R 与 tidyverse——数据分析入门

石天熠

2019-07-13

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 获取资源与帮助（重要!） . . . 11
- 1.3 RStudio 界面介绍，基本操作，和创建新项目 . . . 14
- 1.4 安装和使用 packages (包) . . . 22

2 向量，逻辑，循环和函数 **27**
- 2.1 向量的概念，操作和优越性 . . . 27
- 2.2 数据/对象类型 (Data/Object Types) . . . 34
- 2.3 数学表达和运算 . . . 36
- 2.4 逻辑 . . . 48
- 2.5 判断和循环（流程控制） . . . 51
- 2.6 函数 . . . 61

3 dataframe（数据框）和 tibble **79**
- 3.1 查看 dataframe/tibble 并了解它们的结构 . . . 79
- 3.2 tibble 的创建和基础操作 . . . 81
- 3.3 其它 . . . 85

4 使用 ggplot 绘图 **87**
- 4.1 哲理 . . . 88
- 4.2 基础 . . . 88
- 4.3 进阶 . . . 88
- 4.4 附: Base R 中的作图 . . . 88

5 数据处理 **89**
- 5.1 把 “untidy data” 整成 “tidy data” . . . 89
- 5.2 数据的导入和导出 . . . 89
- 5.3 字符串的处理 . . . 89
- 5.4 Factors . . . 93
- 5.5 日期和时间 . . . 94

6 与 Python 的联合使用 **95**
6.1 在 R 中使用 Python: `reticulate` 95
6.2 在 Python 中使用 R: `rpy` 95
6.3 Beaker Notebook 95

References **97**

欢迎

简介

本书为 R 和 tidyverse 的入门向教程。教学视频在 b 站 (还没开播)。附加资源在R-Tutorial-Resources Github 仓库。

本书有 Gitbook 版本 (https://tianyishi2001.github.io/r-and-tidyverse-book/) 和通过 XeLaTeX 排版的 PDF 版本。

Gitbook 版本使用说明

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。中文衬线体使用的是思源宋体。

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。首次使用需要注册。

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

本书的结构

Hadley Wickham 写 R for Data Science 的时候把绘图放在了第一章，随后再讲加减乘除和数据处理，他认为这样可以降低新人被劝退的概率。我虽然很喜欢他的书，但是我是一个比较保守的人，把所有我认为是基础的内容放在了前 6 章。

本书的每一章有基础部分和（相对）进阶部分；基础部分的段落中会有 "可酌情跳过进阶部分" 的提示。仅阅读基础部分即可学到最重要的知识；如学有余力可阅读进阶部分。

在本书你不会学到:

1. 详细的统计学方法。我本身数学很差，教这个是要谢罪的。
2. **Python (NumPy/SciPy)**。在数据挖掘/数据分析领域, Python 和 R 一样是我们的好伙伴，而且它们经常被联合使用。但是本书作为 R 的入门教程，应当专注于 R。

3. **SAS, SPSS, STATA** 等软件。它们是万恶的资本主义的邪恶产物。R 和 Python 代表着自由，真正的自由。

版权页

技术信息

本书以 R Markdown 格式 (http://rmarkdown.rstudio.com/) 在 RStudio (http://www.rstudio.com/ide/) 中编写。

knitr (http://yihui.name/knitr/) 和 pandoc (https://pandoc.org/) 把 `Rmd` 文件编译成 `html` 和 `tex`, XeLaTeX 将 `tex` 排版为 PDF; 这一系列操作是使用 bookdown (https://bookdown.org) 自动完成的。

本书的源码, Gitbook 和 PDF 版本的书保存在 https://github.com/TianyiShi2001/r-and-tidyverse-book/ , 其中 Gitbook 和 PDF 保存在`/docs/`目录下, 由 GitHub Pages 生成静态网页，通过 https://TianyiShi2001.github.io/r-and-tidyverse-book/ 访问。

编写本书使用的 R packages，和排版本书时 R 的 sessionInfo 显示如下:

```
utils::sessionInfo(c("tibble", "dplyr", "forcats", "ggplot2", "stringr", "tidyr", "rea
```

```
## R version 3.5.3 (2019-03-11)
## Platform: x86_64-apple-darwin15.6.0 (64-bit)
## Running under: macOS  10.15
##
## Matrix products: default
## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib
##
## locale:
## [1] en_GB.UTF-8/en_GB.UTF-8/en_GB.UTF-8/C/en_GB.UTF-8/en_GB.UTF-8
##
## attached base packages:
## character(0)
##
## other attached packages:
##  [1] tibble_2.1.3      dplyr_0.8.3       forcats_0.4.0
```

```
##  [4] ggplot2_3.2.0     stringr_1.4.0     tidyr_0.8.3
##  [7] readr_1.3.1       purrr_0.3.2       yaml_2.2.0
## [10] lubridate_1.7.4   rmarkdown_1.13.6  knitr_1.23
## [13] bookdown_0.11     doParallel_1.0.14 foreach_1.4.4
##
## loaded via a namespace (and not attached):
##  [1] tidyselect_0.2.5 xfun_0.8         haven_2.1.1      lattice_0.20-38
##  [5] vctrs_0.2.0      colorspace_1.4-1 generics_0.0.2   htmltools_0.3.6
##  [9] grDevices_3.5.3  utf8_1.1.4       rlang_0.4.0      pillar_1.4.2
## [13] glue_1.3.1       withr_2.1.2      tidyverse_1.2.1  modelr_0.1.4
## [17] readxl_1.3.1     munsell_0.5.0    gtable_0.3.0     cellranger_1.1.0
## [21] rvest_0.3.4      codetools_0.2-16 evaluate_0.14    parallel_3.5.3
## [25] fansi_0.4.0      broom_0.5.2      methods_3.5.3    Rcpp_1.0.1
## [29] scales_1.0.0     backports_1.1.4  jsonlite_1.6     png_0.1-7
## [33] stats_3.5.3      datasets_3.5.3   graphics_3.5.3   hms_0.4.2
## [37] digest_0.6.20    stringi_1.4.3    grid_3.5.3       cli_1.1.0
## [41] tools_3.5.3      magrittr_1.5     lazyeval_0.2.2   zeallot_0.1.0
## [45] crayon_1.3.4     pkgconfig_2.0.2  xml2_1.2.0       utils_3.5.3
## [49] assertthat_0.2.1 base_3.5.3       httr_1.4.0       rstudioapi_0.10
## [53] iterators_1.0.10 R6_2.4.0         nlme_3.1-140     compiler_3.5.3
```

Chapter 1

R 和 RStudio 介绍和安装教程

本章内容速览

第1.1节：对 R 和 RStudio 进行概念和功能介绍，并介绍安装方式。

第1.2节：很重要的一节。介绍了常用的帮助和学习资源获取方式。

第1.3节：带你快速熟悉 RStudio 界面和基本操作。

第1.4节：也很重要。介绍 packages（包）的概念和功能，并引导安装本书需要使用的 packages.

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）。

不要看到编程就害怕。使用 **R** 不需要懂编程。R 语言最重要的特性之一就是，不懂编程的人可以轻松地用 R 自带的和和其他人编写的 packages，实现 99.9% 他们想要的功能（主要是数据分析）；而懂编程的人可以轻松地使用编程，在 R 中实现他们想要的剩余的 0.1% 的功能。同时，R 的编程语言非常简单易学，尤其是对于编程 0 基础的 R 使用者。像 SAS, STATA 这些商业软件，只能实现你 95% 的需求，且剩下的 5% 很难解决。

安装了 R 之后，你可以在其自带的 "R" 软件中使用（也可以直接在命令行使用），但是那个软件对新手的友好度不如 RStudio. RStudio (RStudio Team 2015) 是广受欢迎的 R 语言 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。

了解更多 R 的优势，请看第1.1.2节

1.1.1 安装 R 和 RStudio

1.1.1.1 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)

1.1.1.2 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。

1.1.2 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 是 GNU 计划的一部分，因此 R 是一个自由软件 (Libre software)。它不仅免费，还允许用户自由地学习，运行该软件；拷贝，分发，修改并改进该软件，以帮助其他人。你可以在GNU 官网了解更多。

R 比各种商业统计软件功能更强大。没错，**免费的 R 比昂贵的商业软件功能更强大**。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 有巨大的用户社群[3]，其中有很多热心的使用者/开发者在论坛上解答问题，或是编写免费获取的教程。SAS 等软件虽然有客户支持，但是如果你用的是盗版.....

R（RStudio）非常稳定。闪退率极低，而且就算闪退了，也完全不会丢失上一次工作中的数据，可以无缝衔接上一次的工作。我经常会创建一两个实验用的 R script 文件，我不需要把它们命名并保存在我的工作目录，重启 RStudio 的时候仍然可以使用它们。总之，关闭 RStudio 的时候，你甚至可以什么都不用保存；关闭，重启，无忧无虑地继续工作。设置 Git 后体验更佳。

R 与其它编程语言/数据库之间有很好的接口。

Python（NumPy 和 SciPy）是近几年兴起的数据分析处理方案。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

[1] Gentleman, R. (2009). *R Programming for Bioinformatics*. Boca Raton, FL: CRC Press.
[2] https://thomaswdinsmore.com/2014/12/15/sas-versus-r-part-two/
[3] https://blog.revolutionanalytics.com/2014/04/a-world-map-of-r-user-activity.html

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6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，数据分析的严谨性，大数据的处理能力，和功能的拓展非常局限。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)。不是说不能用 Excel（或者其它可用的工具），而是要清楚各种工具的优势和局限，物尽其用。比如当需要从 PDF 文件中提取表格数据时，我会把它们复制到 Excel（因为兼容性强）；我也会用 Excel 做一些数据的初步处理，比如删除数量不多的冗余的行和列，重命名变量名等。

虽然 R 是自由软件，但是我们要记得感激所有位 R 贡献智慧的奉献者。出于对知识劳动的尊重和，以及保持 R 的发展壮大，我呼吁有能力出资的使用者在 https://www.r-project.org/foundation/donations.html 对 R 进行捐赠。

1.2 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:

1.2.1 核心/入门资源

1.2.1.1 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（但是客观地来说活跃度还是没爆栈网高）同样不要忘记读新手指引。

1.2.1.2 Reference 类（查找特定的 function/package 的用法）

- 直接在 R console 中执行`?`+ 函数名称或者 `package` 名或者其它，比如`?t.test`，可以查看对应函数的帮助文档（documentation）有一些函数/packages/内容名需要加上引号，比如`?"+"`，`?"if"`。有一个相似的方法，`??`+`" 内容"` 可以根据你输入的内容搜索帮助文档，比如`??"probability distrubution"`。
- 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，相比于函数的帮助文档更为详细且更易读。`vignette()`（无参数）以查看全部可用 vignettes. 试试 `vignette("Sweave")`。

1.2.1.3 教程和书籍类（用来系统地学习）

- *R for Data Science* by Garrett Grolemund & 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）
- *R for Beginners* by Emmanuel Paradis及其中文译本
- R 的官方 Manuals. 是一组严谨，全面但略微枯燥的文档，可能不太适合零基础的新手，但是对于精通 R 有很大的帮助。部分由丁国徽翻译成中文。
- 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.
- 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）

1.2.1.4 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。

1.2.2 进阶资源

1.2.2.1 较为深入

- *The R Book* by Michael J. Crawley
- *Advanced* R by Hadley Wickham及其练习题答案。

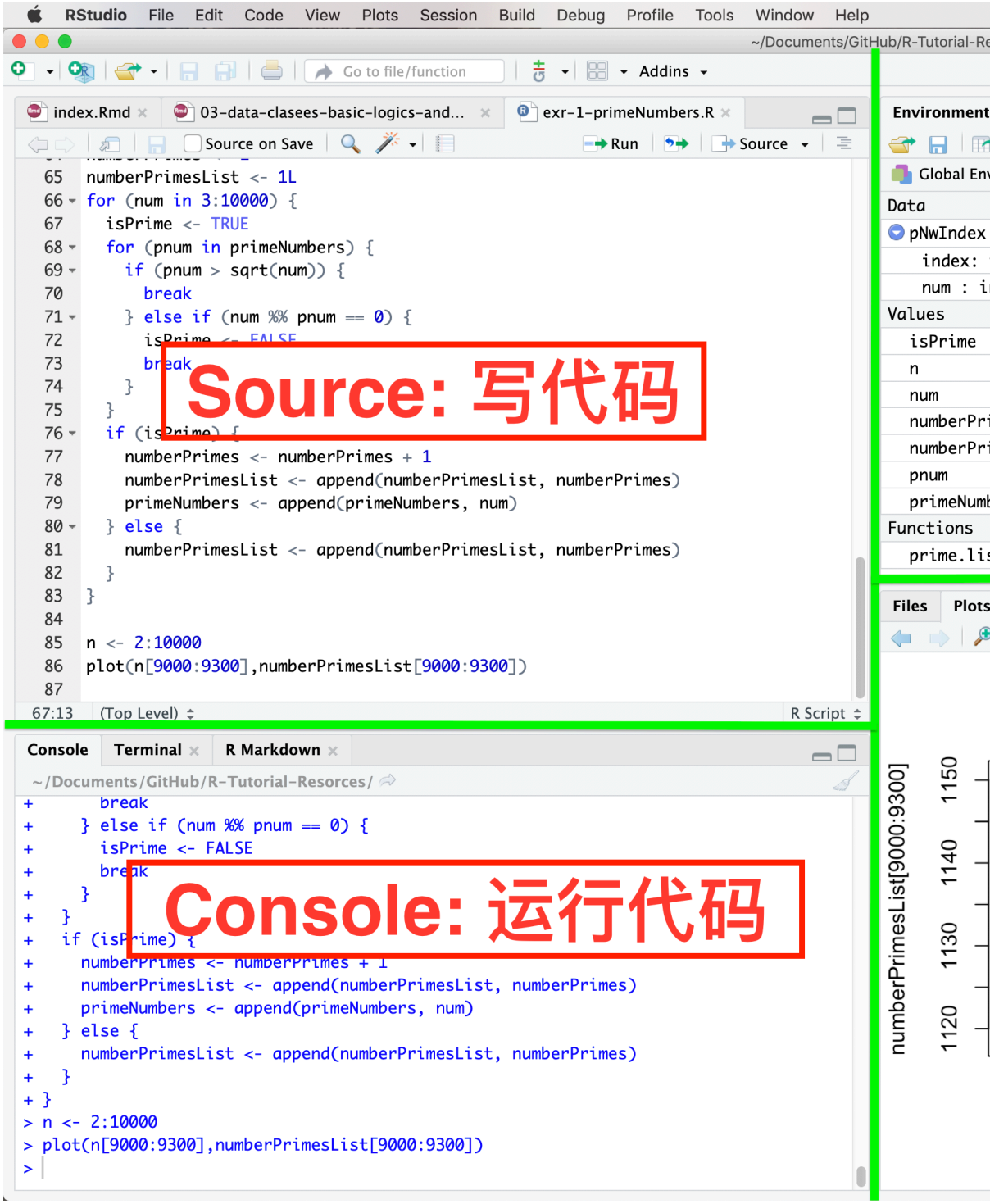
1.2.2.2 真·老司机

- Springer 的*Use R!* 系列。

1.3 RStudio 界面介绍，基本操作，和创建新项目

1.3.1 界面

1.3.1.1 概览



1.3.1.2 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。

1.3.1.3 左上角：**Source**（源）

Source 是写代码的地方。请看第1.3.3.3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")`.

1.3.1.4 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对象，包括变量（"values" 和 "data"）和自定义的函数（functions），并简要显示了它们对应的值。

History（历史）和 Connections（连接）不太常使用。

1.3.1.5 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1.4章。

1.3.2 执行代码

1.3.2.1 计算和赋值

我本来不想在开篇就写一小节使用较多的术语的文字，但是 R 中计算和赋值的概念实在太重要了，我不得不把它放在这里。

几乎所有 R 中的指令可以归为两种。计算 (evaluation) 或者赋值 (assignment).[4]

没有 `<-`符号的为计算，有 `<-`符号的为赋值。

大多数情况下，计算仅仅会产生效果（或是在 console 输出结果，或是在 plot 区产生图像，或是在工作目录新建一个 pdf 文件），赋值会且仅会改变一个对象（变量）的值（包括新建一个对象），并且不会产生其他的效果。[5]

[4]准确地说，赋值也是一种计算。赋值符号本身就是一个函数，你可以用`"<-"(x, 5)` 把 `5` 赋值给 `x`。

[5]有一些计算不支持赋值。当强行赋值的时候，会产生效果，但赋值的值为 `NULL`，比如 `x <- pdf()` 会在工作目录新建一个 pdf，并新建了对象 `x`，但 x 的值为 `NULL`. 有一些计算支持赋值，但是同时也会产生效果，比如 `hist(rnorm(1000))`（以 list 的形式赋值）。

首先我们来做一个计算。

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2.1.2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，这意味着你可以很方便地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 `>`）。

再执行以下指令（在 RStudio 中，可以用 `Alt`+`-` (Mac 是 `option`+`-`) 这个快捷键打出这个符号。）：

```
x <- 5*5+1
```

这是一个**赋值**指令。计算结果不会显示，但是你新建了一个名为 `x` 的变量（准确地说，是“对象”），值为 `5*5+1` 的计算结果，即 `26`. 你可以执行 `x` 来查看 `x` 的值：

```
x
```

```
## [1] 26
```

像一个小箭头的赋值符号（`<-`）的作用是[6]，首先计算出其右边的指令（必须是一个计算指令；即同一条指令不可以出现两个 `<-`符号），然后把**计算结果的值**作为一个**拷贝**赋予给左边的名字，这样就新建了一个对象 (object)。每个对象有一个名称和一个值。[7]左右是很重要的；绝大多数其他的编程语言，虽然赋值符号是 `=`，但也是从右往左赋值，R 使用 `<-`作为赋值符号更形象，避免新手写出像 `5 = x` 之类的指令。当然，如果你喜欢，也可以在 R 中使用 `=`。[8]

`<-`用于给任何对象赋值，包括常用的向量 (vector)，列表 (list)，数据框 (dataframe) 和函数 (function)。

谨记，赋值符号只是把右边的**计算结果**作为一个**拷贝**赋予给左边，而不会做任何其它的事情[9]。变化的仅仅是左边的变量（对象），右边的计算中所用到的任何变量（对象）不会改变！

为什么强调是一个拷贝呢？举个例子，我们现在把 `x` 的值赋予给 `y`，不出所料，`y` 的值将为 `26`。那么要是我们在这之后重定义 `x` 为 `40`，y 的值是多少呢？

```
y <- x
x <- 40
y
```

[6]其实你还可以把这个小箭头反过来，试试 `5 -> x`. 但是不建议这么做。代码易读性会变差。

[7]每个对象还可以有一些（可选的）attributes（属性）。

[8]其实可以用 `=` 替代 `<-`作为赋值符号，但是更多的 R 用户还是采用传统的的 `<-`符号，而 `=` 则用于给函数的参数赋值。这种区分可以使代码可读性更强（更容易看出哪些语句是赋值，哪些是计算）。当然，如果你真的非常非常想用 `=` 符号，也是可以接受的。

[9]一个特例是 environment（环境）的赋值。初学者不需要知道。

```
## [1] 26
```

还是 26（而不是 40）。赋值是一次性的，每次被赋值的对象都将成为独立自主的个体。对象 y 虽然在被赋值的时候需要用到对象 x，但是在那之后 y 和 x 半毛钱关系都没有了（除非再次赋值），所以 x 的变化不会影响 y，y 的变化也不会影响 x。

所有的变化，只可能发生在赋值。

1.3.2.2 计算和函数

所有的计算都是通过函数实现的，包括当你输入 `x` 然后按回车时。[10]像 `+`, `-`这样的运算符也是函数（参见第2.6.1节）。

函数的标志是小（圆）括号，比如 `sum(6, 7, 8)` 是求 6, 7 和 8 的和；其中 `sum()` 是函数，`6, 7, 8` 是（三个参数）。

函数可以嵌套使用，而且很常见。

```
prod(sqrt(sum(2, 3, 4)), 2, 5)
```

```
## [1] 30
```

最 “内部的” 函数先运行，然后把计算结果作为它外面的函数的参数。这里，`sum(2, 3, 4)` 得到 9, `sqrt(9)` 得到 3, `prod(3, 2, 5)` 得到 30. 就像小学的时候学的括号运算规则一样。

更多关于函数的知识请参阅第2.6节。

1.3.3 管理代码

1.3.3.1 创建 R Project

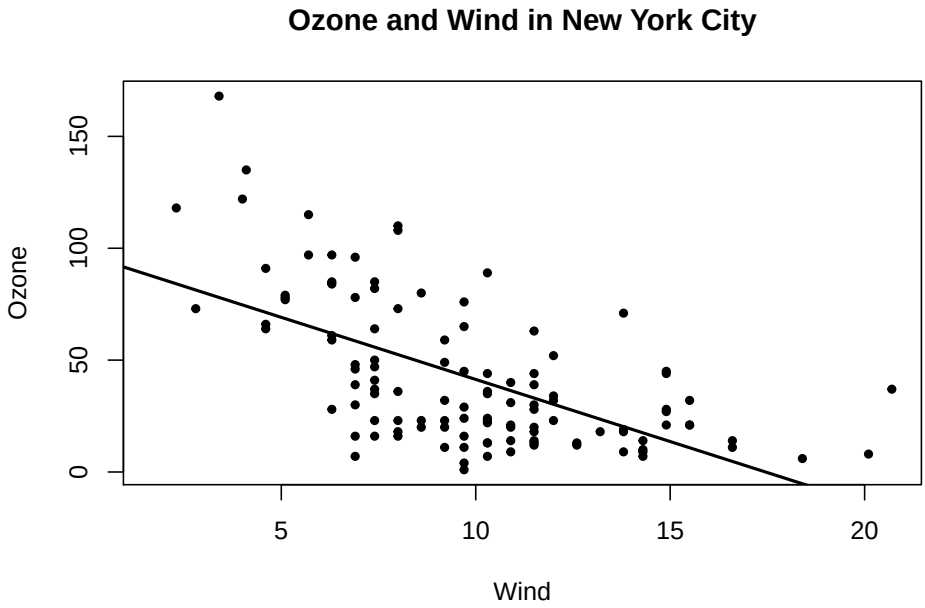
试着在 console 里输入（或者复制）以下代码并执行:

```
attach(airquality)
```

```
## The following objects are masked from airquality (pos = 3):
##
##     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)
```

[10]查看 `x` 的值，我们只需要输入 `x` 然后按回车，然后我们在 console 看到了 `x` 的值。这其实也是用函数实现的。当一个指令不是赋值时，R 默认会对整个指令使用 `print()` 函数。因此，`2 + 4` 等同于 `print(2 + 4)`, `x` 等同于 `print(x)`. 当一个指令是赋值是，R 默认会对右边的整个指令使用 `invisible()` 函数，因此，`y <- x` 等同于 `y <- invisible(x)`.



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。

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。

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

1.3.3.2 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（正规的说法是 “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。

1.3.3.3 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
```

点击你想执行的语句，按 Ctrl+Enter (command+return) 以执行那一“句”语句（比如上面的例子中，从 plot(Wind... 到 pch = 20) 有三行，但是它是一“句”），然后光标会跳至下一句开头。

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

通过 Ctrl（+Shift）+Enter 执行代码时，相关代码相当于是从 R script 中复制到了 console 并执行。

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。

1.3.4 RStudio 的额外福利 {rstudio-fuli}

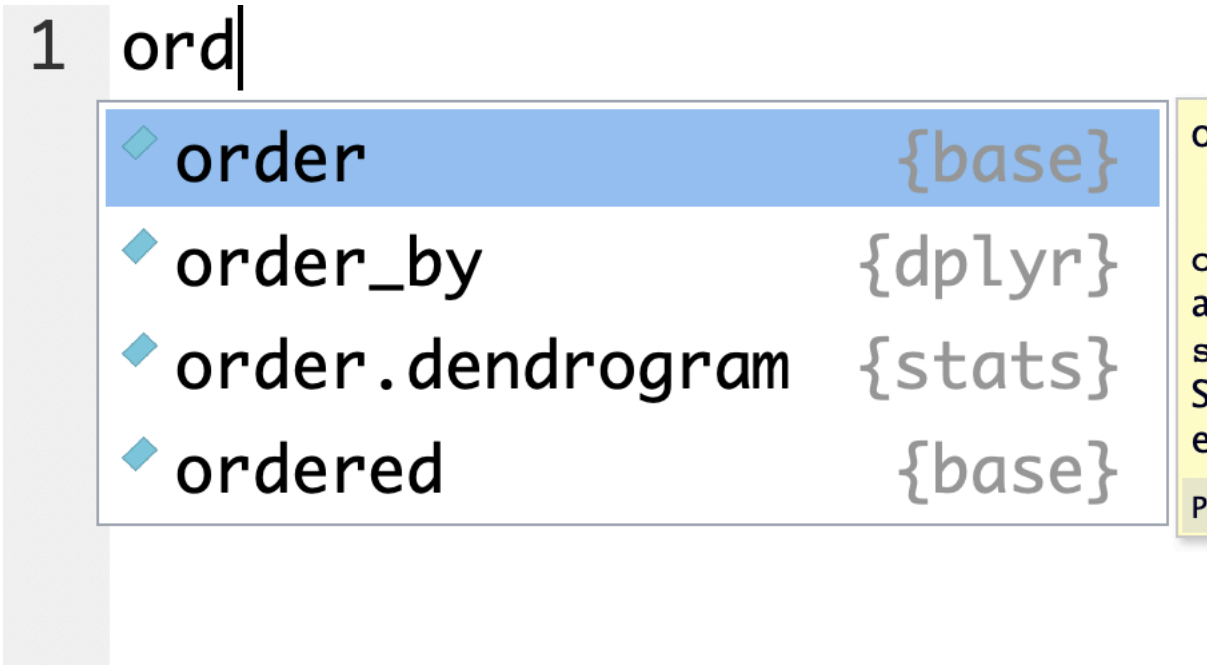
1.3.4.1 括号自动补齐；换行自动缩进

在 RStudio 中，除非你故意，否则很难出现括号不完整的错误。当你打出

1.3.4.2 自动完成/建议提示/快速帮助 (autocomplete)

当你在 console 或者 source 区输入三个[11]或更多字母时，R 会提示以这三个字母开头的所有对象（不一定是 packages 里的函数，也可以自定义的向量，列表，函数等等）。

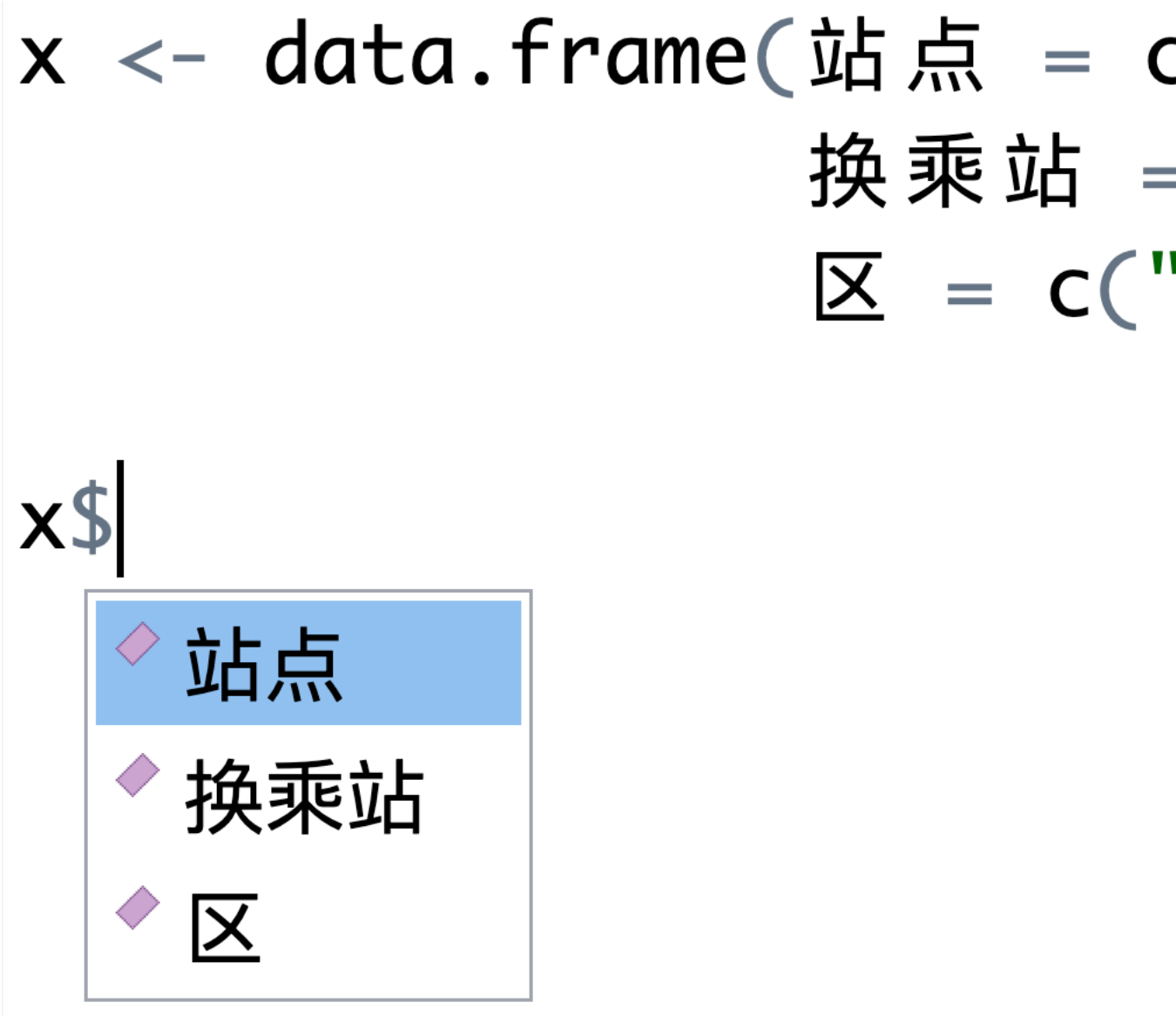
[11]可以在设置中，自定义所需输入的最少字母和延迟。默认分别为 3 个字母和 250 毫秒。



然后，你可以用键盘的 “上”，“下” 键去选择，然后按回车键完成（或者用鼠标点选）。如果对象是一个函数，会自动帮你补齐括号。

注意，当你选中一个函数时，右边会有一个黄色的方框，提示这个函数的参数名称和参数选项，以及一段简要说明。如果你这时想查看此函数的帮助文档，按 `F1` 即可。

通过 `$` 符号取子集的时候，R 会自动列举所有可用的子集（用于列表和数据框/tibble）。类似地，在 package 名后输入`::` 准备调用函数时，R 会列举该 package 所有可用函数（见第1.4.3.2节）。



1.3.5 其他

1.3.5.1 “一句”的概念

一次（一句）计算执行且执行一个函数（不包括里面嵌套着的函数）。

当你通过函数名 + (“开启”一次运算时，从这里开始到这个函数所对应的反括号，即）之前

的内容，即使再多，都只是这个函数的参数。

当你在 R script 里敲击 `Ctrl`+`Enter` 时，光标所在的位置的那一句指令将会被执行（无需在那一句的开头，可以在那一句之中的任何位置）；如果那句命令不完整，很可能会在 console 中用 `+` 提示（见下一小节）。

句与句之间必须换行，或者用分号（`;`）连接：

```
sum(1, 9)
sum(2, 3)
# 和
sum(1, 9); sum(2, 3)
# 都是可以的，而
sum(1, 9) sum(2, 3)
# 不可以，会出错
```

1.3.5.2 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。

1.4 安装和使用 packages (包)

1.4.1 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.

这个课程主要是学习 `tidyverse` 这个 package,

1.4.2 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1.4.4.2节。

1.4.3 如何使用 packages

1.4.3.1 加载 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1.4.4.1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）.[12]

1.4.3.2 使用 packages 里的内容

刚才加载 `tidyverse` 的时候，你也许注意到了这样一条提示:

[12]另一个主要原因是，寻找对象时，R 需要搜索所有已加载的 packages，而且，packages 都被加载在 RAM 里，因此加载过多的 packages 会使 R 显著变慢。(虽然有一些开挂的方法)

```
  Conflicts          tidyverse_conflicts()
dplyr::filter() masks stats::filter()
dplyr::lag()    masks stats::lag()
```

这是因为 R 本来自带了一个叫做 `stats` 的 package,有俩函数名曰 `filter()` 和 `lag()`，而 `dplyr`（`tidyverse` 的一部分）也有同名的俩函数，把原来的覆盖了。所以它提示你，当你使用 `filter()` 和 `lag()` 时，使用的是 `dplyr` 的版本，而不是原来 `stats` 里的。

这不意味着 `stats` 里的这两个函数就不能用了。要使用他们，用这个格式就好了：

```
stats::filter()
```

同样的道理也适用于其他的 packages. 你可以通过

```
dplyr::filter()
```

使用 `dplyr` 版本的 `filter()`。虽然这看起来是个好习惯，但是很少人这么做。Python 里每使用一次 NumPy 里的函数都要加上 `np_` 的前缀，虽然严谨，但是麻烦。R 的一大便利之处就是使用 packages 里的内容时，不强制要求指定 packages 的名称。如果函数/对象名称有重叠，以 packages 的加载顺序决定优先度；最近（即最后）被加载的 package 里的函数/对象胜出，而其余的要通过 `packageName::object` 的形式调取。

1.4.4 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2章）。

1.4.4.1 `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)
```

或者更精简一点，

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

1.4.4.2 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 2

向量，逻辑，循环和函数

本章内容速览

- 第2.1节介绍了 R 中向量的概念，使用方法和优越性。
  - 2.1.1：向量的创建（赋值）和合并
  - 2.1.2：向量的索引（indexing）和取子集（subsetting）
  - 2.1.3：生成有序数列（连续整数，重复数/重复向量，
  - 2.1.4：向量的其它操作
  - 2.1.5：向量的优越性——向量化计算概念基础
- 第2.2节介绍了 R 中的数据/对象类型
  - 2.2.1：如何查看数据/对象的类型；最基础的 5 种（atomic vector 所存储的）数据类型；其它常用数据/对象类型
  - 2.2.2：数据类型详解；更多的数据类型
- 第2.3节介绍了 R 中的数学规则
  - 2.3.1：数的表达；整数，浮点数，科学计数法
  - 2.3.2：基础的数学运算
  - 2.3.3：基础的统计学计算，包括 t 分布，t 检验，卡方检验
- 第??logical-operation) 节介绍了 R 中逻辑值 (`TRUE`, `FALSE`, `NA`) 的概念和玩法。

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]

2.1 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。

[1]如果一定要违反规则，可以使用转义符号\``，比如可以 ``4foo%b=a+r‘ <- 50 “

2.1.1 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
```

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
```

```
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
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2.1.1节）

还是通过 `c()` 函数，可以把多个向量拼接成一个大向量:

```
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] "    "
```

2.1.2 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。

2.1.3 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：

2.1.3.1 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。

当元素比较多的时候：

```
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**2.1.5**节。

2.1.3.2 复读机 rep()

```
rep(6, 8) # 把 6 重复 8 遍；或 rep(6, times = 8)
```

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

```
rep(c(0, 7, 6, 1), 4) # 把 (0, 7, 6, 1) 重复 4 遍
```

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

```
rep(c(0, 7, 6, 1), each = 4) # 把 0, 7, 6, 1 各重复 4 遍
```

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

```
rep(c(0, 7, 6, 1), c(1, 2, 3, 4)) # 把 0, 7, 6, 1 分别重复 1, 2, 3, 4 遍
```

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

想一想，rep(8:15, rep(1:5, rep(1:2, 2:3))) 的计算结果是什么？

2.1.3.3 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：

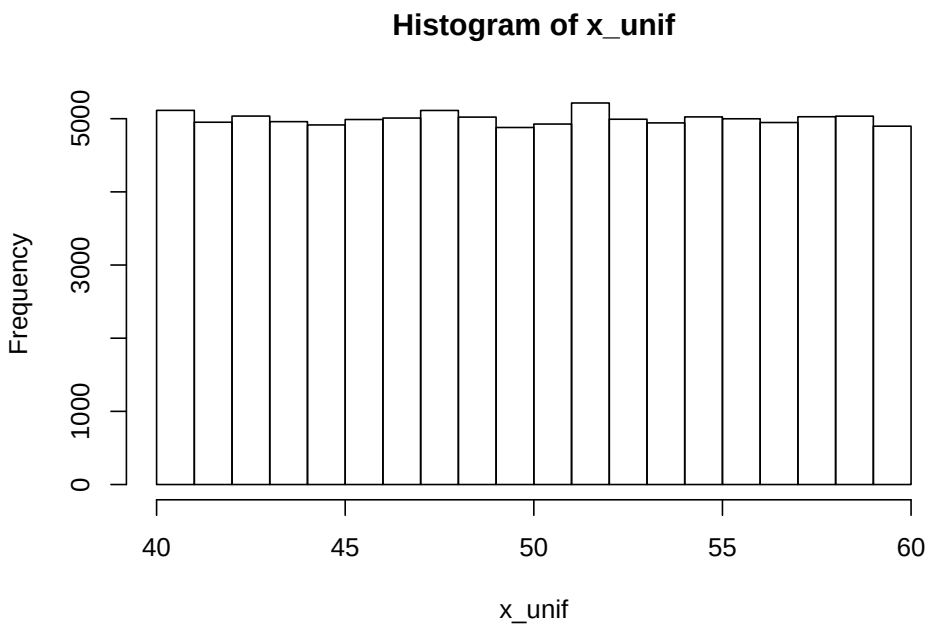
```
 seq(0, 50, length.out = 11) # 其实是 `seq(from = 0, to = 50, length.out = 11)`的简写
```

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

2.1.3.4 随机数：

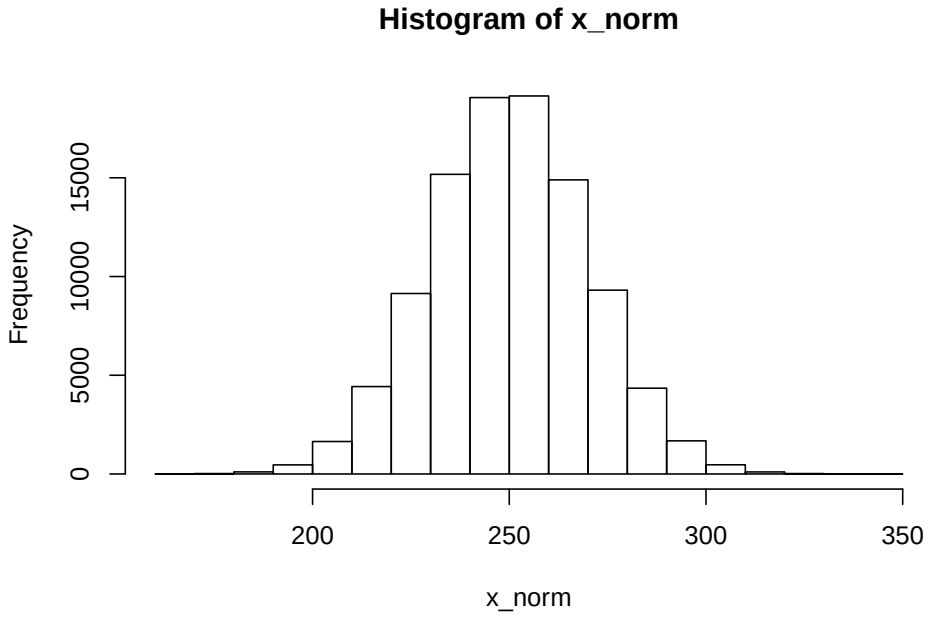
连续型均匀分布随机数用 runif(n, min, max)，n 是数量，min 是最小值，max 是最大值。默认 min 为 0，max 为 1。

```
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 100000
hist(x_norm) # 画直方图
```



2.1.4 向量的其他操作

2.1.4.1 创建长度为 0 的向量

使用循环的时候，经常需要初始化一个长度为 0 的向量（见第2.5节

有两种方法实现：

```
x <- vector("numeric")
# 或 `vector("integer")`, `vector("character")`等
class(x)
```

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

或者：

```
x <- integer(0)
# 或 x <- integer()
# 或 `character(0)`, `numeric(0)`等
class(x)
```

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

其中后面这种方法亦可用于创建长度为 n 的向量，把 0 替换成你想要的长度即可。

2.1.4.2 sort(), rank() 和 order()

```
x <- c(2, 5, 3, 6, 10, 9, 7, 8, 1, 4)
sort(x)
rank(x)
order(x)
rev(sort(x))
# 为方便同框展示，我用的代码是 list(x = x), `sort(x)` = sort(x), `rank(x)` = rank(x), `or
```

```
## $x
## [1] -10   5 -89 999  84
##
## $`sort(x)`
## [1] -89 -10   5  84 999
##
## $`rank(x)`
## [1] 2 3 1 5 4
##
## $`order(x)`
## [1] 3 1 2 5 4
##
## $`rev(sort(x))`
## [1] 999  84   5 -10 -89
```

`sort()` 很好理解,就是把原向量的元素从小到大重新排列。如果要从小到大:`rev(sort(x))`.

`rank()` 是原向量各个元素的（从小到大的）排名。（`-10` 是第 2 名，`5` 是第 3 名，`-89` 是第 1 名，以此类推）

`order()` 是一个原向量索引的排序, 使得 `x[order(x)] = sort(x)`, 即 `x[order(x)] = x[c(3, 1, 2, 5, 4)] = c(-89, -10, 5, 84, 999) = sort(x)`

至于文字向量，英文按 `a, b, c, d, e, ...` 排列，中文按笔画排列。

2.1.4.3 元素的命名

```
scores <- c(ochem = 79, math = 66, mcb = 64, blc = 75, bpc = 72)
scores
```

```
## ochem  math   mcb   blc   bpc
##    79    66    64    75    72
```

然后便可以额外地用名字抓取元素:

```
scores[c("math", "bpc")] == scores[c(2, 5)]
```

```
## math  bpc
## TRUE TRUE
```

2.1.5 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
```

拥有这种特性的计算也被称为向量化计算 (vectorized computation).

相比于常用的编程语言，向量化计算省去了 for 循环，计算效率得到极大的提升；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。

若想更多地了解向量化计算（比如如何把 for 循环需要 39 秒的运算压缩到 0.001 秒），请看第2.5.4节。

2.2 数据/对象类型 (Data/Object Types)

2.2.1 基础的数据/对象类型

2.2.1.1 向量所存储的数据类型

向量所存储的的数据类型有 5 种:

类型	含义与说明	例子
numeric	浮点数向量	`3`, `0.5`, `sqrt(2)`, `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`

一个向量的所有元素必须属于同一种类型。如果尝试把不同类型的元素合并成一个向量，其中一些元素的类型会被强制转换 (coerced)。你可以试试 `c(2, "a")`, `c(2+5i, 4)`, `c(TRUE, 1+9i)` 和 `c(TRUE, 1+9i, "a")`，但是实际操作的时候尽量不要这么做。

2.2.1.2 关于数据类型的简单操作

通过 `class()` 函数，可以查看数据/对象的类型。

```
class(6) # 6 是一个（浮点）数，应为"numeric"
```

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

通过 `is.XXX()` 函数，可以得到一个逻辑值，指明此数据/对象是否属于某个类型，`TRUE` 为是，`FALSE` 为否。比如:

```
is.numeric(6)
```

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

```
is.character("6")
```

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

通过 `as.XXX()` 函数，可以把数据/对象强行转换成另一种类型，比如:

```
as.integer(c(TRUE, FALSE))
```

```
## [1] 1 0
```

```
as.character(c(23, 90))
```

```
## [1] "23" "90"
```

2.2.1.3 NA, Inf, NaN 和 NULL

NA 为缺损值，意思是该元素所代表的数值丢失/不确定/不可用。举个例子，当我们统计学生的 200m 跑成绩时，有一些学生因为身体不适未能参与测试，这时他们的成绩应被记为 NA:

```
time_in_sec <- c(29.37, 28.66, 31.32, NA, 27.91, NA)
```

之前说过，一个向量中，所有的元素都是同一类型的。的确，这里的 NA 的类型是 numeric:

```
class(time_in_sec[4])
```

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

同理，character 向量里的 NA，类型也是 character，其他类型也是一样的道理。如果只是单个的 NA, 它的类型是 logical:

```
y <- c("a", "b", NA)
class(y[3])
```

```
## [1] "character"
```

```
class(NA)
```

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

Inf（无限）NaN（非数）的概念，以及作为 numeric 的 NA 的数学计算在第??小节讨论。

作为 logical 的 NA 的逻辑运算在第??logical-operation) 小节讨论。

NULL 是 “无”，真正的 “无”。它几乎一无是处，因此在此不作更多讨论。学有余力者可以自己去了解。

2.2.1.4 其它的数据/对象类型

- Dataframe/tibble 是 R 中存储复杂（多变量）数据的规范格式，从第??tibble) 章开始将一直占据我们话题的中心。
- 因子 (factor) 有很多向量的特性，尤其是能在 dataframe/tibble 中作为变量，但是它并不是向量；因子的详细内容在第5.4节。
- 函数 (function)。我们刚才用 c() 来创建向量，它就是一个函数：class(c)；函数的详细内容在第2.6节。
- list 类似于向量，但是一个 list 可以包含不同类型的元素。性质和使用方法也和向量大相径庭。详细内容在第??节，算是较为进阶的内容。
- 矩阵 (matrix) 和数组 (array) 可以算作是二维和多维的向量，同样只能存储一种类型的数据，详细内容在第2.6.7节，同样是较为进阶的内容。

2.2.2 数据类型（严谨版）

可以酌情跳到第2.3节。

2.2.2.1 class, type, mode 和 storage mode

其实 class 根本不是基础的数据类型。学过编程的应该猜到了，此 class 正是 OOP 里的 “类”，是 “高层” 的类型。你可以随意篡改 class:

```
x <- c("Joe", "Lynne", "Pat")
class(x) # 本应为"character"
```

```
## [1] "character"
```

```
class(x) <- c("high_school", "student") # 篡改
class(x) # 新 class
```

```
## [1] "high_school" "student"
```

用 typeof(), mode(), storage.mode() 所获取到的三种属性是不可篡改的 “底层” 类型。

2.3 数学表达和运算

2.3.1 数的表达

2.3.1.1 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"
```

2.3.1.2 科学计数法

在 R 中可以使用科学计数法 (AeB= $A \times 10^B$)，比如：

```
3.1e5
```

```
## [1] 310000
```

```
-1.2e-4+1.1e-5
```

```
## [1] -0.000109
```

2.3.1.3 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）。

2.3.1.3.1 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（**2.3.2**）。

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2.3.2.3并不会产生整数。如果需要的话，要用`as.integer()`函数。

2.3.2 运算

2.3.2.1 二元运算符号

R 中的 binary operators（二元运算符）有:

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

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)。

2.3.2.2 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2.6.2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
```

2.3.2.3 近似数（取整，取小数位，取有效数字）

取有效数字用 `signif()` 函数；第一个参数是对象，第二个参数是保留的位数；若保留的位数未指定，默认为 6.

```
signif(12.3456789, 4)
```

```
## [1] 12.35
```

当对象的有效数字小于你想保留的有效数字位数时，它不会让你乱来（下面 `round()` 函数也类似）：

```
signif(12.3, 8)
```

```
## [1] 12.3
```

保留小数位用 `round()` 函数。

```
round(12.3456789, 3) # 保留 3 个小数位
```

```
## [1] 12.346
```

若不指定保留多少位，默认为 0，即四舍五入地取整：

```
round(13.5)
```

```
## [1] 14
```

此外，还有三种取整函数：`floor()`, `ceiling()` 和 `trunc()`

```
floor(5.6) # = 5 # “地板”；比 x 小的最近的整数
ceiling(5.4) # = 6 # “天花板”；比 x 大的最近的整数
floor(-5.6) # = -6 # 不是-5，因为-6 是比-5.6 小的最近的整数
ceiling(-5.4) # = -5 # 不是-6；因为-5 是比 x 大的最近的整数
trunc(-5.6) # = -5 # 你可能需要这个；它无视了小数点后面的位数
```

注意，所有取整函数给出的的结果都并不是整数！

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

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

虽然浮点数使用起来真没啥不方便的，但是如果你一定需要的话，可以用 `as.integer()` 函数把它转换成真·整数。

2.3.2.4 NA, Inf, NaN 相关 {math-NA}

我不知道张三有几个苹果，我也不知道李四有几个苹果；你问我张三和李四共有几个苹果：

```
NA + NA
```

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

鬼才知道咧！

类似地，`NA - NA`, `NA/NA`, `NA*NA`, `log(NA)` 都等于 `NA`

`NA^0` 等于几？别上当！R 的开发者们可没有忘记 $\forall x \in \mathbb{R}: \curvearrowleft^{\curlywedge} = \curlywedge$

`Inf`，即 ∞，表示很大的数字（准确地说，大于等于 2^{1024} 即 1.797693×10^{308} 的数字）它还有个负值，`-Inf`. 以下是几个结果为 `Inf` 的例子：

```
exp(1000) # = Inf; 这个很明显
1/0 # = Inf; 0 被当作很小的数
0^(-1) # = 1/(0^1) = 1/0 = Inf
log(0) # = -Inf; 0 又被当作很小的数
```

`NaN` 是 “非数” (not a number). 运算结果为 `NaN` 的例子有：

```
0/0 # NaN
log(-1) # = NaN
0^(3+8i) # = NaN + NaNi
Inf-Inf; Inf/Inf # = NaN
-NaN # = NaN
```

`Inf` 和 `NaN` 的类型是 `numeric`（浮点数）.

```
class(Inf); class(NaN)
```

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

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

2.3.2.5 R 中自带的数学函数集合

基础

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

函数	描述
`abs(x)`	$\|x\|$（取绝对值）
`sum(...)`, `prod(...)`	所有元素相加之和/相乘之积

2.3.3 简易的统计学计算

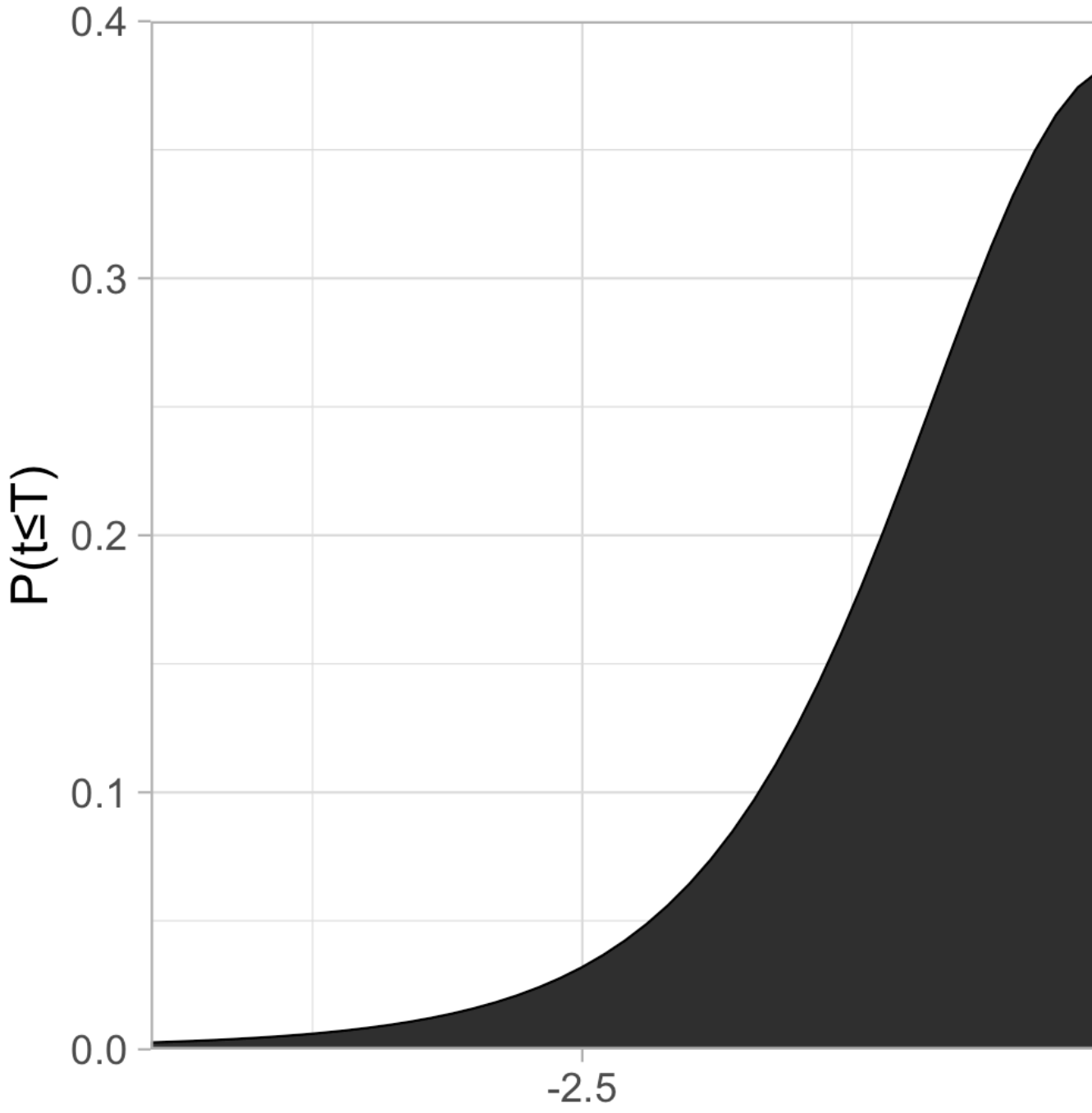
本节简要解释了 R 中的基础统计学函数，t 分布，t 检验和 χ^2 检验。统计学方法并不是本书的重点，因此可以酌情跳到下一章。

2.3.3.1 基础

中位数 `median()`; 平均数 `mean()`; 方差 `var()`; 标准差 `sd()`.

2.3.3.2 t 分布

众所周知，t 分布长这样:


0.4
0.3
0.2
0.1
0.0
P(t≤T)
-2.5

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。

2.3.3.3 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)

2.3.3.4 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
```

2.3.3.5 χ^2 检验

χ^2 检验有两种，goodness of fit test（适配度检验）和 contigency 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。

2.3.3.5.1 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）：

```r
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。

2.3.3.5.2 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:

```r
(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
```

```
##         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:

```r
chisq.test(社团参与)
```

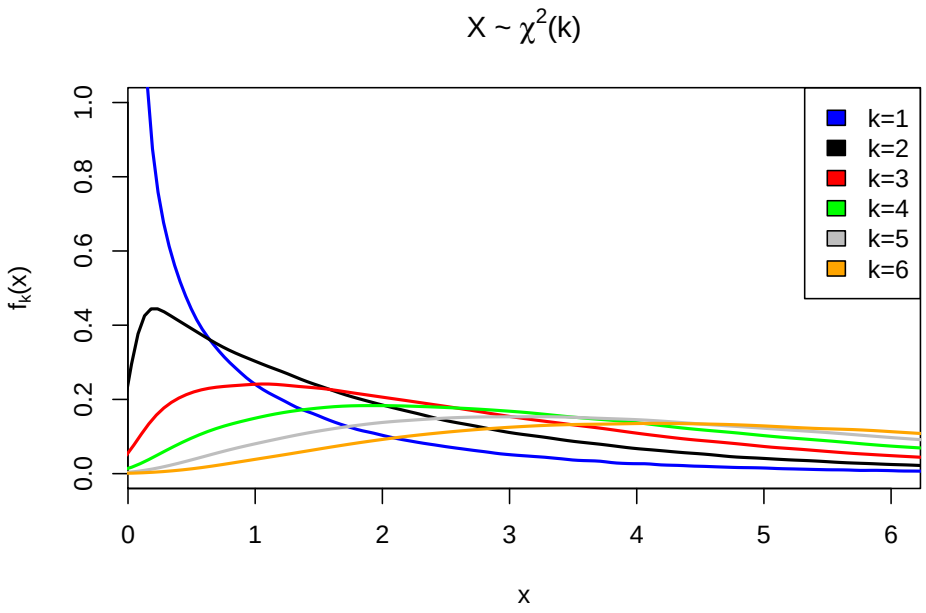
```
##
##  Pearson's Chi-squared test
##
## data:  社团参与
## X-squared = 3.7587, df = 4, p-value = 0.4396
```

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:

```r
# 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)
```



2.3.3.5.3 其他

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。

2.4 逻辑

2.4.1 逻辑值 {logical-values}

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）。

2.4.2 关系运算符和简单的逻辑运算

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）

这些关系运算符只能用于 (atomic) vectors, 不能用于其他类型的 R 对象；`indentical()` 函数可以用于所有类型的对象，用来确认两者是否完全一致。

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。

有很多种运算会以 NA 作为计算结果，在此不一一列举。最重要的一个是：

```
NA == NA
```

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

这看起来像是一个 bug，然而仔细想想才发现这个设计很巧妙。假设你问我是否知道我的一些朋友写完了暑假作业。我说我不知道张三是否写完了，也不知道李四是否写完了。你再问我“张三和李四的作业完成情况是一样的吗”？鬼才知道咧！

这意味着不能直接使用 `x == NA` 来判断 x 是否是 NA，而要用 `is.na()` 函数：

```
x <- NA
is.na(x)
```

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

2.4.2.1 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

2.4.2.2 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
```

2.4.2.3 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）。

| 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，可以不关注方法。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）。

2.5 判断和循环（流程控制）

2.5.1 给有编程基础者的快速指南

如果没编程基础，没接触过判断和循环，请看第2.5.2小节。

如果学过其他编程语言，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，然后跳至第2.6节：

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

```
## numeric(0)
```

```
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
```

2.5.2 无编程基础者的快速指南

我认为，举例子比纯粹的概念灌输更容易理解。

2.5.2.1 if, else if, else 语句（“如果……”，“或者，如果……”，“否则……”）

```
# 以下代码翻译成英语就是：If 1 + 1 = 2, print "hi". Else, print "bye".
# 或中文：如果一加一等于二，那么印出“hi”，否则印出“bye”.
if (1 + 1 == 2) { # 1 + 1 == 2 的运算结果是 TRUE，因此“如果”成真
  print("hi") # 所以会执行 `print("hi")`
} else {
  print("bye")
}
```

```
## [1] "hi"
```

```
# 代码第一行中的 FALSE 可以替换成任何计算结果为 FALSE 的运算，
# 比如 1 + 1 == 3；小括号内的计算过程不重要，
# 但运算结果必须为 TRUE 或 FALSE（不可以是 NA）
if (FALSE) {
  print("hi")
} else { # 因为是 FALSE，所以 `else`里的语句被执行
  print("bye")
}
```

```
## [1] "bye"
```

```
if (FALSE) { # 第一个 `if`为 FALSE
  print("hi")
} else if (FALSE) { # 检查下一个 `else if`, 也是 FALSE
  print("yoo")
} else if (TRUE) { # 再检查下一个 `else if`, 这次是 TRUE
  print("hey") # 所以执行 `print("hey")`
} else {
  print("bye") # 而轮不到 else
}
```

```
## [1] "hey"
```

2.5.2.2 for 循环

```
# 以下代码翻译成英文就是: for every element i in c(2, 4, 6, 8):
# assign i^2 to n, then print n
# 中文: 对 c(2, 4, 6, 8)`中的每一个元素 i:
# 创建一个 n 使得 n 等于 i 的平方, 然后印出 n
for (i in c(2, 4, 6, 8)) { # i 可以是任何你想要的名字, 比如 num
  n <- i^2 # 如果上一行是 for (num in ..., 这一行就要写成 n <- num^2
  print(n)
}
```

```
## [1] 4
## [1] 16
## [1] 36
## [1] 64
```

```
x <- vector(mode = "numeric") # 创建一个空的 numeric vector
for (m in 1:10) {
  if (m %% 2 == 0) {
    x <- append(x, m)
  }
}

x
```

```
## [1]  2  4  6  8 10
```

```
M <- c(1, 2, 3 ,4 ,5)
N <- c(10, 100, 1000)

x <- vector("numeric")
for (m in M) {
  for (n in N) { # 在一个 for 循环中嵌入另一个 for 循环
    x <- append(x, m*n)
```

```
  }
}

x
```

```
##  [1]   10  100 1000   20  200 2000   30  300 3000   40  400 4000   50  500
## [15] 5000
```

实际操作中，要想尽办法避免 **for** 循环，尤其是以上这种双层（多层）嵌套的 **for** 循环！原因和方法请看第2.5.4节。

2.5.2.3 while 循环

```
x <- 1
while (x < 10) { # 当 x<10 的时候，执行大括号内的语句
  print(x)
  x <- x + 3 # 一定要让 x 的值增加，否则会进入无限循环
}
```

```
## [1] 1
## [1] 4
## [1] 7
```

2.5.2.4 break 和 next

```
for (i in 1:10) {
  if (i == 3) {
    next # 当 i == 3 时，跳过它，继续（最近的）for 循环的下一个回合
  } else if (i == 6) {
    break # 当 i == 6 时，结束（最近的）for 循环
  }
  print(i) # 只有当 if 和 else if 里的检验都为 FALSE 时，`print(i)`才会执行。
}
```

```
## [1] 1
## [1] 2
## [1] 4
## [1] 5
```

```
M <- c(1, 2, 3, 4, 5)

x <- vector("numeric")
for (m in M) {
  while (TRUE) { # 原本 while(TRUE){} 将会是一个无限循环（判定条件永远 TRUE）
    x <- append(x, 2*m)
```

```
    break # break 打破了最近的这个 while 循环，而不影响 for 循环。
  }
}

x
```

```
## [1]  2  4  6  8 10
```

2.5.3 严谨版

如果看懂了上一节中的例子，并且作为新手不太想深究，完全可以暂时跳过这一节，前往第2.6节。

2.5.3.1 `if`, `else`, `else if` 语句

`if else` 语句长这样：

```
if (something is true) {
  do something
} else {
  do some other things
}
```

其中小括号内为测试的条件，其运算结果需为 TRUE 或 FALSE（不能是 `NA`!）。如果你还不熟悉关于逻辑值的计算，请看第2.4节。

- 若运算结果为 TRUE：大括号内的语句将会被执行。（如果语句只有一行，大括号可以省略）
- 如运算结果为 FALSE：
  - 如果后面没有 `else` 语句：什么都不会发生。
  - 如果后面有 `else` 语句：`else` 后（大括号里）的语句将会被执行。

R 中没有专门的 `elseif` 语句，但用 `else` 加上 `if` 能实现同样的效果。`else if` 可以添加在 `if` 语句之后，顾名思义（“或者如果”），它的作用是，如果前一个 `if` 测试的条件为 `FALSE`，那么再新加一个测试条件。一整个 `if/else/else if` 代码块里可以包含多个 `else if`.

注意，不能直接用 `x == NA` 来判断 `x` 是否是 `NA`，而要用 `is.na(x)`. 否则会得到 `NA` 的结果。

2.5.3.2 for 循环

2.5.3.3 while 循环

2.5.3.4 repeat 循环

2.5.3.5 break 和 next

2.5.4 如何避免 for 循环——apply() 家族函数

R 中的循环效率是很低的，尤其是有多层嵌套。通过 system.time() 函数，看看你的电脑执行以下运算需要花多少秒：（system.time() 函数在第2.6.5小节有介绍）

```
x <- vector("numeric")
system.time(
  for (l in 1:40) {
    for (m in 1:50) {
      for (n in 1:60) {
        x <- append(x, l*m*n)
      }
    }
  }
)
```

我的 i5 处理器 (i5-8259U CPU @ 2.30GHz) 花了 39 秒左右才能算出来，然而看起来计算量并不大：

$$x = (1 \times 1 \times 1, 1 \times 1 \times 2 \ldots, 40 \times 50 \times 59, 40 \times 50 \times 60)$$

一共有 $40 \times 50 \times 60 = 120000$ 次计算. 一个原因是，无论你的 CPU 有多少核心，R 默认只会使用其中的一个进行计算。在第2.5.5.1节中介绍了开挂使用多核的方法。但是它治标不治本，解决 for 循环缓慢的终极方案是避免使用 for 循环，而使用向量化的方法进行计算 (vectorized computation)。在第2.1.5我介绍了简单的（二元）向量化计算。除了二元运算以外，很多时候，复杂的 for loop 也能用向量化计算实现。我们需要用到 apply() 家族的一系列函数: apply(), sapply(), lapply(), mapply(), tapply(), vapply(), rapply(), eapply(); 此外，像 Map(), rep(), seq() 等函数也会执行向量化的计算。

在学习它们的用法之前，先来看一个直观的数据:

方法	$(L, M, N) = (1:40, 1:50, 1:60)$	$(L, M, N) = (1:500, 1:600, 1:700)$
普通（单核）for 循环	39 秒	等了一小时，无果，遂弃
开挂（四核）for 循环	12.304 秒；CPU 巨热	怕 CPU 炸，不敢试
sapply()	0.001 秒	2.719 秒
rep()	0.002 秒	2.825 秒
mapply()	0.004 秒	4.302 秒
rapply()	0.003 秒	2.094 秒
Map()	0.004 秒	3.106 秒

同样是运算上面那个 for 循环花了 39 秒的例子，使用 `sapply()` 函数和 `rep()` 函数几乎是瞬间完成；而把 (l, m, n) 增至 $(1:500, 1:600, 1:700)$ 时（计算量为 1750 倍），它们仍只需不到 3 秒，而 for 循环则是不可行的。

（源码在）

2.5.4.1 `lapply()`

`lapply()` (list apply) 有两个参数，第一个是对象（可以是 vector 或者 list），第二个是函数。它的作用是把函数作用于对象中的每一个元素，并返回一个 list.

无论对象是 vector 还是 list, 返回的都是一个 list:

```
lapply(c(1, 2, 3), function(i) i*10) # vector
```

```
## [[1]]
## [1] 10
##
## [[2]]
## [1] 20
##
## [[3]]
## [1] 30
```

```
lapply(list(1, 2, 3), function(i) i*10) # list
```

```
## [[1]]
## [1] 10
##
## [[2]]
## [1] 20
##
## [[3]]
## [1] 30
```

```
lapply(list(c(1, 2), c(4, 6), c(7, 9)), function(i) i*10)
```

```
## [[1]]
## [1] 10 20
##
## [[2]]
## [1] 40 60
##
## [[3]]
## [1] 70 90
```

2.5.4.2 sapply()

sapply() (simplified list apply) 的功能和 lapply() 几乎一样。sapply() 额外的一个特点是尽可能地化简结果：

- 当结果只有一个 list 元素时，sapply() 返回一个 vector
- 当结果有多个 list 元素，但每个 list 元素只包含一个 vector 且长度相等时，sapply() 会返回一个 matrix

试试以下计算：

```
lapply(c(1, 2, 3), function(i) i*10)
sapply(c(1, 2, 3), function(i) i*10)

lapply(list(c(1, 2), c(4, 6), c(7, 9)), function(i) i*10)
sapply(list(c(1, 2), c(4, 6), c(7, 9)), function(i) i*10)

lapply(list(1, 2, 3), function(i) i*c(1, 10, 100))
sapply(list(1, 2, 3), function(i) i*c(1, 10, 100))

lapply(list(c(1, 2), c(4, 6), c(7, 9)), function(i) i*10)
sapply(list(c(1, 2, 3), c(4, 6), c(7, 9)), function(i) i*10)
```

2.5.4.3 mapply() 和 Map()

mapply() 的意思是 “Map+sapply”，它的计算过程和 Map() 函数类似，但是会像 sapply() 一样把结果化简。

简单地说，有些函数把参数中的 vector 作为一个整体使用，而用 mapply()/Map() 可以把这些 vector 逐元素地使用。而且，mapply()/Map() 可以对多参数的函数进行向量化计算，因此 mapply() 的 “m” 经常被解释为代表 “multivariate”.

假设你想创建一个这样的 vector：

```
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4
```

通过复习第2.1.3.2节，很容易得出答案：

```
rep(1:4, each = 3)
```

粗心的读者可能以为是：

```
rep(1:4, 3) # 即 rep(1:4, times = 3)
```

而实际上它的运算结果是：

```
##  [1] 1 2 3 4 1 2 3 4 1 2 3 4
```

因为 1:4（即 c(1, 2, 3, 4)）被作为一个整体使用了。正确的写法是：

```
as.vector(mapply(rep, 1:4, 3))
```

`mapply(rep, 1:4, 3)` 的第一个参数是所需函数的名字，其它的参数为所需函数的参数。你可以指名道姓：`mapply(rep, x = 1:4, times = 3)`，或者根据排序键入参数而无需指定参数名（见第2.6.2节）。其中第一个参数 `(1:4)` 不是长度为 1 的 vector，因此它会被 `mapply()` 转换成 list 并执行逐元素运算，即运算过程和结果为：

```
mapply(rep, 1:4, 3)
= mapply(rep, c(1, 2, 3, 4), 3)
= list(rep(1, 3), rep(2, 3), rep(3, 3), rep(4, 3))
= list(c(1,1,1), c(2,2,2), c(3,3,3), c(4,4,4))

# 等价于 sapply(list(1, 2, 3, 4), function(x) rep(x, 3))
```

为什么我们实际上看到的是一个 matrix 呢？这是因为每个 list 元素所含的 vector 的长度相等，因此自动化简为 matrix. 若要查看未化简的版本：

```
mapply(rep, 1:4, 3, SIMPLIFY = FALSE)
```

或者

```
Map(rep, 1:4, 3)
```

```
## [[1]]
## [1] 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3 3
##
## [[4]]
## [1] 4 4 4
```

你可能已经发现，用 `sapply()` 函数其实也可以较为简单地实现这个例子：`sapply(1:4, function(x) rep(x, 3))`；那 `mapply()`/`Map()` 有没有其它特殊的特性呢？有。多参数的向量化运算。对于多参数的函数，`sapply()` 只能使用其中的第一个进行向量化运算，而其他的参数必须为常数。而 `mapply()`/`Map()` 可以这样：

```
Map(seq, c(1, 5, 20), c(5, 25, 100), length.out = 5)
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
## [1]  5 10 15 20 25
##
## [[3]]
```

```
## [1]  20  40  60  80 100
```

它执行了三次计算:`seq(1, 5, length.out = 5)`, `seq(2, 25, length.out = 5)` 和 `seq(20, 100, length.out = 5)`.

想一想，这个结果是否可以化简？如果是，化简结果是什么？用 `mapply()` 函数执行同样的计算来验证你的答案。

当对象是一个 list 的时候，

```
Map(rep, list(c(8, 9), c(6, 7)), 3)
```

```
## [[1]]
## [1] 8 9 8 9 8 9
##
## [[2]]
## [1] 6 7 6 7 6 7
```

```
lapply(list(c(8, 9), c(6, 7)), function(x) rep(x, 3))
```

```
## [[1]]
## [1] 8 9 8 9 8 9
##
## [[2]]
## [1] 6 7 6 7 6 7
```

想一想，`Map(rep, list(c(1,2), list(2,3)), 3)` 的计算结果是什么？

2.5.4.4 `rep()`

还是上一小节中的 vector:

```
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4
```

其实可以仅用 `rep()` 函数简洁地实现:

```
rep(1:4, rep(3, 4))
```

如果不明白为什么，请复习第2.1.3.2节。

2.5.5 `foreach` package: for 循环的进化版

`foreach` package 相对于 base R 中的 for 循环增加了一些特性，不过最实用的是支持多核并行运算:

2.5.5.1 使用多内核进行计算

首先需要安装和使用 `doParallel`，然后才可以使用 `foreach` 中的`%dopar` 进行多核并行运算。

查看和设置内核数量：

```
library(doParallel)
getDoParWorkers() # 查看 R 当前使用的内核数量；默认应为 1
```

```
## [1] 4
```

```
detectCores() # 查看可用内核总数
```

```
## [1] 8
```

```
registerDoParallel(4) # 设置内核数量
getDoParWorkers() # 再次检查内核数量
```

```
## [1] 4
```

设置完之后就可以使用%dopar 进行多核并行运算了：

```
x <- foreach(l = 1:40, .combine = "c") %dopar% {
    foreach(m = 1:50, .combine = "c") %dopar% {
        foreach(n = 1:60, .combine = "c") %do% {
        l*m*n
        }
    }
}
x
```

相比单核 for 循环的 39 秒，开挂（四核）的速度是 12 秒（计算量越大，优势越明显）。

2.6 函数

2.6.1 R 中的函数

不像很多其他语言的函数（和方法）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

```
sample <- c(5.1, 5.2, 4.5, 5.3, 4.3, 5.5, 5.7)
# 根据传统，赋值变量时用 `<-`号，赋值函数参数时才用 `=`
t.test(x = sample, mu = 4.5)
```

```
##
##  One Sample t-test
##
## data:  sample
## 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
```

二元运算符和 `[`（取子集符号）看起来一点都不像函数，而实际上它们也是函数，因此也可以用通用的格式使用他们，只是需要加上引号：

```
"+"(2, 3)
```

```
## [1] 5
```

```
"["(c(" 四川担担面", " 武汉热干面", " 兰州牛肉面", " 北京炸酱面"), 2)
```

```
## [1] " 武汉热干面"
```

可自定义的二元运算符形式为`%x%`, 其中 `x` 为任何字符。（见第2.6.3.3节）

(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"。)

2.6.2 调用函数

根据通用格式(`function(argument1 = value1, argument2 = value2, ...)`)调用函数。对于二元运算符，`a %x% b` 等价于`"x"(a, b)`.

从 "`function(`" 开始到此函数结尾的 "`)`" 中间为参数，参数用逗号隔开，空格和换行会被忽略，"`#`" 符号出现之处，那一行之后的内容都会被忽略。这意味着你可以（丧心病狂地）像这样调用一个函数。

```
sum         (
 # 4
        4 # 我怕不是
,               # 疯了哦
             6


  )
```

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

它实际的好处是，当参数很长或是有嵌套的函数时，可以通过换行和空格使代码更易读，就像其它的编程语言一样。

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

```
## 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要写清楚是 `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。

2.6.3 创建函数

2.6.3.1 普通函数

```
函数名 <- function(参数 1, 参数 2, ...){
  对参数 1 和参数 2
  进行
  一系列
  一行或者多行
  计算
  return(计算结果)
}
```

在 R 中，函数是作为对象保存的，因此定义函数不需要一套另外的符号/语句，还是用赋值符号 `<-`，和 `function()` 函数。

R 自带了计算样本标准差 (standard deviation, s) 的函数，`sd()`，我们可以根据它写一个计算均值标准差（即“标准误”, standard error）($SE = s_{\bar{x}} = \frac{s}{\sqrt{n}}$)

```
SE <- function(x) {
  s <- sd(x)
  n <- length(x)
  result <- s/sqrt(n)
  return(result)
}
# 随后，你就可以使用自定义的函数了
SE(c(5,6,5,5,4,5,6,6,5,4,5,3,8))
```

```
## [1] 0.3367673
```

这里其实可以做一些省略。很多时候，最后一“句”的计算结果（不是赋值计算）就是我们想 `return` 的结果。因此，这时 `return` 可以省略：

```
SE <- function(x) {
  s <- sd(x)
  n <- length(x)
  s/sqrt(n) # 注意不是 `result <- s/sqrt(n)`
}
SE(c(5,6,5,5,4,5,6,6,5,4,5,3,8))
```

```
## [1] 0.3367673
```

很多时候，函数内部有复杂流程控制，这时使用 `return()` 可以很大地增强易读性：

```
# 这是随手写的一个没有意义的函数
myfunc <- function(i){
  k <- 8
  if (i>3) {
    j <- -i
    while(j < 20){
      k <- k + i + j
      j <- j+5
    }
    return(k)
  } else {
    if (i %% 2 == 0) {
      return(5)
    } else return(k*i)
  }
}
myfunc(6)
```

```
## [1] 83
```

2.6.3.2 无名函数

本章剩余的内容，都是比较进阶的了。可以酌情从这里跳转至本章第2.6.9节。

函数不需要名字也可以执行。一般，会与 `apply` 族函数联用（见第2.5.4节）：

```
sapply(1:5, function(x) x^2)
```

```
## [1]  1  4  9 16 25
```

或者用于

2.6.3.3 二元运算符

定义二元运算符的方式和定义普通函数的方法极其类似，只是参数必须要有且仅有两个（否则作为“二元”运算符就无意义了），且运算符名称需要用引号包围。

比如我们可以定义一个计算椭圆面积的函数

```
'%el%' <- function(x, y) pi*x*y

2 %el% 5
```

```
## [1] 31.41593
```

原则上，可自定义的二元运算符不一定要用% 包围；+, -, : 等符号的功能都可以被自定义，但是它们是 R 自带的，非常常用的函数，重定义它们只会带来麻烦。

2.6.3.4 闭包 (Closure)

函数里可以包含着另一个函数，这就形成了一个闭包：

```
myfunc <- function(){
  a = 5
  function(){
    b = 10
    return(a*b)
  }
}
# 执行 myfunc() 的时候，默认结果为最后一句/一行，在这里应为内函数：
myfunc()
```

```
## function(){
##     b = 10
##     return(a*b)
##   }
## <environment: 0x7fb2f2ec9b20>
```

```
# 既然 `myfunc()`的结果是一个函数，那么在后面再加上一个括号就是执行内函数了；内函数可以使用外函数中所定
myfunc()()
```

```
## [1] 50
```

```
speak <- function(x){
  x()$speak
}

speak(cat)
```

```
## NULL
```

2.6.3.5 伪·OOP

我貌似，捣鼓出了一种完全使用 R 中的简易函数实现伪·OOP 的方法（R 中的真·OOP 是有三种，S3，R6 和 S4）：

```
Cat <- function(name){
  name = name
  binomial_name <- "Felis catus"
  speak <- "Meow"
  greet <- function(time = "not_specified"){
    intro <- paste("my name is", name)
    if(time == "morning") print(paste("Good morning,", intro))
    if(time == "afternoon") print(paste("Good afternoon,", intro))
    if(time == "evening") print(paste("Good evening,", intro))
    if(time == "not_specified") print(paste("Hi,", intro))
  }
    paste("Hi, my name is ", name)
  list(name = name, binomial_name = binomial_name, speak = speak, greet = greet)
}

Felix <- Cat("Felix")
Felix$name
```

```
## [1] "Felix"
```

```
Felix$greet("morning")
```

```
## [1] "Good morning, my name is Felix"
```

inheritance 和 polymorphism 的实现

```
Pet <- function(name = NA, common_name = NA, binomial_name = NA, speak = NA){
  name <- name
  common_name <- common_name
  binomial_name <- binomial_name
  speak <- speak
  greet <- function(time = "not_specified"){
    intro <- paste("I'm a", common_name, "and my name is", name)
    if(time == "morning") print(paste("Good morning,", intro))
    if(time == "afternoon") print(paste("Good afternoon,", intro))
    if(time == "evening") print(paste("Good evening,", intro))
    if(time == "not_specified") print(paste("Hi,", intro))
  }
    paste("Hi, my name is ", name)
  list(name = name, common_name = common_name, binomial_name = binomial_name, speak = s
}


Turtle <- function(name = NA){
  Pet(name, "turtle", "Trachemys scripta elegans") # 实现 inheritance # 龟没有叫声
}
```

```
Cat <- function(name = NA, sterilized = NA){
  sterilized <- sterilized # 猫可能绝育 # 新增 attribute, 实现了广义的 polymorphism
  PetAaM <- Pet(name, "cat", "Felis catus", "Meow")
  CatOnlyAaM <- list(sterilized = sterilized)
  c(PetAaM, CatOnlyAaM)
}

# 实现了 Python 语境中的 polymorphism
greet <- function(pet, time = "not_specified"){
  pet$greet(time)
}

binomial_name <- function(pet){
  pet$binomial_name
}
```

使用例:

```
Felix <- Cat("Felix", "TRUE")
Kazuya <- Turtle("Kazuya")

Felix$binomial_name
```

```
## [1] "Felis catus"
```

```
Kazuya$greet("afternoon")
```

```
## [1] "Good afternoon, I'm a turtle and my name is Kazuya"
```

```
greet(Felix, "morning")
```

```
## [1] "Good morning, I'm a cat and my name is Felix"
```

```
sapply(list(Kazuya, Felix), binomial_name)
```

```
## [1] "Trachemys scripta elegans" "Felis catus"
```

没有 `class`, 没有 `self`, 没有 `__init__`, *it just works.*

2.6.4 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 "其他参数" (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**??**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。

2.6.5 测速

当你开始处理复杂，大量的数据时，或是向别人分享自己的代码时，代码执行的速度变得重要。

一段代码/一个函数经常有很多种写法，哪种效率更高呢？实践是检验真理的唯一标准，R 提供了一个测速函数：system.time() 函数。

```
x <- vector('numeric')
system.time(
  for (i in 1:50){
    for (j in 1:100) {
      x <- append(x, i*j)
    }
  }
)
```

```
##    user  system elapsed
##   0.041   0.011   0.051
```

其中第三个数字 (elapsed) 是执行 system.time() 括号内的语句实际消耗的时间。可以使用索引 ([3]) 抓取。

如果括号内的语句大于一句，像这样：

```
system.time(
  1 + 1
  2 + 1
)
```

R 会报错。就像流程控制里学到的那样，需要用大括号包围多行/多句的语句，就像这样：

```
system.time({
  1 + 1
  2 + 1
})
```

2.6.6 列表 (list)

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** 中的对象 (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: 0x7fb2edcdec20>
## <environment: namespace:stats>
```

2.6.6.1 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; 父 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
```

2.6.6.2 合并与拆解

通过 `c()` 函数来合并多个列表。

```
c(list(1, 2), list(3, 4, list(5,6)))
# 将等同于 list(1, 2, 3, 4, list(5,6))
```

也许你想把需要“合并”的列表作为子列表放在另一个列表里；这也很简单，在本节一开始就讲了：

```
list(list(1, 2), list(3, 4))
```

```
## [[1]]
## [[1]][[1]]
## [1] 1
##
## [[1]][[2]]
## [1] 2
##
##
## [[2]]
## [[2]][[1]]
## [1] 3
##
## [[2]][[2]]
## [1] 4
```

通过 `unlist()` 函数来拆解列表中的子列表。若参数 `recursive` 为 `TRUE`（默认值），将一直拆解至无子列表的列表，如果此最简列表的元素都属于五种 atomic vector 中的数据[2]，此列表还会被进一步化简成向量。若 `recursive = FALSE`，最“靠外”的一级列表（可能是多个）将会被拆解。

```
unlist(list(1, list(2, list(3, 4)), list(5, 6), 7, 8, 9))
# 将等同于 c(1, 2, 3, 4, 5, 6, 7, 8, 9)
# 注意被化简成了向量

unlist(list(1, list(2, list("a", 4)), list(5, TRUE), 7L, 8, 9+0i))
# 将等同于 c("1", "2", "a", 4, 5, "TRUE", "7", 8, "9+0i")
# 化简成向量时，非字符元素被强制转换成字符了

unlist(list(1, list(2, list(t.test, 4)), list(5, TRUE), 7L, x, 9+0i))
# t.test 无法存储于向量中，因此最简结果为一个 list:
# list(1, 2, t.test, 4, 5, TRUE, 7L, x, 9+0i)
```

[2] dataframe 也是可以 unlist 成向量的，但是并不实用。(试试 `unlist(list(data.frame(x = c(1,2), y = c(3,4)), 5, 6))`)

```
unlist(list(1, list(2, 3, list(4, 5)), list(6, 7), 8, 9), recursive = FALSE)
# 将等同于 list(1, 2, 3, list(4, 5), 6, 7, 8, 9)
```

因此，当 A, B 为列表，unlist(list(A, B), recursive = FALSE) 等同于 c(A, B).

2.6.6.3 其他性质和操作

上面说到 unlist(list(A, B), recursive = FALSE) 等同于 c(A, B)，你可能很想用 == 验证一下。很不幸，你会得到一条错误信息:

```
comparison of these types is not implemented
```

在第 @{logical-operations} 节讲过，== 只能用于 atomic vectors；对于列表（和其他对象）可以用 identical() 函数确认两者是否完全一致。

```
A <- list("a", 1, TRUE); B <- list(5+8i, NA, 4L)
C1 <- unlist(list(A, B), recursive = FALSE); C2 <- c(A, B)
identical(C1, C2)
```

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

2.6.7 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 4 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。每个二位矩阵存满后，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
```

2.6.8 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,] 0.6718315 0.4770758 0.2446106 0.86823106 0.7428044
## [2,] 0.4101549 0.2141378 0.6973227 0.31838107 0.8595585
## [3,] 0.4380271 0.9520504 0.8705479 0.57497412 0.4063315
## [4,] 0.7047464 0.7772618 0.7139401 0.95823908 0.4899111
## [5,] 0.3364387 0.4288986 0.1960972 0.04000405 0.6076336
## [6,] 0.9677286 0.7396727 0.7087386 0.01501767 0.3936719
##
## , , 2
##
##           [,1]      [,2]      [,3]        [,4]      [,5]
## [1,] 0.9006686 0.7380892 0.5167960 0.703028908 0.9207938
## [2,] 0.1635478 0.2071612 0.5997855 0.001126975 0.1751163
## [3,] 0.2380316 0.4814276 0.2912440 0.277010770 0.9268620
## [4,] 0.3115802 0.9184094 0.6297072 0.915726600 0.2515638
## [5,] 0.7631003 0.3687141 0.1921361 0.405785865 0.2536388
## [6,] 0.8835738 0.4548185 0.3347968 0.816077598 0.6582365
##
## , , 3
##
##           [,1]      [,2]       [,3]       [,4]      [,5]
## [1,] 0.2833486 0.7377325 0.24121750 0.07658559 0.5553004
## [2,] 0.7801159 0.8058647 0.57364324 0.30177394 0.5410325
## [3,] 0.9237641 0.8137241 0.06959596 0.47938977 0.1998257
## [4,] 0.3830497 0.8257545 0.54016980 0.35242275 0.6334874
## [5,] 0.6706461 0.6323191 0.92935551 0.08920604 0.7562945
## [6,] 0.2686587 0.7297881 0.59177542 0.95546656 0.7247634
```

然后我们用 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     0.6718315   0.4770758 0.2446106 0.86823106   0.7428044
## trial. 2     0.4101549   0.2141378 0.6973227 0.31838107   0.8595585
## trial. 3     0.4380271   0.9520504 0.8705479 0.57497412   0.4063315
## trial. 4     0.7047464   0.7772618 0.7139401 0.95823908   0.4899111
## trial. 5     0.3364387   0.4288986 0.1960972 0.04000405   0.6076336
## trial. 6     0.9677286   0.7396727 0.7087386 0.01501767   0.3936719
##
## , , artemisinin
##
##          P. falciparum P. malariae  P. ovale    P. vivax P. knowlesi
## trial. 1     0.9006686   0.7380892 0.5167960 0.703028908   0.9207938
## trial. 2     0.1635478   0.2071612 0.5997855 0.001126975   0.1751163
## trial. 3     0.2380316   0.4814276 0.2912440 0.277010770   0.9268620
## trial. 4     0.3115802   0.9184094 0.6297072 0.915726600   0.2515638
## trial. 5     0.7631003   0.3687141 0.1921361 0.405785865   0.2536388
## trial. 6     0.8835738   0.4548185 0.3347968 0.816077598   0.6582365
##
## , , doxycycline
##
##          P. falciparum P. malariae   P. ovale   P. vivax P. knowlesi
## trial. 1     0.2833486   0.7377325 0.24121750 0.07658559   0.5553004
## trial. 2     0.7801159   0.8058647 0.57364324 0.30177394   0.5410325
## trial. 3     0.9237641   0.8137241 0.06959596 0.47938977   0.1998257
## trial. 4     0.3830497   0.8257545 0.54016980 0.35242275   0.6334874
## trial. 5     0.6706461   0.6323191 0.92935551 0.08920604   0.7562945
## trial. 6     0.2686587   0.7297881 0.59177542 0.95546656   0.7247634
```

清清楚楚，一目了然。

2.6.9 小测

(1) 转换年份到世纪。写一个名为 as.century 的函数，把存储着年份的向量，比如 years <- c(2014, 1990, 1398, 1290, 1880, 2001)，转换成对应的世纪（注意，19XX 年是 20 世纪）。

(2) 斐波那契数列。

- 背景：斐波那契数列是指 $F = [1, 1, 2, 3, 5, 8, ...]$，其中：
  - $F_1 = 1$，$F_2 = 1$
  - 从 F_3 开始，$F_i = F_{i-2} + F_{i-1}$

- (也有 $F_0 = 0, F_1 = 1$ 的说法，但是为了方便我们不用这个定义)
- 题目：创建一个函数名为 `fibon()` 的函数，使得 `fibon(i)`：
  - 当 $i \in \mathbb{Z}^+$ 时，返回向量 $[F_1, F_2, \ldots, F_i]$
  - 当 $i \notin \mathbb{Z}^+$ 时，返回" 请输入一个正整数作为 `` `fibon()` ``的参数。"
- 提示：
  - 虽然在 R 中整数用 1L, 2L 等表示，用户在被指示 “输入整数” 的时候很有可能输入的是 2 而不是 2L. 2 是否等于 2L? 如果是，如何利用它检测输入的是否是整数?（2 和 2L 都要被判定为 “是整数”）
  - 斐波那契数列前两位是定义，从第三位开始才是计算得出的。

Chapter 3

dataframe（数据框）和 tibble

3.1 查看 dataframe/tibble 并了解它们的结构

3.1.1 dataframe/tibble 的基本概念

dataframe 是 R 中存储复杂（多变量）数据的规范格式，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：

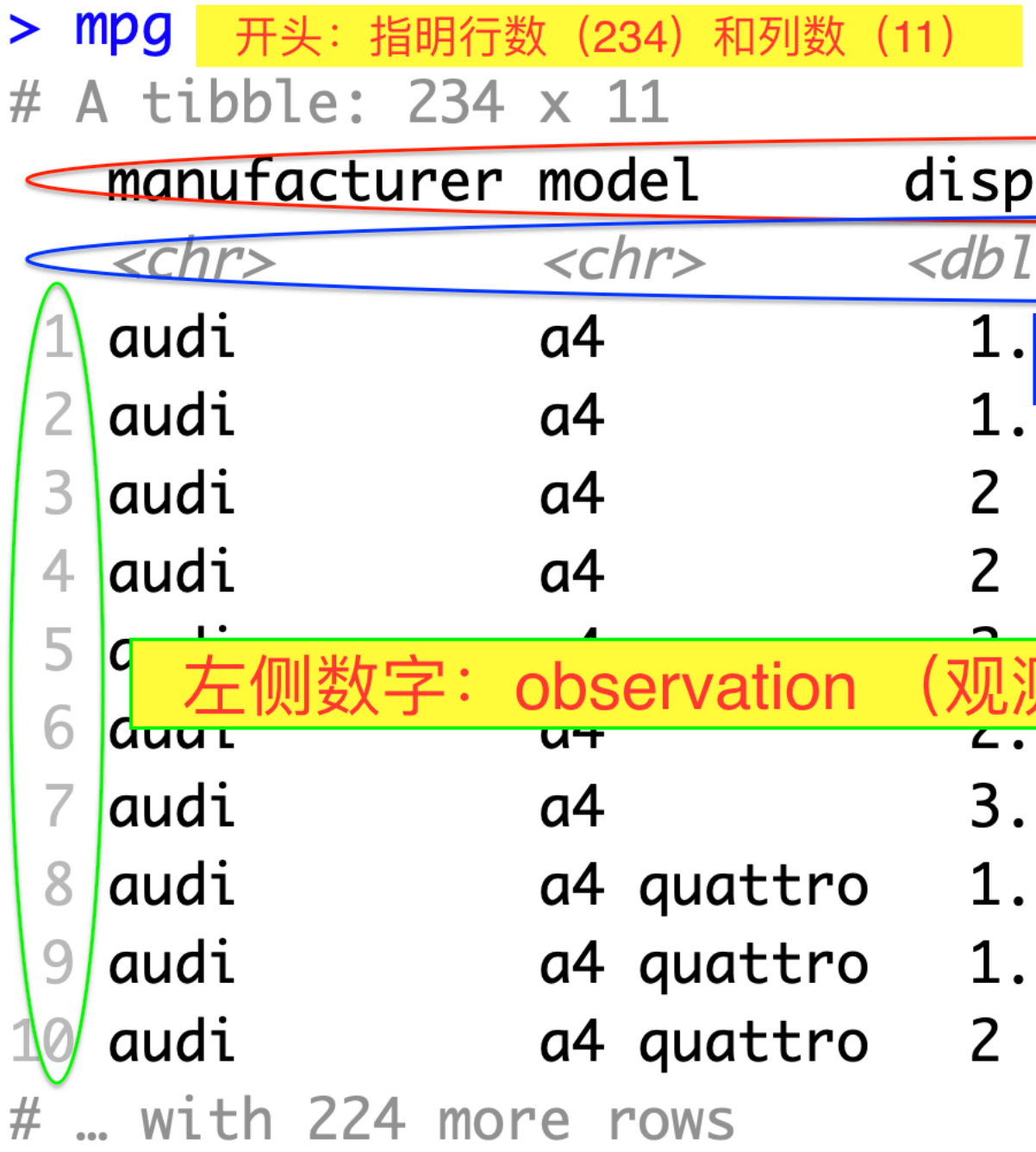
首先加载 tidyverse：

```
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
```



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，但是现实中上人们经常会做出 “不整齐” 的数据。把不整齐的数据弄整齐是下一章的重点。

3.1.2 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)
```

3.2 tibble 的创建和基础操作

3.2.1 创建 tibble

3.2.1.1 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
```

3.2.1.2 把 dataframe 转换成一个 tibble

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

3.2.1.3 从外部数据创建 tibble

参见第5.2.1节（数据的导入）

3.2.2 取子集（抓取行，列）{tbl-subsetting}

本节介绍了如何使用 dplyr package 提供的 select(), filter(), slice 取子集方法更详细的解释请看第3.3.1.2节。

3.2.2.1 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
```

一般我们抓取单列是为了在 tibble 中新建一个与那一列相关的变量，或是建一个新 tibble，或是做统计学分析。以上三种情况（是绝大多数的情况）用 vector 进行操作很方便。

假设你在写一个复杂的函数，且需要保持数据的完整性和一致性，可以使用单方括号 `[`；这样得到的是一个 tibble（试试 `mpg[3]`）这个特性在第3.3.1.2节中有解释。

3.2.2.2 抓取多列并返回一个 tibble

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
```

显然，使用变量名抓取列比使用索引更好。虽然打字较多，但是易读性比使用索引强太多了。在向其他人展示或者分享你的工作时，易读性尤为重要。

3.2.2.3 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2.4.2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~
```

3.2.2.4 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~
```

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 audi         a4 qu~   2.8  1999     6 manu~ 4        17    25 p     comp~
## 2 lincoln      navig~   5.4  2008     8 auto~ r        12    18 r     suv
## 3 ford         f150 ~   5.4  2008     8 auto~ 4        13    17 r     pick~
## 4 dodge        dakot~   4.7  2008     8 auto~ 4        14    19 r     pick~
```

3.3 其它

3.3.1 list 和 dataframe/tibble

3.3.1.1 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2.6.6.1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"
```

3.3.1.2 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~
```

3.3.2 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?

```
sapply(MyDF, class)
dim(MyDF)
```

zaocheng(Gentleman 2009, 33; Wickham 2019)

请查看Advanced R了解更多。

Chapter 4

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: Elegand Graphics for Data Analysis*(Wickham 2015)。

4.1 哲理

4.2 基础

4.2.1 基本语法

4.2.2 图像类型

4.3 进阶

4.3.1 逐层作图

4.3.2 尺寸，轴，和图例

4.3.3 位置

4.3.4 背景/主题的修改

4.3.5 与 ggplot 编程

4.4 附：Base R 中的作图

Chapter 5

数据处理

5.1 把 “untidy data” 整成 “tidy data”

5.1.1 “untidy data” 的主要问题

5.1.2 解决方案

5.2 数据的导入和导出

5.2.1 导入

5.2.1.1 csv

5.2.1.2 excel

5.2.1.3 其它

5.2.2 导出

5.3 字符串的处理

Base R 中有一些用于操作字符串的函数，但是因为各种原因它们很难用。因此我们使用一系列 `stringr` 中的函数（`stringr` 是 `tidyverse` 的一部分）。`stringr` 的函数都以 `str_` 开头。

5.3.1 基础

5.3.1.1 引号的使用

字符串可以用单引号和双引号包围。在双引号包围的环境下，可以很容易打出英澳常用的单引号和欧洲语言中的“撇”；在单引号包围的环境下，可以很容易打出北美和中国常用的双引号。否则需要使用转义字符 (escape character), \. 以下是几个正确的例子。

```
"'The unexamined life is not worth living' —Socrates"
```

```
## [1] "'The unexamined life is not worth living' —Socrates"
```

```
"La science n'a pas de patrie."
```

```
## [1] "La science n'a pas de patrie."
```

```
'" 老子曰：“知不知，尚矣；不知知，病矣。"'
```

```
## [1] "\"老子曰：“知不知，尚矣；不知知，病矣。\""
```

```
'l\'homme'
```

```
## [1] "l'homme"
```

5.3.1.2 换行符和制表符

假设你想显示以下效果：

```
## Guten
##
##  Morgen.
```

即 “Guten” 后有两次换行，第三行开头有一个制表符 (TAB)

你需要的源代码是：

```
"Guten\n\n\tMorgen."
```

\n (newline) 为换行符，\t (tab) 为制表符。所有可用的通过\实现的符号请参见 help("'")（关于引号的帮助）.

5.3.1.3 print() 和 writeLines()

print() 只显示源码，writeLines() 显示真实效果。

```
print(c("Guten\n\n\tMorgen.", "Guten\n\n\tTag"))
```

```
## [1] "Guten\n\n\tMorgen." "Guten\n\n\tTag"
```

```
writeLines(c("Guten\n\n\tMorgen.", "Guten\n\n\tTag"))
```

```
## Guten
##
##  Morgen.
## Guten
##
##  Tag
```

索引和引号消失了，不同的元素之间有换行。

5.3.2 使用 str_sub() 取子集

```
A <- "D. rerio"
str_sub(A, 1, 5) # 第 1 到第 5 个字母。计入符号和空格。
```

```
## [1] "D. re"
```

```
str_sub(A, 4, 4) # 抓取一个字母
```

```
## [1] "r"
```

```
str_sub(A, -4, -2) # 倒数第 4 至倒数第 2
```

```
## [1] "eri"
```

我们还可以通过索引修改某个位置的字符：

```
W <- "D. Rerio"
str_sub(W, 4, 4) <- str_to_lower(str_sub(W, 4, 4))
W
```

```
## [1] "D. rerio"
```

和 str_to_lower() 相关的函数还有 str_to_upper()，str_to_title() 和 str_to_sentence()。它们的作用都顾名思义。

5.3.3 使用 str_c() 进行字符串的合并

一个简单的例子：

```
str_c("a", "b", "c", sep = "")
```

```
## [1] "abc"
```

其中参数 sep 是被合并的字符串之间的连接字符；它可以是任何字符，包括空格和无（比如上面的例子；用 sep = "" 表示无连接字符）。

当需要合并的字符串保存在一个向量里时，用 collapse 而不是 sep：

```
str_c(c("a", "b", "c"), collapse = "[x@")
```

```
## [1] "a[x@b[x@c"
```

str_c() 可以执行向量化运算：

```
str_c("prefix", c("a", "b", "c"), "suffix", sep = "-")
```

```
## [1] "prefix-a-suffix" "prefix-b-suffix" "prefix-c-suffix"
```

所以我们可以这么玩：

```
混沌在各地的称呼 <- str_c(
  str_c(
    " 地区",
    c(" 北京", " 湖北", " 巴蜀", " 两广", " 闽台"),
    sep = ": "
  ),
  str_c(
    " 称呼",
    c(" 混沌", " 包面", " 抄手", " 云吞", " 扁食"),
    sep = ": "
  ),
  sep = " "
)

writeLines(混沌在各地的称呼)
```

```
## 地区: 北京 称呼: 混沌
## 地区: 湖北 称呼: 包面
## 地区: 巴蜀 称呼: 抄手
## 地区: 两广 称呼: 云吞
## 地区: 闽台 称呼: 扁食
```

它还可以和 if 语句联用：

```
win <- 2
score <- str_c(
  " 张三",
  if (win == 1) " 赢\n" else " 输\n",
  " 李四",
  if (win == 2) " 赢" else " 输",
  sep = ""
)
writeLines(score)
```

```
## 张三输
## 李四赢
```

5.3.4 使用 str_view() 来查找特定的字符组合

5.3.5 str_detect()

```
suomi <- "Suomen kieli on uralilaisten kielten itämerensuomalaiseen ryhmään kuuluva kieli."
```

5.4 Factors

5.4.1 基础

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
```

5.4.2 在绘图中的应用

5.4.3 高端操作

5.5 日期和时间

5.5.1 基础

5.5.2 计算

5.5.3 在绘图中的应用

Chapter 6

与 Python 的联合使用

6.1 在 R 中使用 Python: `reticulate`

6.2 在 Python 中使用 R: `rpy`

6.3 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)

References

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

Gentleman, Robert. 2009. *R Programming for Bioinformatics*. Book. Boca Raton, FL: CRC Press.

Kernighan, Brain W., and Ednnis 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.

———. 2019. *Advanced R*. 2nd ed. CRC Press.

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.