr	0.4-1	2018-04-06	[2]	CRAN	(R 3.5.0)
##	stringi	1.4.3	2019-03-12	[2]	CRAN	(R 3.5.2)
##	stringr	* 1.4.0	2019-02-10	[2]	CRAN	(R 3.5.2)
##	sys	3.2	2019-04-23	[2]	CRAN	(R 3.5.2)
##	tibble	* 2.1.3	2019-06-06	[2]	CRAN	(R 3.5.3)
##	tidyr	* 0.8.3	2019-03-01	[2]	CRAN	(R 3.5.2)
##	tidyselect	0.2.5	2018-10-11	[2]	CRAN	(R 3.5.0)
##	tidyverse	* 1.2.1	2017-11-14	[2]	CRAN	(R 3.5.0)
##	tinytex	0.14	2019-06-25	[1]	CRAN	(R 3.5.2)
##	utf8	1.1.4	2018-05-24	[2]	CRAN	(R 3.5.0)
##	vctrs	0.1.0	2018-11-29	[2]	CRAN	(R 3.5.0)
##	viridisLite	0.3.0	2018-02-01	[2]	CRAN	(R 3.5.0)
##	whisker	0.3-2	2013-04-28	[2]	CRAN	(R 3.5.0)
##	withr	2.1.2	2018-03-15	[2]	CRAN	(R 3.5.0)

```
## xfun          0.8      2019-06-25 [2] CRAN (R 3.5.2)
## xml2          1.2.0    2018-01-24 [2] CRAN (R 3.5.0)
## yaml          2.2.0    2018-07-25 [2] CRAN (R 3.5.0)
## zeallot       0.1.0    2018-01-28 [2] CRAN (R 3.5.0)
##
## [1] /Users/tianyishi/Library/R/3.5/library
## [2] /Library/Frameworks/R.framework/Versions/3.5/Resources/library
```


Chapter 1

R 和 RStudio 介绍和安装教程

1.1 什么是 R

R (R Core Team 2019) 包含 R 语言和一个有着强大的统计分析及作图功能的软件系统, 由新西兰奥克兰大学的 Ross Ihaka 和 Robert Gentleman 共同开发。R 语言虽然看起来只能做统计, 实际上它麻雀虽小, 五脏俱全, 编程语言该有的特性它基本都有 (甚至支持 OOP)。

安装了 R 之后, 你可以在其自带的“R”软件中 (也可以直接在命令行使用), 但是那个软件界面比较简单, 需要记住一些命令来执行“查看当前存储的数据”, “导出 jpg 格式的图像”等操作, 对新手不太友好。因此我们使用 RStudio。

RStudio (RStudio Team 2015) 是 R 语言官方的 IDE (集成开发环境), 它的一系列功能使得编辑, 整理和管理 R 代码和项目方便很多。

了解 R 的优势, 请看第1.3节

1.2 安装 R 和 RStudio

1.2.1 安装 R

<https://cran.r-project.org>

前往CRAN, 根据自己的操作系统 (Linux, MacOS 或 Windows) 选择下载安装 R. (Linux 用户亦可参考此处)

1.2.2 安装 RStudio

<https://www.rstudio.com/products/rstudio/download/>

前往RStudio 下载页, 选择最左边免费的开源版本, 然后选择对应自己的操作系统的版本, 下载并安装。

1.3 为什么使用 R, R 与其他统计软件的比较¹

(这一小节不影响 R 的学习进度, 可以直接跳过到下一章)

SAS, SPSS, Prism, R 和 Python 是数据分析和科研作图常用的软件。

SAS, SPSS 和 Prism 都是收费的, 而且不便宜。比如 SAS 第一年需要10000 多美元, 随后每年要缴纳几千美元的年费。

R 比 SAS 功能更强大。所有 SAS 中的功能, 都能在 R 中实现, 而很多 R 中的功能无法在 SAS 中实现²。

R 和 Python (NumPy 和 SciPy) 是开源、免费的。在数据分析的应用中, R 比 Python 历史更悠久, 因此积攒了很多很棒的 packages (包)。一般来说, python 的强项是数据挖掘, 而 R 的强项是数据分析, 它们都是强大的工具。不用担心需要在二者之中做选择, 因为 rpy, reticulate 等 packages 可以让你在 python 中使用 R, 在 R 中使用 python, 详情请见第9章。无论你是数据分析零基础, 还是有 python 数据分析的经验, 都能从本书中获益。

至于 Excel, 它的定位原本就是办公 (而不是学术) 软件, 用作数据收集和初步整理是可以的, 但是做不了严谨的数据分析和大数据, 功能也非常局限。有五分之一的使用了 Excel 的遗传学论文, 数据都出现了偏差 (Ziemann, Eren, and El-Osta 2016)。

R 是 GNU 计划的一部分, 因此 R 是一个自由软件 (Libre software)。你可以在GNU 官网了解更多。

此外, R 可以完美地在 Linux 中运行。

¹Gentleman, R. (2009). *R Programming for Bioinformatics*. Boca Raton, FL: CRC Press.

²<https://thomaswdinsmore.com/2014/12/15/sas-versus-r-part-two/>

Chapter 2

获取资源与帮助（重要!）

这本书可以帮助你快速学会 R 和 tidyverse 的最常用和最重要的操作，但这仅仅是冰山一角。当你在做自己的研究的时候，会用到很多这本书中没有讲到的方法，因此学会获取资源和帮助是很重要的。以下列举几个常用的获取 R 的帮助的网站/方法：

2.1 论坛类（解答实际操作中的问题）

- 爆栈网 (StackOverflow) 是著名计算机技术问答网站（如果你有其他的编程语言基础，一定对它不陌生）。查找问题的时候加上 [R]，这样搜索结果就都是与 R 相关的了（为了进一步缩小搜索范围，可以加上其他的 tag，比如 [ggplot], [dplyr]）。注意，提问和回答的时候话语尽量精简，不要在任何地方出现与问题无关的话（包括客套话如“谢谢”），了解更多请查看其新手向导。
- 由谢益辉大佬在 2006 年（竟然比爆栈网更早!）创建的“统计之都”论坛，是做的最好的一个面向 R 的中文论坛（但是客观地说活跃度还是没爆栈网高）同样不要忘记读新手指引。

2.2 Reference 类（查找特定的 function/package 的用法，就像查字典一样）

- 直接在 R console 中执行 ? + 函数名称，比如 ?t.test
- RDocumentation 上有基础 R 语言和来自 CRAN, GitHub 和 Bioconductor 上的近 18000 个 packages 的所有的函数的说明和使用例。
- 有些 packages 会在官网/github 仓库提供使用说明，比如 tidyverse
- 有些 packages 会提供 vignettes，它们类似于使用指南，相比于函数的 documentation 更为详细且更易读。vignette()（无参数）以查看全部可用 vignettes。你可以试试 vignette("Sweave")，它是用 LaTeX 排版的，很漂亮。

2.3 教程和书籍类 (用来系统地学习)

- R 的官方 Manuals. 其中新手只需要看 *An Introduction to R*, 随后选看 *R Language Definition* 即可。部分由丁国徽翻译成中文 (点击量其实并不高.....要想把握前沿信息还是需要阅读英语的能力的)。
- *R for Data Science* by Garrett Golemund & Hadley Wickham. **tidyverse** 的作者写的一本书, 较为详细地介绍了 **tidyverse** 的用法以及一些更高深的关于编程的内容。(练习题答案)
- RStudio Resources是 RStudio 的资源区, 有关于 R 和 RStudio 的高质量教程, 还可以下载很多方便实用的 Cheat Sheet.
- *The R Book* by Michael J. Crawley
- R 的官方 FAQ (在左侧菜单栏中找到 “FAQ”)
- 存储在 CRAN 上的中文 FAQ (注意这不是英文 FAQ 的翻译, 而是一本独立的 R 入门教程)
- *Advanced R* by Hadley Wickham及其练习题答案。

2.4 速查表 (Cheat sheets) (用来贴墙上)

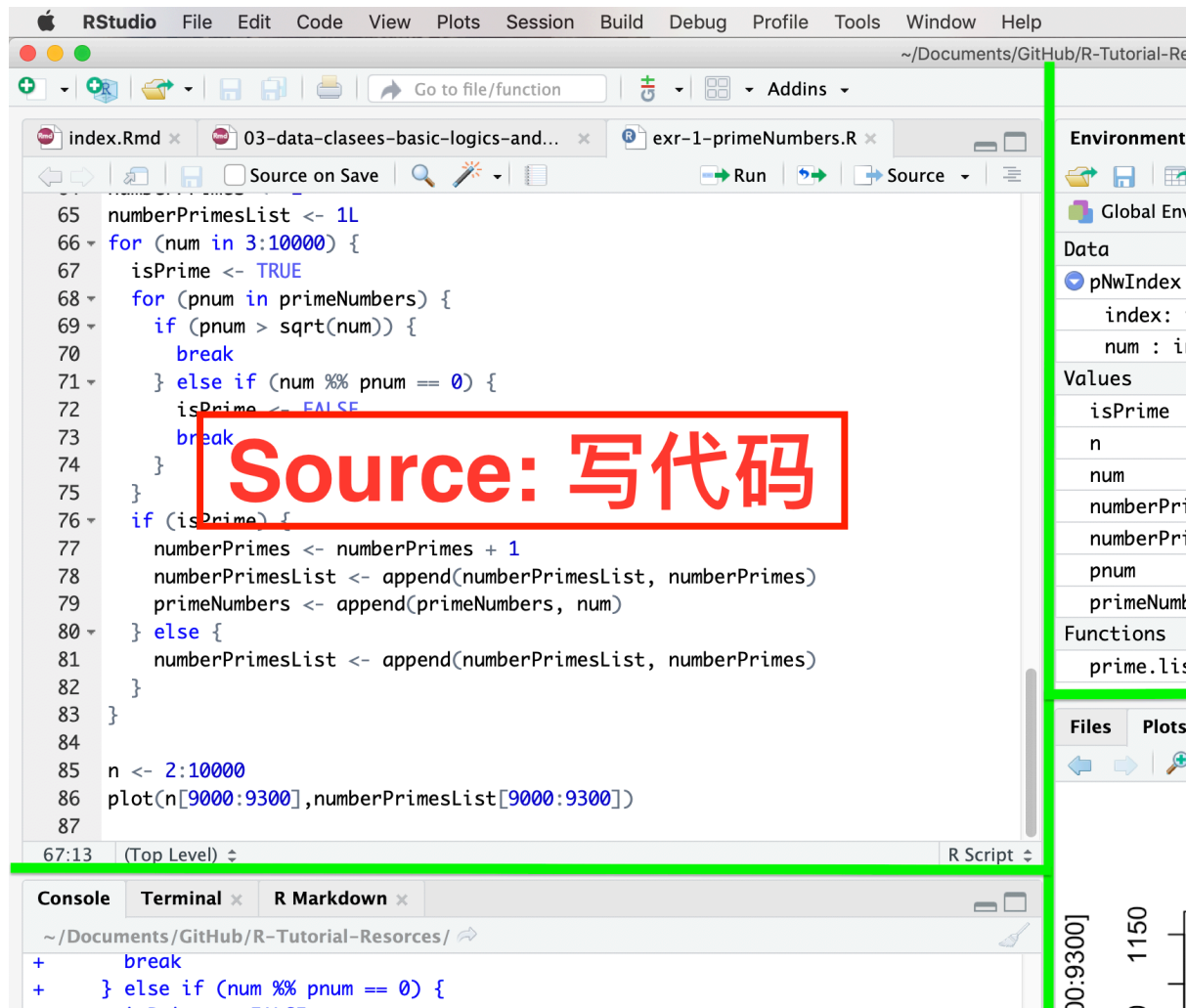
- R Reference Card 2.0 by Mayy Baggott & Tom Short以及其第一版的中文翻译
- RStudio Cheat Sheets包含了 RStudio IDE 和常用 packages 的 cheat sheets。2019 年版的合集在这里。

Chapter 3

RStudio 界面介绍, 基本操作, 和创建新项目

3.1 界面

3.1.1 概览



3.1.2 左下角: Console (控制台)

Console 是执行代码的地方。试试在里面输入 `1 + 1` 并按回车以执行。

3.1.3 左上角: Source (源)

Source 是写代码的地方。请看第3.2.2节。

这个位置也是用来查看文件和数据的地方。试试在 console 中执行 `View(airquality)` 或 `library(help = "stats")`。

3.1.4 右上角: Environment (环境),

Environment 是一个列表, 显示了所有当前工作环境中所有的变量 (“values” 和 “data”) 和自定义的函数 (functions)。

History (历史) 和 Connections (连接) 不太常使用。

3.1.5 右下角: Plots (绘图), Help (帮助), Files (文件) 和 Packages (包)

Plots 是预览图像的区域。试试在 console 中执行 `hist(rnorm(10000))`。

Help 是查看帮助文件的区域。试试在 console 中执行 `?hist` 或 `?norm`。

Files 是查看文件的区域, 默认显示工作目录 (working directory)。

Packages 是安装/查看/更新 packages (包) 的区域。详情请看第4.3章。

3.2 基本操作

3.2.1 执行代码

试着在 console 里输入 `1 + 1`, 并按回车以执行。你的 console 会显示:

```
> 1 + 1
[1] 2
```

其中 2 是计算结果, [1] 是索引, 在第5.1.2节有解释。`> 1 + 1` 是 input, `[1] 2` 是 output。

还是用 `1 + 1` 举例, 在本书中, 对于 input 和 output 的展示格式是这样的:

```
1+1
```

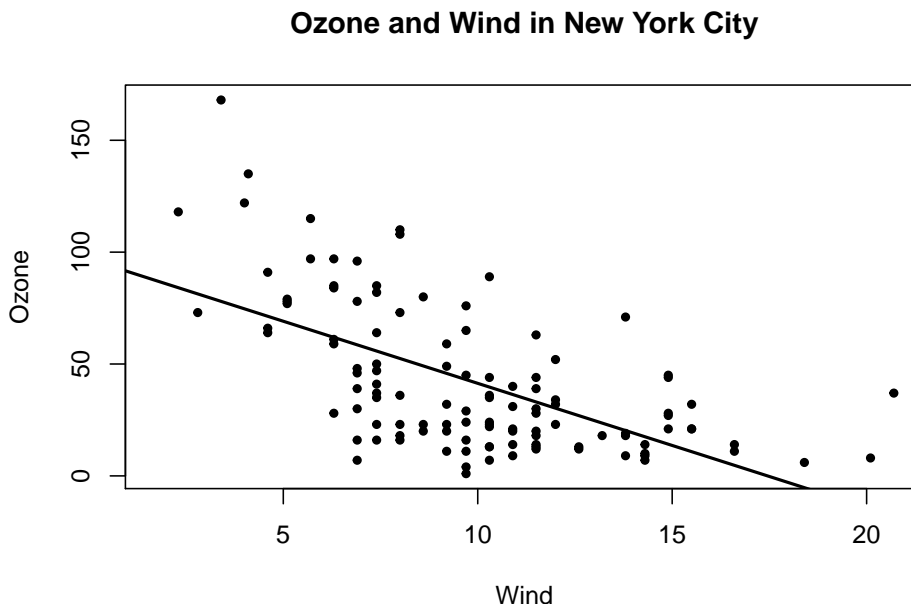
```
## [1] 2
```

注意 input 中的 `>` 被省略了, 这意味着你可以直接把代码从本书复制到你的 console 并按回车执行 (因为 console 本身自带了 `>`), 类似地, 你从其他各种网站上找到的说明书, 教程和论坛帖子中看到的 R 代码, 大多数也都是这种形式呈现, 便于复制粘贴。

再来一个例子, 试着在 console 里输入 (或者复制) 以下代码并执行:

```
attach(airquality)

## The following objects are masked from airquality (pos = 12):
##
##      Day, Month, Ozone, Solar.R, Temp, Wind
plot(Wind, Ozone, main = "Ozone and Wind in New York City", pch = 20)
model <- lm(Ozone ~ Wind, airquality)
abline(model, lwd = 2)
```



可以看到, 在 plots 区, 生成了一副漂亮的图。(先别在意每行代码具体的作用, 在之后的章节我会一一讲述)

这时, 把 RStudio 关掉, 再重新启动, 你会发现你的图没了。因此我们需要记录和管理代码。

3.2.2 记录和管理代码

初学者经常会在 console 里写代码, 或者从别处复制代码, 并执行。这对于一次性的计算 (比如写统计学作业时用 R 来算线性回归的参数) 很方便, 但是如果你想保存你的工作, 你需要把它们记录在 R script 文件里。如果你的工作比较复杂, 比如有一个 excel 表格作为数据源, 然后在 R 中用不同的方法分析, 导出图表, 这时候你会希望这些文件都集中在一起。你可以使用 R Project 来管理它们。

3.2.2.1 创建 R Project

1. 左上角 File > New Project
2. 点选 New Directory > New Project
3. 输入名称和目录并 Create Project

3.2.2.2 使用 R Project

在创建 R project 的文件夹中打开.Rproj 文件。或者, RStudio 启动的时候默认会使用上一次所使用的 R project.

随后, 你在 RStudio 中做的所有工作都会被保存到.Rproj 所在的这个文件夹 (正式的说法是“工作目录”(working directory)). 比如, 在 console 中执行:

```
pdf("normalDistrubution.pdf")
curve(dnorm(x), -5, 5)
dev.off()
```

一个正态分布的图像便以 pdf 格式保存在了工作目录。你可以在系统的文件管理器中, 或是在 RStudio 右下角 File 面板中找到。

3.2.3 写/保存/运行 R script

在 console 中运行代码, 代码得不到保存。代码需保存在 R script 文件 (后缀为.R) 里。

Ctrl+Shift+N (Mac 是 command+shift+N) 以创建新 R script.

然后就可以写 R script. 合理使用换行可以使你的代码更易读。# 是注释符号。每行第一个 # 以及之后的内容不会被执行。之前的例子, 可以写成这样:

```
# 读取数据
attach(airquality)

# 绘图
plot(Wind, Ozone, # x 轴和 y 轴
      main = "Ozone and Wind in New York City", # 标题
      pch = 20) # 使用实心圆点
model <- lm(Ozone ~ Wind, airquality) # 线性回归模型
abline(model, lwd = 2) # 回归线
```

Ctrl+Enter (command+return) 以执行一“句”代码 (比如上面的例子中, 从 plot(Wind... 到 pch = 20) 有三行, 但是它是一“句”)。

Ctrl+Shift+Enter (command+shift+return) 以从头到尾执行所有代码。

试试复制并执行以上代码吧。

Ctrl+S (command+S) 以保存 R script. 保存后会在工作目录找到你新保存的.R 文件。重新启动 RStudio 的时候, 便可以打开对应的 R script 文件以重复/继续之前的工作。

3.2.4 关于换行

Console 中每个命令开头的 `>` 叫做 prompt (命令提示符), 当它出现在你所编辑的那一行的开头时, 按下回车的时候那行的命令才会被执行。有时候它会消失, 这时候按 `esc` 可以将其恢复。

prompt 消失的主要原因是你的代码没有写完, 比如括号不完整:

```
> 2+(3+4
```

这时你按回车, 它会显示:

```
> 2+(3+4
+

```

`+` 号是在提示代码没写完整。这时你把括号补上再按回车:

```
> 2+(3+4
+ )

```

```
[1] 9
```

便可以完成计算。

这意味着我们可以把一条很长的命令分成很多行。比如我们可以写这样的代码 (在 R script 编辑器中!)

```
if(TRUE){
  print(2)
} else{
  print(3)
}
```

然后 `Ctrl+Enter` 执行。你会发现在 console 中, 从第二行开始每行开头都有一个 `+` 号。

Chapter 4

安装和使用 packages (包)

4.1 Package 是什么，为什么使用它们？

Package 是别人写好的在 R 中运行的程序（以及附带的数据和文档），你可以免费安装和使用它们。

Packages 可以增加在基础 R 语言中没有的功能，可以精简你代码的语句，或是提升使用体验。比如有个叫做 `tikzDevice` 的 package 可以将 R 中的图表导出成 tikz 语法的矢量图，方便在 LaTeX 中使用。本书的编写和排版也是使用 R 中的一个叫做 `bookdown` 的 package 完成的（真的超棒）。

这个课程主要是学习 `tidyverse` 这个 package,

4.2 如何安装 packages

首先我们安装 `tidyverse`（很重要，本书接下来的部分都要使用这个 package）：

```
install.packages("tidyverse")
```

在 console 中运行以上代码，R 就会从CRAN中下载 `tidyverse` 并安装到你电脑上的默认位置。因此安装 packages 需要网络连接。

如果想安装多个 packages，你可以一行一行地安装，或是把多个 packages 的名字合成一列，同时安装，比如：

```
install.packages(c("nycflights13", "gapminder", "Lahman"))
```

绝大多数的 packages 都能用这个方法安装，因为它们是被存储在 CRAN 上的。Bioconductor packages 请看第4.4.2节。

4.3 如何使用 packages

安装 packages 后, 有两种方法使用它们。以 `tidyverse` 为例:

```
library('tidyverse')
```

或

```
require('tidyverse')
```

两者的效果很大程度上都是一样的, 都可以用来读取单个 package。它们的不同, 以及如何通过一行指令读取多个 packages, 请参看第4.4.1节。

每次重启 R 的时候, 上一次使用的 packages 都会被清空, 所以需要重新读取。因此我们要在 R script 里面记录此 script 需要使用的 packages (这个特性可以帮助你养成好习惯: 当你把你的代码分享给别人的时候, 要保证在别人的电脑上也能正常运行, 就必须要指明要使用哪些 packages)

4.4 其它

这小节是一些不重要的内容, 因此可酌情跳到下一章 (第5章)。

4.4.1 `library()` 和 `require` 的区别; 如何使用一行指令读取多个 packages

1. `require()` 会返回一个逻辑值。如果 package 读取成功, 会返回 `TRUE`, 反之则返回 `FALSE`.
2. `library()` 如果读取试图读取不存在的 package, 会直接造成错误 (error), 而 `require()` 不会造成错误, 只会产生一个警告 (warning).

这意味着 `require()` 可以用来同时读取多个 packages:

```
lapply(c("dplyr", "ggplot2"), require, character.only = TRUE)
```

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

或者更精简一点,

```
lapply(c("dplyr", "ggplot2"), require, c = T)
```

```
## [[1]]  
## [1] TRUE  
##
```

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

4.4.2 安装 Bioconductor packages

Bioconductor是一系列用于生物信息学的 R packages. 截止 2019 年 7 月 2 日, 共有 1741 个可用的 bioconductor packages. 它们没有被存储在 CRAN 上, 因此需要用特殊的方法安装. 首先, 安装一系列 Bioconductor 的核心 packages (可能需要几分钟):

```
source("http://bioconductor.org/biocLite.R")  
biocLite()
```

然后, 通过 `biocLite()` 函数安装其它 packages, 比如:

```
biocLite("RforProteomics")
```


Chapter 5

向量，逻辑，循环和函数

注意，*R* 中的变量名/自定义函数名不能以数字和特殊符号开头，中间只能使用“_”“和”“.”作为特殊符号¹

5.1 向量的概念，操作和优越性

R 没有标量，它通过各种类型的向量 (vector) 来存储数据。

5.1.1 创建向量 (赋值)

与很多其他的计算机语言不同，在 *R* 中，`<-`（像一个小箭头）用于给向量，数据框和函数赋值（即在每行的开头）。在 *RStudio* 中，可以用 `Alt+-` (Mac 是 `option+-`) 这个快捷键打出这个符号。

```
x <- 2
x
```

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

虽然 `=` 也可以用，但是绝大多数 *R* 用户还是采用标准的 `<-` 符号，而 `=` 则用于给函数的参数赋值。

要创建一个多元素的向量，需要用到 `c()` (concatenate) 函数：

```
nums <- c(1,45,78)
cities <- c("Zürich", "上海", "Tehrān")
nums
```

```
## [1] 1 45 78
```

¹如果非要违反规则，可以使用转义符号`\``，比如可以 `\`4foo%b=a+r` <- 50` “

```
cities
```

```
## [1] "Zürich" "上海" "Tehrān"
```

通过 `length()` 函数, 可以查看向量的长度。

```
length(nums)
```

```
## [1] 3
```

如果无后续使用, 没必要赋值一个变量; `c(...)` 的计算结果就是一个向量, 并直接传给 `length()` 函数

```
length(c("Guten Morgen"))
```

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

(每个被引号包围的一串字符, 都只算做一个元素, 因此长度为 1; 多元素的向量请看第5.1.1节)

还是通过 `c()` 函数, 可以把多个向量拼接起来:

```
cities_1 <- c("Zürich", "上海", "Tehrān")
cities_2 <- c("大阪", "Poznań", " ")

cities <- c(cities_1, cities_2, c("Jyväskylä", "□□", " "))

cities
```

```
## [1] "Zürich"      "上海"      "Tehrān"      "大阪"
## [5] "Poznań"      " "         "Jyväskylä"   "□□"
## [9] " "          " "          " "          " "
```

5.1.2 索引/取子集 (indexing/subsetting)

索引 (index) 就是一个元素在向量中的位置。R 是从 1 开始索引的, 即索引为 1 的元素是第一个元素 (因此用熟了 Python 和 C 可能会有些不适应)。在向量后方使用方括号进行取子集运算 (即抓取索引为对应数字的元素; 虽然 subsetting 翻译成“取子集”有点怪, 但是没毛病; 不知大家有没有更好的翻译方法, 或是不翻译更好)。

```
x <- c("one", "two", "three", "four", "five", "six", "seven", "eight", "nine")
x[3]
```

```
## [1] "three"
```

可以在方括号中使用另一个向量抓取多个元素:

```
x[c(2,5,9)] # 第 2 个, 第 5 个, 第 9 个元素
```

```
## [1] "two" "five" "nine"
```

经常, 我们会抓取几个连续的元素。如果想知道方法, 请继续往下看。

5.1.3 生成器

有时候我们需要其元素按一定规律排列的向量，这时，相对于一个个手动输入，有更方便的方法：

5.1.3.1 连续整数

```
1:10 # 从左边的数（包含）到右边的数（包含），即 1:10
```

```
## [1] 1 2 3 4 5 6 7 8 9 10
```

这时，你应该会有个大胆的想法：

```
x[3:6]
```

```
## [1] "three" "four" "five" "six"
```

没错就是这么用的，而且极为常用。

当元素比较多时：

```
y <- 7:103 # 复习一下赋值
y
```

```
## [1] 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [18] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
## [35] 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
## [52] 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74
## [69] 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91
## [86] 92 93 94 95 96 97 98 99 100 101 102 103
```

注意到了左边方括号中的数字了吗？它们正是所对应的那一行第一个元素的索引。

下面的内容可能有点偏，可以酌情从这里跳到第5.1.5节。

5.1.3.2 复读机 rep()

```
rep(c(0, 7, 6, 0), 4) # 把 [0, 7, 6, 0] 重复 4 遍
```

```
## [1] 0 7 6 0 0 7 6 0 0 7 6 0 0 7 6 0
```

5.1.3.3 等差数列：seq()

公差确定时：

```
seq(0, 15, 2.5) # 其实是 `seq(from = 0, to = 50, by = 5)` 的简写
```

```
## [1] 0.0 2.5 5.0 7.5 10.0 12.5 15.0
```

长度确定时:

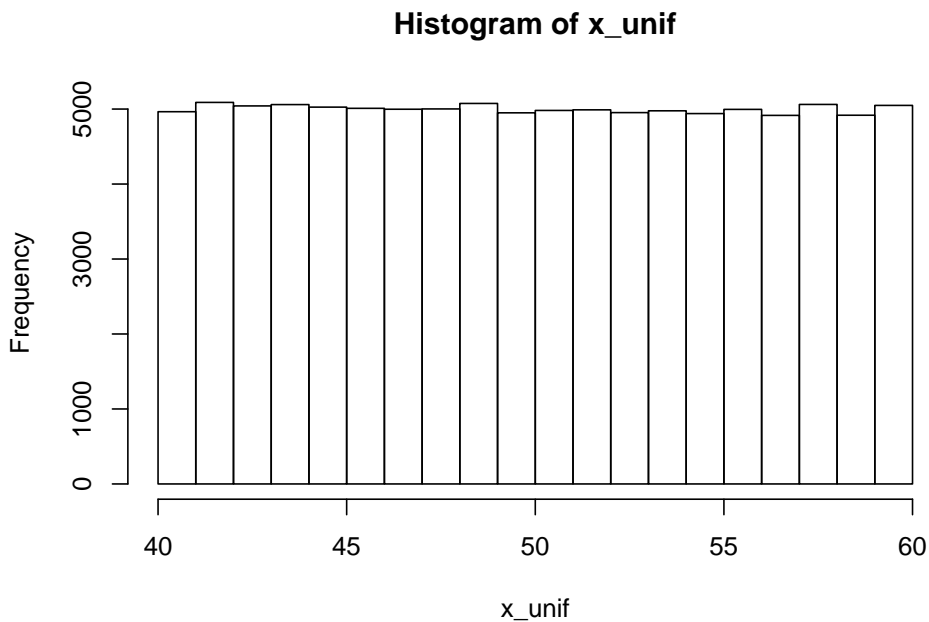
```
seq(0, 50, length.out = 11) # 其实是 `seq(from = 0, to = 50, length.out = 11)` 的简写
```

```
## [1] 0 5 10 15 20 25 30 35 40 45 50
```

5.1.3.4 随机数:

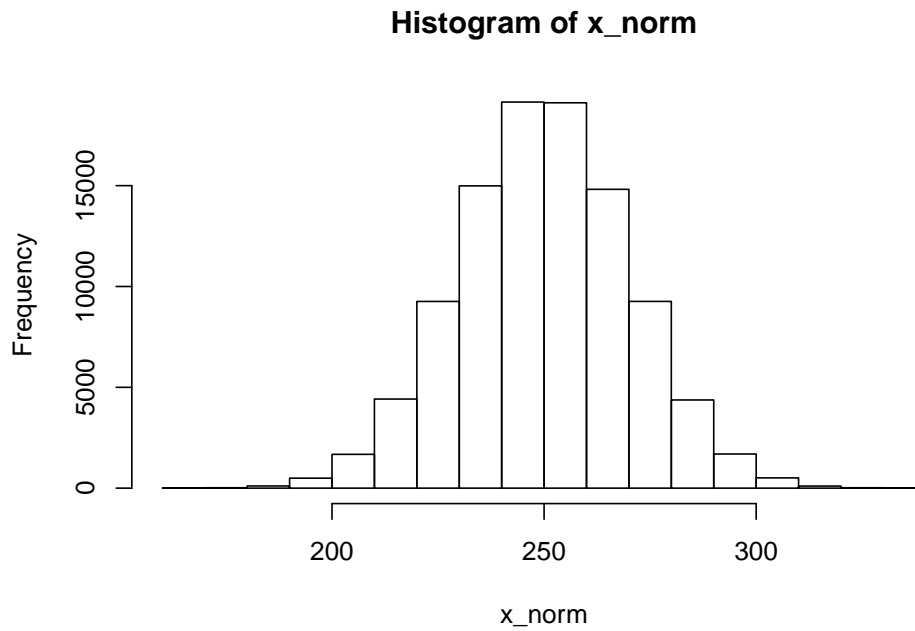
连续型均匀分布随机数用 `runif(n, min, max)`, `n` 是数量, `min` 是最小值, `max` 是最大值。默认 `min` 为 0, `max` 为 1。

```
x_unif <- runif(100000, 40, 60) # 生成 100000 个 40 到 60 之间, 连续均匀分布的随机数
hist(x_unif) # 画直方图
```



正态分布随机数用 `rnorm(n, mean, sd)`, 三个参数分别为数量, 平均值, 标准差。默认 `mean` 为 0, `sd` 为 1。

```
x_norm <- rnorm(100000, 250, 20) # 按照平均值为 250, 标准差为 20 的正态分布的概率密度函数生成
hist(x_norm) # 画直方图
```



5.1.4 向量的其它操作

5.1.4.1 `sort()`, `rank()` 和 `order()`

5.1.5 R 向量的优越性

R 中的向量（矩阵和数列也是）的各种计算默认都是逐元素（elementwise）的。比如：

```
x <- c(4, 9, 25)
y <- c(8, 6, 3)
x + y
```

```
## [1] 12 15 28
```

```
x * y # 在 matlab 中这样乘是不行的，要用 `.*`，除法也是
```

```
## [1] 32 54 75
```

```
sqrt(x)
```

```
## [1] 2 3 5
```

相比于常用的编程语言，省去了 for 循环；相比于 matlab 的默认矩阵乘法，逐元素乘法在数据处理中更有用。

5.2 数据类型 (Data Types)

向量所存储的数据类型有：

类型	含义与说明	例子
numeric	浮点数向量	3, 0.5, <code>sqrt(2)</code> , NaN, Inf
integer	整数向量	3L, 100L
character	字符向量；需被引号包围	"1", "\$", " 你好"
logical	逻辑向量	TRUE, FALSE, NA
complex	复数向量	3+5i, 1i, 1+0i

通过 `class()` 函数，可以查看向量的类型。

```
class(" 早上好")

## [1] "character"

(除此之外，typeof()，mode()，storage.mode() 这三个函数的功能与 class() 类似，
但有重要的区别；为避免造成困惑，此处不展开讨论)。
```

5.3 数学表达和运算

5.3.1 数的表达

5.3.1.1 浮点数

除非指定作为整数（见下），在 R 中所有的数都被存储为双精度浮点数的格式 (double-precision floating-point format)，其 `class` 为 `numeric`。

```
class(3)

## [1] "numeric"

这会导致一些有趣的现象，比如  $(\sqrt{3})^2 \neq 3$ ：—(强迫症患者浑身难受)—

sqrt(3)^2-3

## [1] -4.440892e-16
```

浮点数的计算比精确数的计算快很多。如果你是第一次接触浮点数，可能会觉得它不可靠，其实不然。在绝大多数情况下，牺牲的这一点点精度并不会影响计算结果（我们的结果所需要的有效数字一般不会超过 10 位；只有当两个非常，非常大且数值相近对数字相减才会出现较大的误差）。

NaN（非数）和 Inf（无限大）也是浮点数！

```
class(NaN)

## [1] "numeric"
class(Inf)

## [1] "numeric"
```

5.3.1.2 科学计数法

在 R 中可以使用科学计数法 ($A \times 10^B$), 比如:

```
3.1e5

## [1] 310000
-1.2e-4+1.1e-5

## [1] -0.000109
```

5.3.1.3 整数

整数的 class 为 `integer`。有两种常见的方法创建整数: 1) 在数后面加上 `L`;

```
class(2)

## [1] "numeric"
class(2L)

## [1] "integer"
```

2) 创建数列

```
1:10 # 公差为 1 的整数向量生成器, 包含最小值和最大值

## [1] 1 2 3 4 5 6 7 8 9 10
class(1:10)

## [1] "integer"
seq(5,50,5) # 自定义公差, 首项, 末项和公差可以不为整数

## [1] 5 10 15 20 25 30 35 40 45 50
class(seq(5,50,5)) # 因此产生的是一个浮点数向量

## [1] "numeric"
seq(5L,50L,5L) # 可以强制生成整数

## [1] 5 10 15 20 25 30 35 40 45 50
```

```
class(seq(5L,50L,5L)) # 是 整数没错
```

```
## [1] "integer"
```

整数最常见的用处是 indexing (索引)。

5.3.1.3.1 整数变成浮点数的情况

这一小段讲的比较细, 请酌情直接跳到下一节 (5.3.2)。

整数与整数之前的加, 减, 乘, 求整数商, 和求余数计算会得到整数, 其他的运算都会得到浮点数, (阶乘 (factorial) 也是, 即便现实中不管怎么阶乘都不可能得到非整数):

```
class(2L+1L)
```

```
## [1] "integer"
```

```
class(2L-1L)
```

```
## [1] "integer"
```

```
class(2L*3L)
```

```
## [1] "integer"
```

```
class(17L%%3L)
```

```
## [1] "integer"
```

```
class(17L%%3L)
```

```
## [1] "integer"
```

```
class(1000L/1L)
```

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

```
class(3L^4L)
```

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

```
class(sqrt(4L))
```

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

```
class(log(exp(5L)))
```

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

```
class(factorial(5L))
```

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

整数与浮点数之间的运算, 显然, 全部都会产生浮点数结果, 无需举例。

另外一个需要注意的地方是，取整函数`5.3.2.3`并不会产生整数。如果需要的话，要用 `as.integer()` 函数。

5.3.2 运算

5.3.2.1 二元运算符

R 中的 binary operators (二元运算符) 有：

符号	描述
<code>+</code>	加
<code>-</code>	减
<code>*</code>	乘
<code>/</code>	除以
<code>^</code> 或 <code>**</code>	乘幂
<code>%%</code>	求整数商，比如 <code>7%%3= 2</code>
<code>%%</code>	求余数，比如 <code>7%%3= 1</code>

其中求余/求整数商最常见的两个用法是判定一个数的奇偶性，和时间，角度等单位的转换。(后面再详细介绍)。

5.3.2.2 e^x 和 $\log_x y$

`exp(x)` 便是运算 e^x 。如果想要 $e = 2.71828...$ 这个数：

```
exp(1)
```

[1] 2.718282

`log(x, base=y)` 便是运算 $\log_y x$ ，可以简写成 `log(x,y)` (简写需要注意前后顺序，第5.6.1有解释)。

默认底数为 e ：

```
log(exp(5))
```

[1] 5

有以 10 和 2 为底的快捷函数，`log10()` 和 `log2()`

```
log10(1000)
```

[1] 3

```
log2(128)
```

[1] 7

5.3.2.3 近似数 (取整, 取小数位, 取有效数字)

注意, 取整函数给出的结果不是整数!

```
class(ceiling(7.4))
```

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

5.3.2.4 R 中自带的数学函数集合

函数	描述
<code>exp(x)</code>	e^x
<code>log(x,y)</code>	$\log_y x$
<code>log(x)</code>	$\ln(x)$
<code>sqrt(x)</code>	\sqrt{x}
<code>factorial(x)</code>	$x! = x \times (x-1) \times (x-2) \dots \times 2 \times 1$
<code>choose(n,k)</code>	$\binom{n}{k} = \frac{n!}{k!(n-k)!}$ (二项式系数)
<code>gamma(z)</code>	$\Gamma(z) = \int_0^\infty x^{z-1} e^{-x} dx$ (伽马函数)
<code>lgamma(z)</code>	$\ln(\Gamma(z))$
<code>floor(x), ceiling(x), trunc(x),</code> <code>round(x, digits = n)</code>	取整; 见上一小节。 四舍五入, 保留 n 个小数位, n 默认为 0
<code>signif(x,digits = n)</code>	四舍五入, 保留 n 个有效数字, n 默认为 6)
<code>sin(x), cos(x), tan(x)</code>	三角函数
<code>asin(x), acos(x), atan(x)</code>	反三角函数
<code>sinh(x), cosh(x), tanh(x)</code>	双曲函数
<code>abs(x)</code>	$ x $ (取绝对值)

5.4 逻辑

5.4.1 Logical Values (逻辑值)

逻辑值有三个。TRUE, FALSE 和 NA.

```
class(c(TRUE,FALSE,NA))
```

```
## [1] "logical"
```

TRUE 为真, FALSE 为假, NA 为未知 (即真假难辨)。

5.4.2 Logical Operations (逻辑运算)

R 中常用的关系运算符有:

符号	描述
==	equal to (等于)
!=	equal to (不等于)
<	less than (小于)
>	more than (大于)
<=	less than or equal to (小于等于)
>=	more than or equal to (大于等于)

使用关系运算符进行计算，会产生逻辑值作为结果。比如：

```
x <- 5
x != 3 #x 等于 5，所以 “x 不等于 3” 为真
```

```
## [1] TRUE
```

有一些其他的运算符或函数也会返回逻辑值，比如

```
7 %in% c(1,4,5,6,7)
```

```
## [1] TRUE
```

顾名思义，这个运算符是用来检测一个元素是否在另一个向量中。其它类型的运算符，我在需要用到的时候再讲。

5.4.3 逻辑运算详解

5.4.3.1 逻辑运算符

以下是最常用的三个逻辑运算符。

符号	描述
&	AND (且)
	OR (或)
!	反义符号

5.4.3.2 反义符号 (!)

! 使 TRUE FALSE 颠倒。一般，我们用小括号来包住一个逻辑运算，然后在它的前面加上一个! 来反转结果，比如

```
!(3 < 4) # 这个例子很简单，反义符号意义不大。后面实操的时候才能领略到它的用处。
```

```
## [1] FALSE
```

5.4.3.3 多个逻辑运算的组合 (& (且) 和 | (或))

& 和 | 可以把多个逻辑运算的结果合并成一个逻辑值。

& 判断是否两边运算结果都为 TRUE。如果是, 才会得到 TRUE (即一真和一假得到假)。(正统的翻译貌似是“与”, 但是我觉得不太与中文语法适配; 想想“ x 大于 2 与 y 小于 4”是不是比“ x 大于 2 且 y 小于 4”别扭)

| 判断两边运算结果是否至少有一个 TRUE, 如果是, 就会得到 TRUE。

不用死记硬背! 其实就是“且”和“或”的逻辑。

用脑子想一下以下三条运算的结果, 然后复制代码到 R console 对答案。

```
1 == 1 & 1 == 2 & 3 == 3 # 即: “1 等于 1 且 1 等于 2 且 3 等于 3”, 是真还是假?
FALSE | FALSE | TRUE # FALSE/TRUE 等价于一个运算结果
!(FALSE | TRUE) & TRUE # 注意反义符号
```

我们可以查看三个逻辑值所有两两通过 & 组和的计算结果 (如果你不感兴趣, 可以不看方法。这里重点是结果):

```
vals <- c(TRUE, FALSE, NA)
names(vals) <- paste('[', as.character(vals), ']', sep = '')
outer(vals, vals, "&")
```

```
##           [TRUE] [FALSE] [NA]
## [TRUE]      TRUE  FALSE  NA
## [FALSE]  FALSE  FALSE FALSE
## [NA]         NA   FALSE  NA
```

可以看到, FALSE 与任何逻辑值组合, 结果都是 FALSE。这个好理解, 因为一旦一个是 FALSE, 那么不可能两边都是 TRUE。TRUE & NA 之所以为 NA (而不是 FALSE), 是因为 NA 的意思是“不能确定真假”, 即有可能真也有可能假。因此 TRUE & NA 也无法辨真假。

再来看 | 的组合:

```
outer(vals, vals, "|")
```

```
##           [TRUE] [FALSE] [NA]
## [TRUE]      TRUE   TRUE  TRUE
## [FALSE]     TRUE  FALSE  NA
## [NA]        TRUE    NA   NA
```

可以看到, TRUE 与任何一个逻辑值组合, 都是 TRUE, 而 FALSE | NA 为 NA。原因一样 (因为 NA 的不确定性)。

5.5 判断和循环 (控制流程)

如果你学过其他编程语言, 知道判断和循环的作用, 只是需要知道在 R 中的表达, 那么请看以下两个例子快速入门, 然后跳过本节 (如果没学过, 请往后面看):

```
foo <- 1:100 # 产生一个 [1,2,3,...,99,100] 的整数向量。上面讲过。
bar <- vector("numeric")
for (i in foo) {
  if (i %% 2 == 0) {
    bar <- append(bar, i^2)
  } else if (i == 51) {
    break
  }
}
bar
```

```
## [1] 4 16 36 64 100 144 196 256 324 400 484 576 676 784
## [15] 900 1024 1156 1296 1444 1600 1764 1936 2116 2304 2500
```

```
logi = TRUE
num <- 1
while (num <= 100) {
  if (logi) {
    num = num + 10 # R 不支持 num += 5 的简写
    print(num)
    logi = FALSE
  } else {
    num = num + 20
    print(num)
    logi = TRUE
  }
}
```

```
## [1] 11
## [1] 31
## [1] 41
## [1] 61
## [1] 71
## [1] 91
## [1] 101
```

5.5.1 if, else, else if 语句

if 语句长这样：

```
if (something is true/false) {
  do something
}
```

其中小括号内为测试的条件，其运算结果需为 TRUE 或 FALSE（关于逻辑值的计算请看第5.4节。若运算结果为 TRUE，大括号内的语句将会被执行。

注意, 不能直接用 `x == NA` 来判断是否是 NA, 而要用 `is.na(x)`

R 中没有专门的 `elseif` 语句, 但用 `else` 加上 `if` 能实现同样的效果。

5.6 函数

不像很多其他语言的函数有 `value.func()` 和 `func value` 等格式, R 中所有函数的通用格式是这样的:

```
function(argument1=value1, argument2=value2, ...)
```

比如

```
x1 <- c(5.1,5.2,4.5,5.3,4.3,5.5,5.7)
t.test(x=x1, mu = 4.5)
```

```
##
##  One Sample t-test
##
## data:  x1
## t = 3.0308, df = 6, p-value = 0.02307
## alternative hypothesis: true mean is not equal to 4.5
## 95 percent confidence interval:
##  4.612840 5.558589
## sample estimates:
## mean of x
##  5.085714
```

(英语中, “parameter” 或 “formal argument” 二词用于函数定义, “argument” 或 “actual argument” 二词用于调用函数 (Kernighan and Ritchie 1988), 中文里分别是 “形式参数” 和 “实际参数”, 但是多数场合简称 “参数”。)

5.6.1 调用函数时的简写

以 `seq` 函数为例, 通过查看 documentation (在 console 执行 `?seq`) 可以查看它的所有参数:

```
## Default S3 method:
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)
```

可以看到第一个参数是 `from`, 第二个是 `to`, 第三个是 `by`, 以此类推。因此我们执行 `seq(0, 50, 10)` 的时候, R 会自动理解成 `seq(from = 0, to = 50, by = 10)`。而想用指定长度的方法就必须写清楚是 `length.out` 等于几。

`length.out` 本身也可以简写:

```
seq(0, 25, l = 11)
```

```
## [1] 0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5 20.0 22.5 25.0
```

因为参数中只有 `length.out` 是以 `l` 开头的, `l` 会被理解为 `length.out`. 但是这个习惯并不好; 自己用用就算了, 与别人分享自己的工作时请务必使用标准写法。

5.6.2 关于...

有时候, 你想写的函数可能有数量不定的参数, 或是有需要传递给另一个函数的“其它参数”(即本函数不需要的参数), 这时候可以在函数定义时加入一个名为... 的参数, 然后用 `list()` 来读取它们。list 是进阶内容, 在第6.3.1.1节有说明。

比如我写一个很无聊的函数:

```
my_func <- function(arg1, arg2 = 100, ...){
  other_args <- list(...)
  print(arg1)
  print(arg2)
  print(other_args)
}

my_func("foo", cities = c(" 崇阳", "A ", " つがる"), nums = c(3,4,6))

## [1] "foo"
## [1] 100
## $cities
## [1] " 崇阳" "A " " つがる"
##
## $nums
## [1] 3 4 6
```

`arg1` 指定了是"foo" (通过简写), 因此第一行印出"foo"; `arg2` 未指定, 因此使用默认值100, 印在第二行。`cities` 和 `nums` 在形式参数中没有匹配, 因此归为 "...", 作为 `list` 印在第三行及之后。

5.7 简易的统计学计算

本小节简要解释了 R 中的 `t` 分布, `t` 检验和 χ^2 检验。统计学方法并不是本书的重点, 因此可以酌情跳到下一章。

5.7.1 `t` 分布

众所周知, `t` 分布长这样:

```

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <e2>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <89>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <a4>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <e2>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <89>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <a4>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <e2>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <89>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <a4>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <e2>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <89>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <a4>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <e2>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <89>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <a4>

```

```
## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <e2>

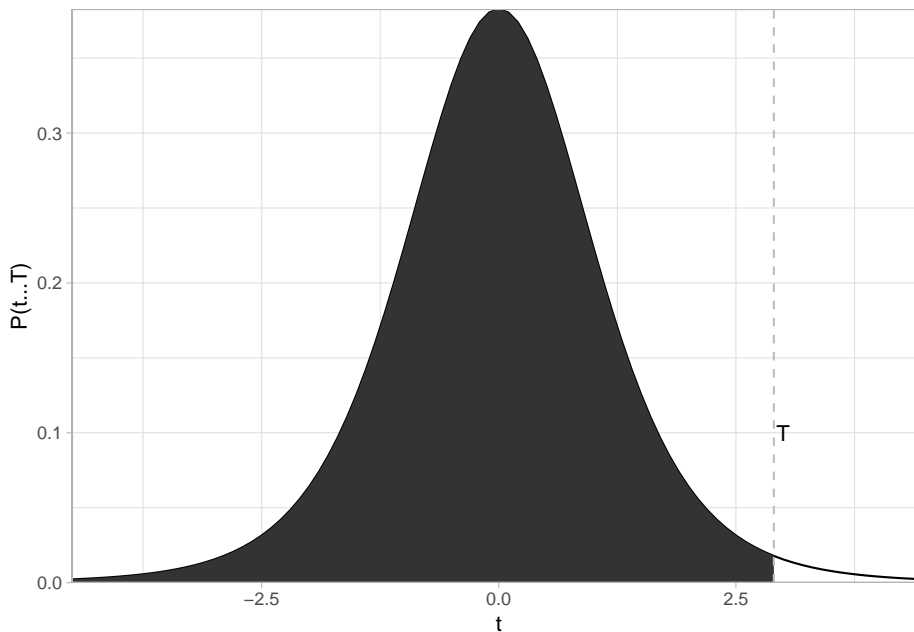
## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <89>

## Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
## conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for <a4>

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x,
## x$y, : conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for
## <e2>

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x,
## x$y, : conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for
## <89>

## Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x,
## x$y, : conversion failure on 'P(t T)' in 'mbcsToSbcs': dot substituted for
## <a4>
```



阴影面积为 $P(t < T)$, 虚线对应的 t 为 T . `qt()` 可以把 $P(tT)$ 的值转化成 T , `pt()` 则相反。

假设你需要算一个 confidence interval (置信区间), confidence level (置信等级) 为 95%, 即 $\alpha = 0.05$, degrees of freedom(自由度) 为 12, 那么怎么算 t^* 呢?

```
qt(0.975, df = 12)
```

```
## [1] 2.178813
```

为什么是 0.975? 因为你要把 0.05 分到左右两边, 所对应的 t^* 就等同于 t 分布中, $P(tT) = 0.975$ 时 T 的值。

再举一个例子, 你在做 t 检验, 双尾的, 算出来 $t = 1.345$, 自由度是 15, 那么 p 值怎么算呢?

```
p <- (1-(pt(2.2, df = 15)))*2
p
```

```
## [1] 0.04389558
```

其中 `pt(2.2, df = 15)` 算出阴影面积 ($P(tT)$ 的值), 1 减去它再乘以 2 就是对应的双尾 t 检验的 p 值。

5.7.2 z 分布

没有 z 分布专门的函数。可以直接用 t 分布代替, 把 `df` 调到很大 (比如 999999) 就行了。比如我们试一下 95% 置信区间所对应的 z^* :

```
qt(0.975, 9999999)
```

```
## [1] 1.959964
```

(果然是 1.96)

5.7.3 t 检验

t 检验分为以下几种:

- One sample t test (单样本)
- paired t test (配对)
- Two sample... (双样本)
 - Unequal variance (Welch) t test (不等方差)
 - Equal variance t test (等方差)

在 R 中做 t 检验, 很简单, 以上这些 t 检验, 都是用 `t.test` 这个函数去完成。

以单样本为例:

```
x <- c(2.23, 2.24, 2.34, 2.31, 2.35, 2.27, 2.29, 2.26, 2.25, 2.21, 2.29, 2.34, 2.32)
t.test(x, mu = 2.31)
```

```
##
```

```
## One Sample t-test
```

```
##
```

```
## data: x
```

```
## t = -2.0083, df = 12, p-value = 0.06766
```

```
## alternative hypothesis: true mean is not equal to 2.31
```

```
## 95 percent confidence interval:
```



```
## 2.257076 2.312155
## sample estimates:
## mean of x
## 2.284615
```

可以看到 $p = 0.06766$ 。

R 的默认是双尾检验，你也可以设置成单尾的：

```
x <- c(2.23,2.24,2.34,2.31,2.35,2.27,2.29,2.26,2.25,2.21,2.29,2.34,2.32)

t.test(x, mu = 2.31, alternative = "less") # 检验是否 *less* than
```

```
##
## One Sample t-test
##
## data: x
## t = -2.0083, df = 12, p-value = 0.03383
## alternative hypothesis: true mean is less than 2.31
## 95 percent confidence interval:
## -Inf 2.307143
## sample estimates:
## mean of x
## 2.284615
```

p 值瞬间减半。

双样本/配对：

```
x <- c(2.23,2.24,2.34,2.31,2.35,2.27,2.29,2.26,2.25,2.21,2.29,2.34,2.32)
y <- c(2.27,2.29,2.37,2.38,2.39,2.25,2.39,2.16,2.55,2.81,2.19,2.44,2.22)

t.test(x, y)
```

```
##
## Welch Two Sample t-test
##
## data: x and y
## t = -1.5624, df = 13.65, p-value = 0.1411
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -0.18460351 0.02921889
## sample estimates:
## mean of x mean of y
## 2.284615 2.362308
```

R 的默认是 non-paired, unequal variance, 你可以通过增加 `paired = TRUE`, `var.equal = TRUE` 这两个参数来改变它。

```
t.test(x, y, paired = TRUE)

##
## Paired t-test
##
## data: x and y
## t = -1.4739, df = 12, p-value = 0.1662
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
## -0.19253874 0.03715412
## sample estimates:
## mean of the differences
## -0.07769231
```

5.7.4 χ^2 检验

χ^2 检验有两种, goodness of fit test (适配度检验) 和 contingency table test/test of independence (列联表分析/独立性检验)。都是用 `chisq.test()` 函数去完成。

5.7.4.1 适配度检验

假设我们制造了一个有问题的骰子, 使 1 至 6 朝上的概率分别为:

```
expected_probs <- c(0.05, 0.1, 0.15, 0.2, 0.2, 0.3)
```

然后我们投掷了 100 次, 实际 1 至 6 朝上的次数分别为:

```
observed_vals <- c(6, 9, 14, 24, 18, 29)
```

通过 `chisq.test()`, 检验实际的 1 至 6 朝上概率是否与预期有偏差:

```
chisq.test(observed_vals, p = expected_probs) # 参数 p 是指概率
```

```
##
## Chi-squared test for given probabilities
##
## data: observed_vals
## X-squared = 1.4, df = 5, p-value = 0.9243
```

p 值很大 (远大于 0.05), 因此结论是骰子各面朝上的概率符合预期。

如果不指定 p 参数, 默认为检测是否所有值相等 (即骰子的所有面朝上的概率相等):

```
chisq.test(observed_vals)
```

```
##
## Chi-squared test for given probabilities
##
```

```
## data:  observed_vals
## X-squared = 23.24, df = 5, p-value = 0.0003037
```

这时 p 值小于 0.05. 得出“骰子各面朝上的概率不等”的结论。

5.7.4.2 列联表分析/独立性检验

假设我们有一组不同年级的学生参加社团的人数数据：

```
(社团参与 <- matrix(c(28,36,40,40,32,33,38,29,36), nrow = 3, dimnames = list(c(" 一年级", " 二年级", " 三年级", " 四年级", " 五年级", " 六年级"), c(" 棒球", " 足球", " 网球")))
```

```
##          棒球  足球  网球
## 一年级    28   40   38
## 二年级    36   32   29
## 三年级    40   33   36
```

我们想知道社团的参与，与所在年级是否是独立事件：

```
chisq.test(社团参与)
```

```
##
##  Pearson's Chi-squared test
##
## data:  社团参与
## X-squared = 3.7587, df = 4, p-value = 0.4396
```

p 值不小于 0.05, 无法拒绝“社团的参与，与所在年级是独立事件”的虚无假设。

彩蛋：用 R 代码实现卡方分布的概率密度函数的图像：

```
# 其实还可以更精简，但是为了易读性不得不牺牲一点精简度。
Z <- matrix(rep(rnorm(1000000), 6), nrow = 6)^2

X <- Z^2

Q <- matrix(nrow = 6, ncol = 1000000)

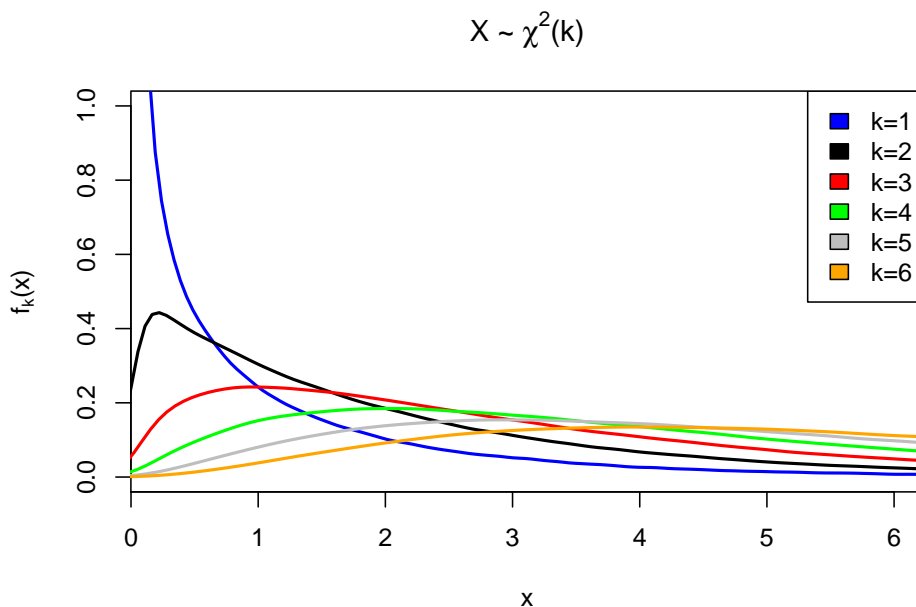
for (i in (1+1):6) {
  Q[1,] = Z[1,]
  Q[i,] = Q[(i-1),] + Z[i,]
}

plot(NULL, xlim=c(0.23,6), ylim = c(0,1),
     main = expression(paste('X ~ ', chi^'2', '(k)')),
     xlab = "x",
     ylab= expression(f[k]*'(x)'))
colors <- c('blue', 'black', 'red', 'green', 'gray', 'orange')
```

```

for (i in 1:6) {
  lines(density(Q[i,]),
        col=colors[i],
        lwd=2)
}
legend('topright',c('k=1','k=2','k=3','k=4','k=5','k=6'),
      fill = colors)

```



5.7.4.3 其它

R 自带的检验还有 `Box.test()`, `PP.test()`, `ansari.test()`, `bartlett.test()`, `wilcox.test` 等共 31 种。查看帮助文件或利用网络资源以了解更多/

Chapter 6

dataframe（数据框）和 tibble

6.1 查看 dataframe/tibble 并了解它们的结构

6.1.1 dataframe/tibble 的基本概念

dataframe 是 R 中存储复杂数据的格式，它直观易操作。tibble 是 tidyverse 的一部分，它是 dataframe 的进化版，功能更强大，更易操作。

我们来看个例子：

首先加载 tidyverse：

```
require(tidyverse)
```

以后每次跟着本书使用 R 的时候，都要先加载 tidyverse，不再重复提醒了。

tidyverse 中自带一些范例数据，比如我们输入：

```
mpg
```

```
> mpg  开头：指明行数 (234) 和列数 (11)
# A tibble: 234 x 11
```

	manufacturer	model	displ
	<chr>	<chr>	<dbl>
1	audi	a4	1.8
2	audi	a4	1.8
3	audi	a4	2.0
4	audi	a4	2.0
5	audi	a4	2.0
6	audi	a4	2.0
7	audi	a4	3.0
8	audi	a4 quattro	1.8
9	audi	a4 quattro	1.8
10	audi	a4 quattro	2.0

```
# ... with 224 more rows
```

左侧数字：observation (观测单位)

这张图是重中之重。一个正确的 dataframe/tibble, 每一行代表的是一个 observation (硬翻译的话是“观测单位”, 但是我觉得这个翻译不好), 每一列代表的是一个 variable (变量),

且同一个变量的数据类型必须一样¹。像这样的数据被称为“tidy data”（“整齐的数据”）。虽然看起来简单，直观，理所当然，但是现实中人们经常会做出“不整齐”的数据。把不整齐的数据弄整齐是下一章的重点。

6.1.2 查看更多数据

R 默认显示 dataframe/tibble 的前 10 行。如果想看最后 6 行，可以使用 `tail()` 函数，比如：

```
tail(mpg)
```

```
## # A tibble: 6 x 11
##   manufacturer model  displ  year   cyl trans drv   cty   hwy fl   class
##   <chr>          <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
## 1 volkswagen    passat  1.8  1999     4 auto~ f     18    29 p   mids~
## 2 volkswagen    passat  2    2008     4 auto~ f     19    28 p   mids~
## 3 volkswagen    passat  2    2008     4 manu~ f     21    29 p   mids~
## 4 volkswagen    passat  2.8  1999     6 auto~ f     16    26 p   mids~
## 5 volkswagen    passat  2.8  1999     6 manu~ f     18    26 p   mids~
## 6 volkswagen    passat  3.6  2008     6 auto~ f     17    26 p   mids~
```

若要从头到尾查看全部数据，可以使用 `View` 函数：

```
View(mpg)
```

6.2 tibble 的创建和基础操作

6.2.1 创建 tibble

6.2.1.1 手动输入数据以创建 tibble

使用 `tibble` 函数，按以下格式创建 tibble。换行不是必须的，但是换行会看得更清楚。如果换行，不要忘记行末的逗号。

```
my_tibble_1 <- tibble(
  nums = c(4, 5, 6),
  chars = c("hej", "你好", "こんにちは"),
  cplxnums = c("4+8i", "3+5i", "3+4i")
)
my_tibble_1
```

```
## # A tibble: 3 x 3
##   nums chars      cplxnums
##   <dbl> <chr>      <chr>
```

¹<https://thomaswdinsmore.com/2014/12/15/sas-versus-r-part-two/>

```
## 1      4 hej      4+8i
## 2      5 你好      3+5i
## 3      6 こんにちは 3+4i
```

类似地, 可以从现有的 vector 创建。所有的变量长度必须一样。

```
x <- c(1,4,5)
y <- c(211,23,45)
z <- c(20,32)
```

```
my_tibble_2 <- tibble(v1 = x, v2 = y)
my_tibble_2
```

```
## # A tibble: 3 x 2
##       v1     v2
##   <dbl> <dbl>
## 1     1    211
## 2     4     23
## 3     5     45
```

而试图把 x 和 z 做成 tibble 就会报错:

```
my_tibble_3 <- tibble(w1 = x, w2 = z)
```

```
# Error: Tibble columns must have consistent lengths, only values of length one are r
```

6.2.1.2 把 dataframe 转换成一个 tibble

```
d1 <- as.tibble(d) # 其中 d 是一个 dataframe
```

6.2.1.3 从外部数据创建 tibble

参见第8.2.1节 (数据的导入)

6.2.2 取子集 (抓取行, 列)

6.2.2.1 抓取单列

抓取单列很简单, 也很常用 (比如我们只想从一个大的 tibble 中抓两个变量研究它们之间的关系)。有两个符号可以用于抓取列, \$ (仅用于变量名称) 与 [[]] (变量名称或索引)。还是以 mpg 为例, 假设我们要抓取第 3 列 (displ):

```
#####
# 通过变量名称抓取:
mpg[["displ"]]
```



```
# 或
mpg$displ # 一般, 在 RStudio 中此方法最方便, 因为打出 “$” 之后会自动提示变量名。

#####
# 通过索引抓取:
mpg[[3]]
```

以上三种方法都应得到同样的结果 (是一个 vector):

```
## [1] 1.8 1.8 2.0 2.0 2.8 2.8 3.1 1.8 1.8 2.0 2.0 2.8 2.8 3.1 3.1 2.8 3.1
## [18] 4.2 5.3 5.3
```

如果使用单方括号, 得到的是一个 tibble (试试 `mpg[3]`) 这个特性在第6.3.1.3节中有解释。

6.2.2.2 抓取多列

有时候, 一个 tibble 中含有很多冗余信息, 我们可能想把感兴趣的几个变量抓出来做一个新 tibble. 这时 `select()` 函数最为方便。可以用变量名称或者索引来抓取。比如:

```
mpg_new <- select(mpg, 3:5, 8, 9)
# 等同于
mpg_new <- select(mpg, displ, year, cyl, cty, hwy)

mpg_new
```

```
## # A tibble: 234 x 5
##   displ year   cyl   cty   hwy
##   <dbl> <int> <int> <int> <int>
## 1  1.8  1999     4    18    29
## 2  1.8  1999     4    21    29
## 3  2    2008     4    20    31
## 4  2    2008     4    21    30
## 5  2.8  1999     6    16    26
## 6  2.8  1999     6    18    26
## 7  3.1  2008     6    18    27
## 8  1.8  1999     4    18    26
## 9  1.8  1999     4    16    25
## 10 2    2008     4    20    28
## # ... with 224 more rows
```

base

6.2.2.3 通过 `filter()`, 抓取满足某条件的行

通过 `filter()`, 我们可以过滤出某个或多个变量满足某种条件的 observations. 如果你还不熟悉逻辑运算, 请看第5.4.2节

假设我们只想看 mpg 中的奥迪品牌的, 排量大于等于 2 且小于 4 的车辆的数据:

```
mpg_audi_displ2to4 <- filter(mpg, manufacturer == "audi", displ >= 2.5 & displ < 4)
```

```
mpg_audi_displ2to4
```

```
## # A tibble: 9 x 11
##   manufacturer model  displ  year  cyl trans drv   cty   hwy fl   class
##   <chr>          <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
## 1 audi          a4      2.8  1999    6 auto~ f     16    26 p   comp~
## 2 audi          a4      2.8  1999    6 manu~ f     18    26 p   comp~
## 3 audi          a4      3.1  2008    6 auto~ f     18    27 p   comp~
## 4 audi          a4 qu~  2.8  1999    6 auto~ 4     15    25 p   comp~
## 5 audi          a4 qu~  2.8  1999    6 manu~ 4     17    25 p   comp~
## 6 audi          a4 qu~  3.1  2008    6 auto~ 4     17    25 p   comp~
## 7 audi          a4 qu~  3.1  2008    6 manu~ 4     15    25 p   comp~
## 8 audi          a6 qu~  2.8  1999    6 auto~ 4     15    24 p   mids~
## 9 audi          a6 qu~  3.1  2008    6 auto~ 4     17    25 p   mids~
```

6.2.2.4 用 slice(), 通过行数 (索引) 抓取行。

```
mpg_1to6 <- slice(mpg, 21:26) # 抓取 mpg 的第 21 行至 26 行
mpg_1to6
```

```
## # A tibble: 6 x 11
##   manufacturer model  displ  year  cyl trans drv   cty   hwy fl   class
##   <chr>          <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
## 1 chevrolet    c1500~  5.3  2008    8 auto~ r     14    20 r   suv
## 2 chevrolet    c1500~  5.7  1999    8 auto~ r     13    17 r   suv
## 3 chevrolet    c1500~  6    2008    8 auto~ r     12    17 r   suv
## 4 chevrolet    corve~  5.7  1999    8 manu~ r     16    26 p   2sea~
## 5 chevrolet    corve~  5.7  1999    8 auto~ r     15    23 p   2sea~
## 6 chevrolet    corve~  6.2  2008    8 manu~ r     16    26 p   2sea~
```

我觉得 slice() 更实际的用途是随机选择个体:

```
mpg_random4 <- slice(mpg, sample(length(mpg[[1]]), 4)) # 随机四辆车
mpg_random4
```

```
## # A tibble: 4 x 11
##   manufacturer model  displ  year  cyl trans drv   cty   hwy fl   class
##   <chr>          <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
## 1 dodge        duran~  5.2  1999    8 auto~ 4     11    16 r   suv
## 2 ford         f150 ~  5.4  1999    8 auto~ 4     11    15 r   pick~
## 3 pontiac      grand~  3.8  1999    6 auto~ f     17    27 r   mids~
## 4 dodge        carav~  3.3  1999    6 auto~ f     16    22 r   mini~
```

6.2.3 关于 Factor

有时候, 我们的变量是以文字的形式呈现, 但是它们不是单纯的文字, 而是有大小的差别, 或是能以一定顺序排列, 比如十二个月份 (Jan, Feb, ...), 成绩的“优、良、中、差”, 衣服的尺寸 (XS, S, M, XL, ...). 假设我们在做客户满意度调查, 七位客户的反馈是

```
满意度 _v <- c(" 满意", " 非常满意", " 满意", " 不满意", " 满意", " 非常不满", " 不满意")
```

我们试图用 `sort()` 把七个反馈按满意度从小到大排列:

```
sort(满意度 _v)
```

```
## [1] "不满意" "不满意" "满意" "满意" "满意" "非常不满"
## [7] "非常满意"
```

可见其排序并不是有意义的。(因为默认英语根据'abcde...' 排序, 中文根据笔画排序)

我们可以把这个 vector 做成 factor, 并用参数 `levels` 规定排序顺序:

```
# 按照惯例, 小的值在前, 大的在后; “非常不满”应为满意度最低的值。
满意度 _f <- factor(满意度 _v, levels = c(" 非常不满", " 不满意", " 满意", " 非常满意"))
sort(满意度 _f)
```

```
## [1] 非常不满 不满意 不满意 满意 满意 满意 非常满意
## Levels: 非常不满 不满意 满意 非常满意
```

这样排序就是正确的了。

```
class(满意度 _f) # "factor"
is.vector(满意度 _f) # FALSE
```

6.3 进阶内容: list, array 和 matrix

这一节为进阶内容, 不用看。可以直接跳到下一章 (第7

其中的很多操作和 dataframe 或 tibble 中的操作是等效的。一般, tibble 中的操作更直观, 更容易上手。

6.3.1 list (列表) 简介

R 中的 list 是一种特殊的数据存储形式。使用 `list()` 函数来创建 lists.

尝试对 lists 和 vectors 使用 `is.vector()`, `is.list()`, `is.atomic()` 和 `is.recursive()` 函数, 你会发现 list 虽然也是“vector”, 但我们一般说的“vector”都是指只能存储一种数据类型的 atomic vector; 而 lists 是 recursive vector.

这意味着一个 `list` 能存储多种类型的数据, 且可以包含子 `list`。list 中的每个元素可以是任何 `R` 中的物件 (object): 除了常用的 (atomic) vector 和另外一个 (子) list 以外, 还可以有 dataframe/tibble 和函数:

```
y <- list(1, c("a", "あ"), list(1+3i, c(FALSE, NA, TRUE)),
          tibble(x = c("阿拉木图", "什切青"), y = c(2, 3)),
          t.test)
```

```
y
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a" "あ"
##
## [[3]]
## [[3]][[1]]
## [1] 1+3i
##
## [[3]][[2]]
## [1] FALSE    NA    TRUE
##
##
## [[4]]
## # A tibble: 2 x 2
##       x           y
##   <chr>     <dbl>
## 1 阿拉木图     2
## 2 什切青     3
##
## [[5]]
## function (x, ...)
## UseMethod("t.test")
## <bytecode: 0x7fcfae8c8020>
## <environment: namespace:stats>
```

6.3.1.1 list 的索引/取子集

使用上面的例子:

```
y[2] # 使用单方括号, 得到的是一个只有一个 list 元素的 list
```

```
## [[1]]
## [1] "a" "あ"
```

```
y[[2]] # 使用双方括号, 得到的是一个 vector
```

```
## [1] "a"  "あ"
y[[3]][[2]] # 得到的也是一个 vector; 母 list 的索引在前, 子 list 的在后

## [1] FALSE    NA    TRUE
y[[3]] # 这个位置包含两个子 list, 因此得到一个有两个 list 元素的 list

## [[1]]
## [1] 1+3i
##
## [[2]]
## [1] FALSE    NA    TRUE
y[[3]][[2]][2] # 得到 vector 时, 直接在后面用单方括号

## [1] NA

list 里的元素可以有名字; 被命名的元素可以通过 $ 符号抓取:
z <- list(c(1, 3), z2 = c(4, 5, 6), c("a", "b"))
z # `[[2]]` 被 `$z2` 所取代

## [[1]]
## [1] 1 3
##
## $z2
## [1] 4 5 6
##
## [[3]]
## [1] "a" "b"

z$z2 == z[[2]] # `z[[2]]` 仍然是可用的, 结果和 `z$z2` 一样

## [1] TRUE TRUE TRUE
```

6.3.1.2 Dataframe 和 tibble 的本质

聪明的你也许已经注意到了, *dataframe*/*tibble* 抓取单列的方法和 *list* 的取子集惊人地相似 (我曾经的语文老师喜欢调侃抄作业的同学的答案与参考答案“惊人地相似”)。

事实上, *dataframe* 的本质正是 *list*, 而 *tibble* 也是 *dataframe* (只是进化了一些功能):

```
is.list(mpg)

## [1] TRUE

class(mpg)

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

6.3.1.3 Dataframe/tibble 的取子集

Tibble 既有 list 的特征, 也有 matrix 的特征。

当使用一个参数取子集的时候, 比如 `mpg[[3]]`, `mpg[["displ"]]` 或 `mpg$displ`, tibble 表现得像 list, 其中每一列是一个有命名的 list element;

当使用两个参数取子集的时候, 比如 `mpg[3,4]`, `mpg[3,]`, `mpg[,4]`, tibble 表现得像 matrix

```
mpg[3, ]
```

```
## # A tibble: 1 x 11
##   manufacturer model displ  year   cyl trans  drv      cty   hwy fl      class
##   <chr>          <chr> <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
## 1 audi          a4      2  2008     4 manua~ f      20    31 p      comp~
```

请查看Advanced R了解更多。

6.3.2 Base R dataframe 和 Tidyverse tibble 的区别

你可能已经注意到, 上面 `mpg[3,]` 和 `select(mpg, 3)` 是等效的; `mpg[3,]` 显然更精简, 那么为什么要使用 `select()` 函数呢?

6.3.3 array (数组) 和 matrix (矩阵) 简介

Vector 是一维的数据。Array 是多维的数据。Matrix 是二维的数据, 因此 matrix 是 array 的一种特殊情况。

Dataframe 不是 matrix (虽然都是方的)。Matrix 是二维的, 仅包含数字的 array。Dataframe 是一个二维的 list, 不同列 (即 list 元素) 可以存储不同的数据类型。

我们可以用 `dim()` 来创建 arrays:

```
A <- 1:48 # 创建一个 (1,2,3,...24) 的 numeric vector
dim(A) <- c(6,8) # 给 A assign 一个 6 乘 8 的 dimensions
A
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    1    7   13   19   25   31   37   43
## [2,]    2    8   14   20   26   32   38   44
## [3,]    3    9   15   21   27   33   39   45
## [4,]    4   10   16   22   28   34   40   46
## [5,]    5   11   17   23   29   35   41   47
## [6,]    6   12   18   24   30   36   42   48
```

可以看到我们创建了一个二维的, array, 因此它也是一个 (4 行 6 列的) matrix。

```
is.array(A)
```

```
## [1] TRUE
```

```
is.matrix(A)
```

```
## [1] TRUE
```

注意 24 个数字排列的方式。第一个维度是行，所以先把 4 行排满，随后再使用下一个维度(列)，使用第 2 列继续排 4 行，就像数字一样，(十进制中)先把个位从零数到 9，再使用第二个位数(十位)，以此类推。下面三维和四维的例子可能会更清晰。

同时注意最左边和最上边的 [1,], [,3] 之类的标记。你应该猜出来了，这些是 index。假设你要抓取第五行第三列的数值：

```
A[5,3]
```

```
## [1] 17
```

或者第三行的全部数值：

```
A[3,]
```

```
## [1] 3 9 15 21 27 33 39 45
```

或者第四列的全部数值：

```
A[,4]
```

```
## [1] 19 20 21 22 23 24
```

接下来我们再看一个三维的例子（还是用 1-48）：

```
dim(A) <- c(2,8,3)
```

```
A
```

```
## , , 1
```

```
##
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
```

```
## [1,]    1    3    5    7    9   11   13   15
```

```
## [2,]    2    4    6    8   10   12   14   16
```

```
##
```

```
## , , 2
```

```
##
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
```

```
## [1,]   17   19   21   23   25   27   29   31
```

```
## [2,]   18   20   22   24   26   28   30   32
```

```
##
```

```
## , , 3
```

```
##
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
```

```
## [1,]   33   35   37   39   41   43   45   47
```

```
## [2,] 34 36 38 40 42 44 46 48
```

它生成了三个二维的矩阵。在每个 2*8 的矩阵存储满 16 个元素后，第三个维度就要加一了。每个矩阵开头的，，x 正是第三个维度的值。同理，我们可以生成四维的 array：

```
dim(A) <- c(3,4,2,2)
```

```
A
```

```
## , , 1, 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
##
## , , 2, 1
##
##      [,1] [,2] [,3] [,4]
## [1,]   13   16   19   22
## [2,]   14   17   20   23
## [3,]   15   18   21   24
##
## , , 1, 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   25   28   31   34
## [2,]   26   29   32   35
## [3,]   27   30   33   36
##
## , , 2, 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   37   40   43   46
## [2,]   38   41   44   47
## [3,]   39   42   45   48
```

观察每个矩阵开头的，，x，y. x 是第三个维度，y 是第四个维度。每个二维矩阵存满后，第三个维度 (x) 加一。x 达到上限后，第四个维度 (y) 再加一。

类似二维矩阵，你可以通过 index 任意抓取数据，比如：

```
A[,3, , ] # 每个矩阵第 3 列的数据，即所有第二个维度为 3 的数值
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    7   19
## [2,]    8   20
## [3,]    9   21
```



```
##
## , , 2
##
##      [,1] [,2]
## [1,]   31  43
## [2,]   32  44
## [3,]   33  45
```

6.3.4 给 *matrices* 和 *arrays* 命名

假设我们记录了 3 种药物 (chloroquine, artemisinin, doxycycline) 对 5 种疟原虫 (*P. falciparum*, *P. malariae*, *P. ovale*, *P. vivax*, *P. knowlesi*) 的疗效, 其中每个药物对每种疟原虫做 6 次实验。为了记录数据, 我们可以做 3 个 6*5 的矩阵: (这里只是举例子, 用的是随机生成的数字)

```
B <- runif(90, 0, 1) # 从均匀分布中取 100 个 0 到 1 之间的数
dim(B) <- c(6, 5, 3) # 注意顺序
B
```

```
## , , 1
##
##      [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 0.41434308 0.4711039 0.52351186 0.24397207 0.46256603
## [2,] 0.15769432 0.6805976 0.61237873 0.87682343 0.5571892
## [3,] 0.94001228 0.9190032 0.33773087 0.78464163 0.5326318
## [4,] 0.15331747 0.7203922 0.06009472 0.06504344 0.8711724
## [5,] 0.31464623 0.7447924 0.61182622 0.29551607 0.8208334
## [6,] 0.08620387 0.4059124 0.08478483 0.30622528 0.3560613
##
## , , 2
##
##      [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 0.30572878 0.7109056 0.8285097 0.1512264 0.1364098
## [2,] 0.58939743 0.7827361 0.9920996 0.6964091 0.8213966
## [3,] 0.78398957 0.2038322 0.6339607 0.3883223 0.6107262
## [4,] 0.83339878 0.3359570 0.5056570 0.8003802 0.2552703
## [5,] 0.58338802 0.9891593 0.5527622 0.9193629 0.6708516
## [6,] 0.02926021 0.8434040 0.7502204 0.1120765 0.5984930
##
## , , 3
##
##      [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 0.35012960 0.84427517 0.371783353 0.9731701 0.8114402
## [2,] 0.02750938 0.79724164 0.679812154 0.8952902 0.2046514
## [3,] 0.03439413 0.05004879 0.489619261 0.2140061 0.9397663
## [4,] 0.43359605 0.68806399 0.717347769 0.1830532 0.3205851
```

```
## [5,] 0.40261460 0.86628986 0.002737983 0.2492440 0.5619048
## [6,] 0.64570339 0.98411626 0.436186512 0.2754077 0.1139996
```

然后用 `dimnames()` 来命名:

```
dimnames(B) <- list(paste("trial.", 1:6), c('P. falciparum', 'P. malariae', 'P. ovale',
B
```

```
## , , chloroquine
##
##          P. falciparum P. malariae   P. ovale   P. vivax P. knowlesi
## trial. 1    0.41434308  0.4711039 0.52351186 0.24397207  0.4625603
## trial. 2    0.15769432  0.6805976 0.61237873 0.87682343  0.5571892
## trial. 3    0.94001228  0.9190032 0.33773087 0.78464163  0.5326318
## trial. 4    0.15331747  0.7203922 0.06009472 0.06504344  0.8711724
## trial. 5    0.31464623  0.7447924 0.61182622 0.29551607  0.8208334
## trial. 6    0.08620387  0.4059124 0.08478483 0.30622528  0.3560613
##
## , , artemisinin
##
##          P. falciparum P. malariae   P. ovale   P. vivax P. knowlesi
## trial. 1    0.30572878  0.7109056 0.8285097 0.1512264  0.1364098
## trial. 2    0.58939743  0.7827361 0.9920996 0.6964091  0.8213966
## trial. 3    0.78398957  0.2038322 0.6339607 0.3883223  0.6107262
## trial. 4    0.83339878  0.3359570 0.5056570 0.8003802  0.2552703
## trial. 5    0.58338802  0.9891593 0.5527622 0.9193629  0.6708516
## trial. 6    0.02926021  0.8434040 0.7502204 0.1120765  0.5984930
##
## , , doxycycline
##
##          P. falciparum P. malariae   P. ovale   P. vivax P. knowlesi
## trial. 1    0.35012960  0.84427517 0.371783353 0.9731701  0.8114402
## trial. 2    0.02750938  0.79724164 0.679812154 0.8952902  0.2046514
## trial. 3    0.03439413  0.05004879 0.489619261 0.2140061  0.9397663
## trial. 4    0.43359605  0.68806399 0.717347769 0.1830532  0.3205851
## trial. 5    0.40261460  0.86628986 0.002737983 0.2492440  0.5619048
## trial. 6    0.64570339  0.98411626 0.436186512 0.2754077  0.1139996
```

清清楚楚, 一目了然。

6.3.5 apply

```
apply(A, 1, sum)
```

```
## [1] 376 392 408
```

Chapter 7

使用 ggplot 绘图

若要了解更多, 请阅读 ggplot 开发者本人所编写的 *ggplot2: Elegant Graphics for Data Analysis*(Wickham 2015)。

7.1 哲理

7.2 基础

7.2.1 基本语法

7.2.2 图像类型

7.3 进阶

7.3.1 逐层作图

7.3.2 尺寸, 轴, 和图例

7.3.3 位置

7.3.4 背景/主题的修改

7.3.5 与 `ggplot` 编程

7.4 附: **Base R** 中的作图

Chapter 8

数据处理

8.1 把“untidy data”整成“tidy data”

8.1.1 “untidy data”的主要问题

8.1.2 解决方案

8.2 数据的导入和导出

8.2.1 导入

8.2.1.1 csv

8.2.1.2 excel

8.2.1.3 其它

8.2.2 导出

Chapter 9

与 Python 的联合使用

9.1 在 R 中使用 Python: `reticulate`

9.2 在 Python 中使用 R: `rpy`

9.3 Beaker Notebook

<https://decisionstats.com/2015/12/07/decisionstats-interview-scott-draves-beaker-notebook/>

Inspired by Jupyter, Beaker Notebook allows you to switch from one language in one code block to another language in another code block in a streamlined way to pass shared objects (data)

Bibliography

Coghlan, Avril. 2016. “A Little Book of R for Biomedical Statistics.”

Kernighan, Brian W., and Dennis M. Ritchie. 1988. *The C Programming Language*. 2nd ed. Prentice Hall.

R Core Team. 2019. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. <https://www.R-project.org/>.

RStudio Team. 2015. *RStudio: Integrated Development Environment for R*. Boston, MA: RStudio, Inc. <http://www.rstudio.com/>.

Wickham, Hadley. 2015. *Ggplot2: Elegant Graphics for Data Analysis*. Use R! Springer.

Ziemann, Mark, Yotam Eren, and Assam El-Osta. 2016. “Gene Name Errors Are Widespread in the Scientific Literature.” Journal Article. *Genome Biology* 17 (1): 177. <https://doi.org/10.1186/s13059-016-1044-7>.