第 一 章　基础语法

1.1 搭建 R 环境及常用操作

1.1.1 搭建 R 环境

R 语言原生官网速度慢，建议直接到到 R 镜像站，目前国内有 9 个镜像站，我常用的两个是清华大学和同济大学的：

https://mirrors.tuna.tsinghua.edu.cn/CRAN

https://mirrors.tongji.edu.cn/CRAN

根据自己的操作系统，下载相应的最新版 R-4.0.2 安装即可，由于免费软件都是简单的下一步，不再赘述。Windows 系统安装时可根据系统只选择 32 位或 64 位版本。

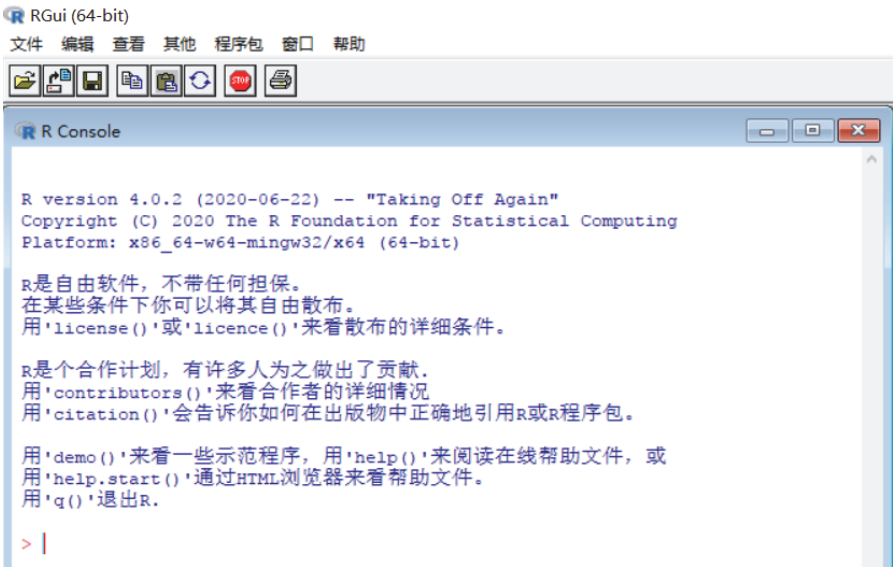


图 1.1: R 4.0.2 运行界面

建议安装在 D 盘，不要有中文路径，且路径不要有空格。

切记：若 Windows 系统用户名为中文，先改成英文！

注意，最好保证电脑里有且只有一个版本的 R，否则 RStudio 自动关联 R 的时候会出现一些麻烦。

安装 RStudio

不要直接使用 R, 而是使用更好用的 R 语言集成开发环境 Rstudio，官网下载地址：

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

下载安装（或直接下载 zip 版解压），将自动关联已安装的 R。

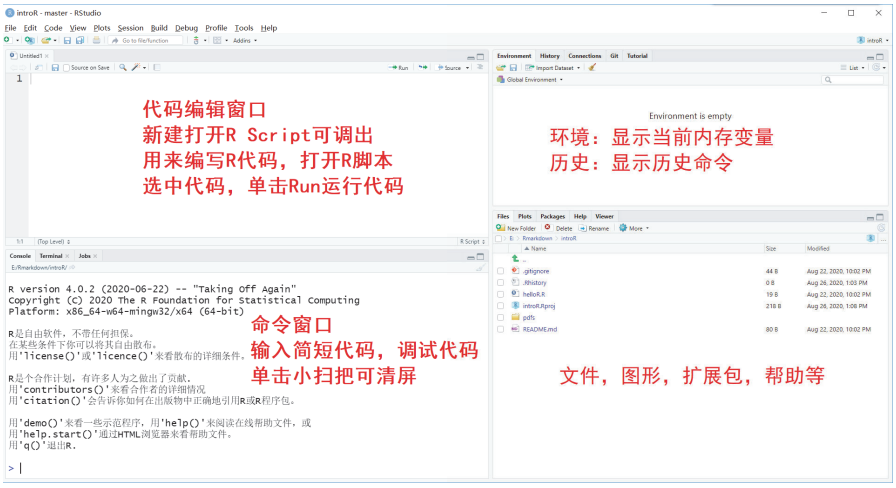


图 1.2: R Studio 操作界面

一些必要的设置

- 切换安装扩展包的国内镜像源（任选其一）

【Tools】—--【Global Options. . .】，在 Options 窗口点 Packages，点 Change 修改为，比如同济大学镜像源：

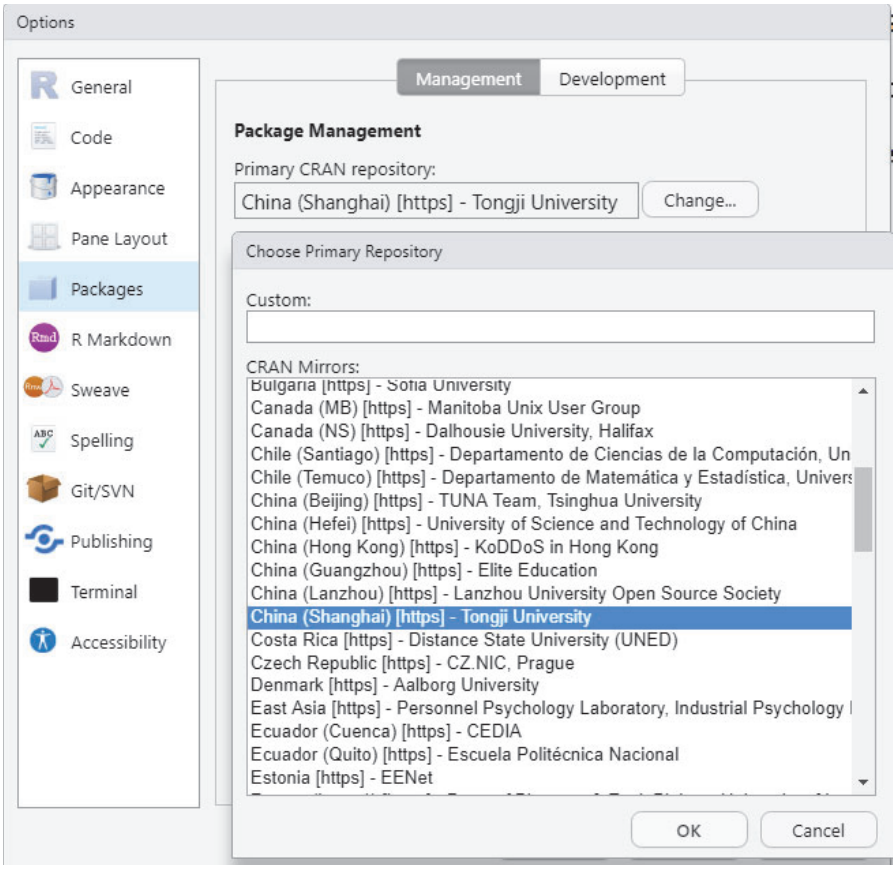


图 1.3: R Studio 设置国内镜像源

- 设置中文编码方式（避免 R 脚本 Rmarkdown 等文件乱码）

【Tools】——-【Global Options. . . 】，点【code】——-【Saving】，在 Default text encoding 框，点【change】，修改为 UTF-8，【OK】

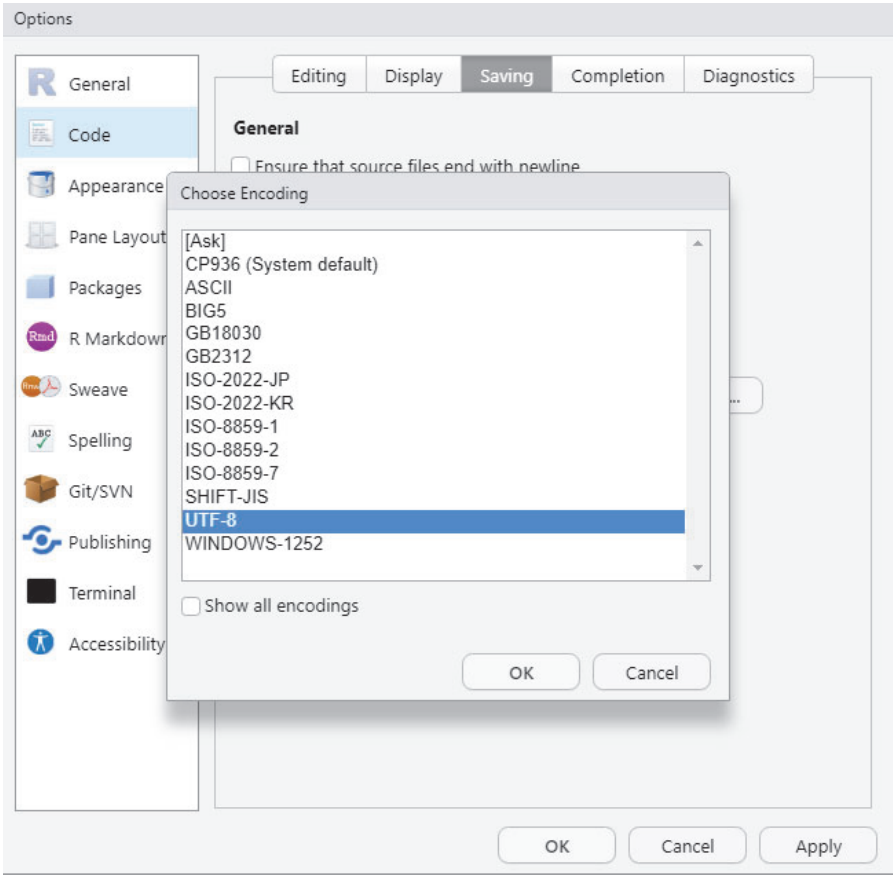


图 1.4: R Studio 设置 code 编码

1.1.2 常用操作

安装包

扩展包（package），简称包。通常 R 包都来自 CRAN，审核比较规范严格，包的质量相对更有保障。建议使用命令安装：

```
install.packages("openxlsx")
```

> openxlsx 为包名，必须要加引号（R 中单双引号通用）。

有些包不能自动安装，可以手动从网上搜索到下载.zip 或.tar.gz 文件到本地，再手动安装（不建议）：

【Tools】——-【Install Packages】，修改 Install from, 然后浏览安装

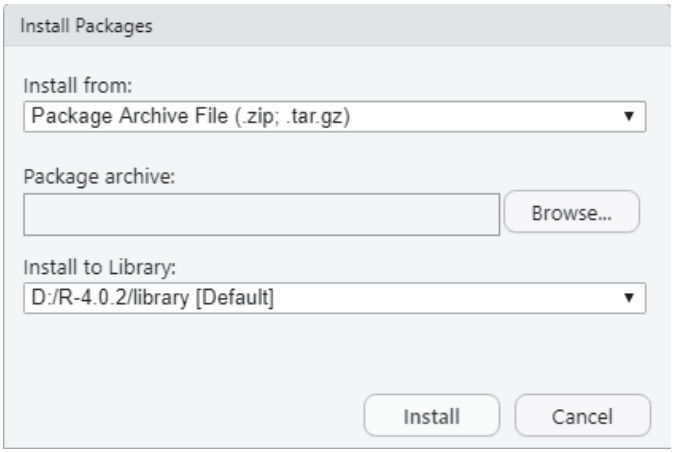


图 1.5: 手动安装扩展包

手动安装包，经常容易安装失败，通常是因为没有先安装该包的依赖包，故需要去包的网页查询其依赖包，确定若未安装，需要先安装它们。这往往又涉及到依赖包的依赖包，所以最好不要手动安装包。另外，尽量用最新版本的 R 会减少很多安装包失败。

Github 也是 R 扩展包的较大的来源，有些作者自己开发的 R 包只放在 Github，也有很多 CRAN R 包的最新开发版都位于 Github。可以先安装 devtools 或 remotes 包，再用其 `install_github()` 安装 Github 来源的包：

```
devtools::install_github("hadlley/dplyr")  # 或者
remotes::install_github("hadlley/dplyr")
```

`::` 前面是包名，这是不单独加载包，而使用包中函数的写法。

`hadlley` 为 Gighub 用户名，`dplyr` 为该用户的名为 dplyr 的 repository（仓库），也是包名。

若网络等原因，导致直接从 Github 安装包失败，也可以将整个包文件夹从网页下载下来，解压缩到当前路径（或提供完整路径），再从本地安装它：

```
install.packages(" 解压文件夹名", repos=NULL, type="source")
```

另外，生信领域在 R 中自成一派，有专门的包的大本营：

https://www.bioconductor.org

先安装 BiocManager 包，再用 `install()` 函数安装 bioconductor 来源的包：

```
BiocManager::install("openxlsx")
```

实用场景

R 包默认都安装在 .../R-4.0.x/library 路径下。

你在自己电脑上搭建好 R 语言环境，并安装好了很多常用包，然后你想到一台没有 R 环境、没有联网的电脑上复现你的代码。

方法非常简单：你只需要在那台电脑安装相同版本的 R 软件，安装到相同路径下，将新的 library 文件夹完全替换为你电脑里的 library 文件夹即可[a]，这样运行起 R 代码与在你电脑没有任何区别。

[a]可以用添加压缩包再解压的方式，速度能快一些。

加载包

```
library(openxlsx)
```

更新包

```
update.packages("openxlsx")
update.packages()          # 更新所有包
```

删除包

```
remove.packages("openxlsx")
```

获取或设置当前路径

```
getwd()
```

```
## [1] "E:/Rmarkdown/R_Programing_book"
```

```
setwd("D:/R-4.0.2/tests")
getwd()
```

```
## [1] "D:/R-4.0.2/tests"
```

特别注意：路径中的 \ 必须用 / 或 \\ 代替。

赋值

R 标准语法中赋值不是用 = ，而是 <- 或 ->

```
x <- 1:10
x + 2
```

```
##  [1]  3  4  5  6  7  8  9 10 11 12
```

R 也允许用 = 赋值，建议用更现代和简洁的 = 赋值。

R 是一种基于对象的向量化编程语言，即在定义类的基础上，创建与操作对象；数值向量、函数、图形等都是对象。

基本运算

- 数学运算：
  - `+ - * /`、`^`（求幂）、`%%`（求模[1]）、`%/%`（整除）
- 比较运算
  - `>`、`<`、`>=`、`<=`、`==`、`!=`
  - `identical(x,y)` —— 判断两个对象是否严格相等；
  - `all.equal(x,y)` 或 `dplyr::near(x,y)` —— 判断两个浮点数是否近似相等（误差 1.5e8）

```
0L == 0
```

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

```
identical(0L, 0)
```

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

```
sqrt(2)^2 == 2
```

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

```
identical(sqrt(2)^2, 2)
```

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

```
all.equal(sqrt(2)^2, 2)
```

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

```
dplyr::near(sqrt(2)^2, 2)
```

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

- 逻辑运算：
  - `|`（或），`&`（与），`!`（非），`xor()`（异或）

[1]可以关于小数求模，例如 `5.4 %% 2.3` 为 0.8

`&&` 和 `||` 是短路运算，即遇到 `TRUE`（`FALSE`）则返回 `TRUE`（`FALSE`）而不继续往下计算；
而 `&` 和 `|` 是向量运算符，对向量中所有元素分别进行运算。

基本数据类型

- R 中的基本数据类型包括：
  - `numeric` —— 数值型，又分为 `integer`（整数型）和 `double`（浮点型）
  - `logical` —— 逻辑型，只有 TRUE 和 FALSE，或 T 和 F
  - `character` —— 字符型，引号[2] 括起来的若干字符
- R 中用 NA 表示缺失值，NULL 表示空值，NaN 表示非数，Inf 表示无穷大
- 对于 R 中大多数函数，NA 具有传染性，即 NA 参与的运算，结果会变成 NA
- R 中注释一行代码用 #
- 可用函数 `class(x)` / `typeof(x)` / `mode(x)` 查看对象 `x` 的类型
  - 在展现数据的细节上，`mode()` < `class()` < `typeof()`
  - `str(x)` 显示对象 `x` 的结构

保存和载入数据

```
save(x, file = "dat.Rda")
load("dat.Rda")
```

关于相对路径与绝对路径

编程中的文件路径，可以用绝对路径也可以用相对路径。

绝对路径，是从盘符开始的完整路径：比如 `E:/R 语言/datas/a123.csv`。

相对路径，是指相对于当前路径的路径，因为通常操作的文件都是在当前路径，那么“从盘符到当前路径”这部分是大家所共有的，所以可以省略不写，只写从当前路径再往下的路径即可。比如，当前文件夹 `E:/R 语言` 中有 `datas` 文件夹，里面有数据文件 `a123.csv`，要写能访问到它的路径，只需写 `datas/a123.csv`。

清屏和清除内存变量

Ctrl + L 或单击命令窗口右上角的小刷子可对命令窗口清屏。

若要清除当前变量，用：

```
rm(x)                          # 清除变量 x
rm(list = ls(all = TRUE))      # 清除所有当前变量
```

[2] R 中单双引号通用。

注：单击 Environment 窗口的小刷子也是清除所有当前变量。

获取帮助

编程语言最好的学习资料就是帮助。

- 函数帮助

命令窗口执行：

```
?plot
```

则在 help 窗口打开 `plot()` 函数的帮助: 包括函数来自哪个包、函数的描述、参数说明、更多解释、实例等。

- 在线帮助（需联网）

若想根据某算法的名字或关键词，搜索哪个包能实现该算法：

```
RSiteSearch("network")
```

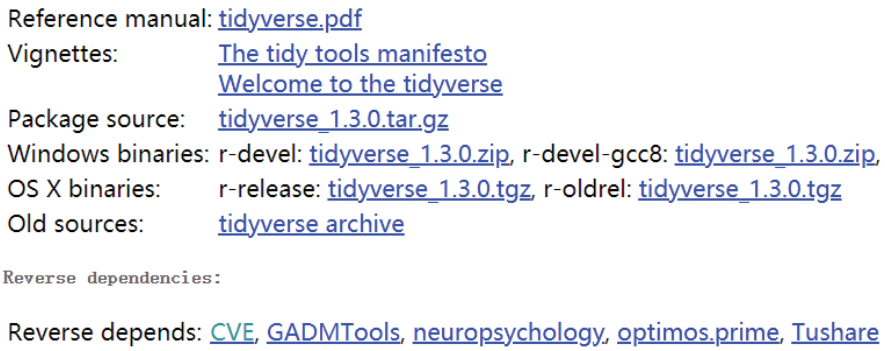
注：很奇怪，现在只能查一个单词，在打开的网页，可以输入多个单词查询

- 其它主要网络资源
  R 官方镜像站

https://mirrors.tongji.edu.cn/CRAN

下的各种资源，建议自己去发掘。

比如，最常用的是包的帮助文档：在镜像站，点左侧的 Packagess，再点 sort by name，则出现所有可用的 CRAN 包列表。点击某个包名，则进入该包的介绍页：


Reference manual: tidyverse.pdf
Vignettes: The tidy tools manifesto
Welcome to the tidyverse
Package source: tidyverse_1.3.0.tar.gz
Windows binaries: r-devel: tidyverse_1.3.0.zip, r-devel-gcc8: tidyverse_1.3.0.zip,
OS X binaries: r-release: tidyverse_1.3.0.tgz, r-oldrel: tidyverse_1.3.0.tgz
Old sources: tidyverse archive

Reverse dependencies:

Reverse depends: CVE, GADMTools, neuropsychology, optimos.prime, Tushare

Reference manual 为参考手册，包含该包所有函数和自带数据集的说明，供查阅使用；Vignettes （若有），是包的作者写的使用文档，它是该包的最佳学习资料。

在使用 R 语言过程中遇到各种问题，都建议优先用 Google 搜索相应关键词，更容易找到答案。另外，Github 是丰富的程序代码仓库，在 Google 搜索时，加上 github 关键词，可能有意想不到的收获。
其它开放的 R 社区：

- Stack overflow: https://stackoverflow.com/questions/tagged/r
- R-Bloggers: https://www.r-bloggers.com
- Tidyverse: https://www.tidyverse.org
- Rstudio: https://rstudio.com
- 统计之都：https://d.cosx.org

R Script 与 R Project

R 脚本是单个可执行的 R 代码文件，后缀名为.R，单击 `New File` 按钮，选择 R Script，或使用快捷键 `Ctrl + Shift + N`, 则新建 R 脚本。

R 脚本中都是可执行的 R 代码 + 注释，选中部分代码，点击 Run 运行选中的代码。

R 工程是完成某个项目或任务的一系列文件的合集（文件夹），包括数据文件、若干 R 脚本及其它附件，其中包含一个 `*.Rproj` 文件；

强烈建议使用 R 工程，它能方便系统地管理服务于共同目的一系列的文件，可以方便移动位置甚至是移到其它电脑，而不需要做任何路径设置就能成功运行。

创建 R 工程：单击 Create a Project 按钮

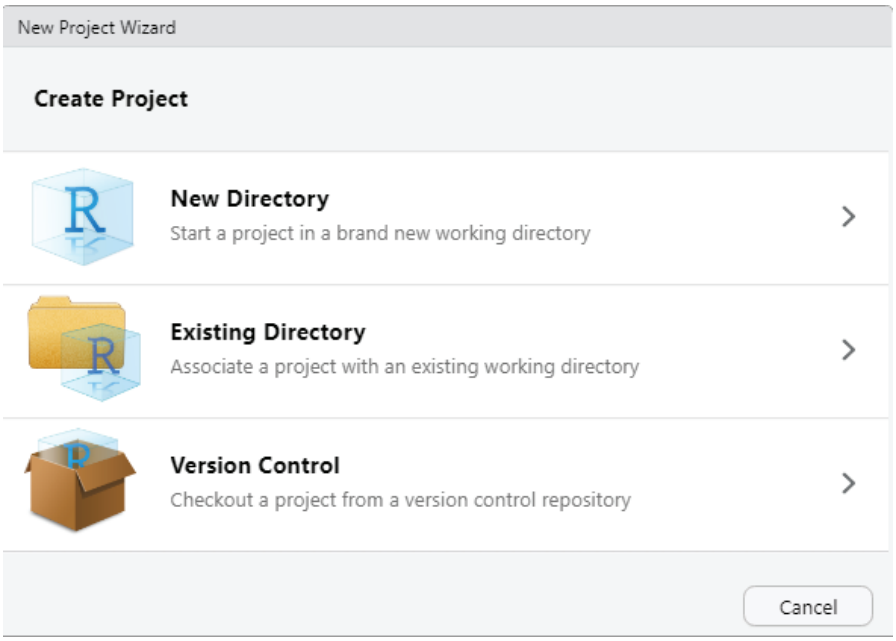


图 1.6: 创建 R Project

若在某个已存在的文件夹下创建工程，则选择 Existing Directory；若需要新建文件

夹创建工程，则选择 New Directory。

创建完成后，在文件夹下出现一个 `*.Rproj` 文件，双击它（关联 RStudio 打开），则进入该 R 工程，做各种具体访问、编辑文件和运行脚本等操作。

Rmarkdown

后缀名为 `.Rmd` 的交互式文档，是 `markdown` 语法与 R 脚本的结合，可以将可执行 R 代码和不可执行的文字叙述，融为一个文件。

单击 New File 按钮，选择 R Markdown 创建 Rmarkdown，建议优先使用自带和来自网络的现成模板。

Rmarkdown 适合编写包含 R 语言代码的学习笔记、演示文档、论文、书籍等，可以生成 `docx`，`pptx`，`html`，`pdf` 等多种文档格式。更多 Rmarkdown 内容将在第五章展开讨论。

1.2 数据结构 i：向量、矩阵、多维数组

数据结构是为了便于存储不同类型的数据而设计的**数据容器**。学习数据结构，就是要把各个数据容器的特点、适合存取什么样的数据理解透彻，只有这样才能在实际中选择最佳的数据容器，数据容器选择的合适与否，直接关系到代码是否高效简洁，甚至能否解决问题。

R 中常用的数据结构可划分为：

- 同质数据类型（homogeneous），即所存储的一定是相同类型的元素，包括向量、矩阵、多维数组；
- 异质数据类型（heterogeneous），即可以存储不同类型的元素，这大大提高了存储的灵活性，但同时也降低了存储效率和运行效率，包括列表、数据框。

另外，还有字符串、日期时间数据、时间序列数据、空间地理数据等。

R 中的数据结构还有一种从**广义向量（可称之为序列）**[3]的角度的划分：

- **原子向量**：各个值都是同类型的，包括 6 种类型：`logical`、`integer`、`double`、`character`、`complex`、`raw`，其中 `integer` 和 `double` 也统称为 `numeric`；
- **列表**：各个值可以是不同类型的，NULL 表示空向量（长度为 0 的向量）

向量都有两个属性：`type`（类型）、`length`（长度）；还能以属性的方式向向量中任意添加额外的 `metadata`（元数据），属性可用来创建扩展向量，以执行一些新的操作。常用的扩展向量有：

- 基于整数型向量构建的因子

[3]由一系列可以根据位置索引的元素构成，元素可以很复杂和不同类型。

- 基于数值型向量构建的日期和日期时间
- 基于数值型向量构建的时间序列
- 基于列表构建的数据框和 tibble

> 列表是序列，从这个角度有助于理解 `purrr::map_*()` 系列的泛函式编程。

1.2.1 向量（一维数据）

向量是由一组相同类型的原始值构成的序列，可以是一组数值、一组逻辑值、一组字符串等。

常用的向量有：数值向量、逻辑向量、字符向量。

1. 数值向量

数值向量就是由数值组成的向量，单个数值是长度为 1 的数值向量

```
x = 1.5
x
```

```
## [1] 1.5
```

可以用 `numeric()` 来创建全为 0 的指定长度的数值向量：

```
numeric(10)
```

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

R 中经常用函数 `c()` 实现将多个对象合并到一起：

```
c(1, 2, 3, 4, 5)
```

```
## [1] 1 2 3 4 5
```

```
c(1, 2, c(3, 4, 5))    # 将多个数值向量合并成一个数值向量
```

```
## [1] 1 2 3 4 5
```

> 创建等差的数值向量，用 `:` 或者函数 `seq()`，基本格式为：
>
> ```
> seq(from, to, by, length.out, along.with, ...)
> ```
>
> `from`：设置首项（默认为 1）；
> `to`：设置尾项；
> `by`：设置等差值（默认为 1 或 -1）；
> `length.out`：设置序列长度；
> `along.with`：以该参数的长度作为序列长度。

```r
1:5                     # 同 seq(5) 或 seq(1,5)
```

```
## [1] 1 2 3 4 5
```

```r
seq(1, 10, 2)           # 从 1 开始，到 10 结束，步长为 2
```

```
## [1] 1 3 5 7 9
```

```r
seq(3, length.out=10)
```

```
##  [1]  3  4  5  6  7  8  9 10 11 12
```

创建重复的数值向量用函数 rep()，基本格式为：

```r
rep(x, times,length.out, each, ...)
```

x：为要重复的序列；
times：设置序列重复次数；
length.out：设置产生的序列的长度；
each：设置每个元素分别重复的次数（默认为 1）。

```r
x = 1:3
rep(x, 2)
```

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

```r
rep(x, each = 2)
```

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

```r
rep(x, c(2, 1, 2))              # 按照规则重复序列中的各元素
```

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

```r
rep(x, each = 2, length.out = 4)
```

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

```r
rep(x, each = 2, times = 3)
```

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

注意，R 中两个不同长度的向量做运算，短的会自动循环补齐以配合长的。

```r
2:3 + 1:5
```

```
## [1] 3 5 5 7 7
```

2. 逻辑向量

逻辑向量，是一组逻辑值（TRUE 或 FALSE，或简写为 T 或 F）的向量。

```
c(1, 2) > c(2, 1)          # 等价于 c(1 > 2, 2 > 1)
```

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

```
c(2, 3) > c(1, 2, -1, 3)  # 等价于 c(2 > 1, 3 > 2, 2 > -1, 3 > 3)
```

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

除了比较运算符外，还可以用 %in% 判断元素是否属于集合：

```
c(1, 4) %in% c(1, 2, 3)   # 左边向量每一个元素是否属于右边集合
```

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

> match(v1, v2) 逐个检查向量 v1 中元素是否在向量 v2 中，若是则返回该元素，否则返回 NA。

3. 字符向量

字符（串）向量，是一组字符串组成的向量，R 中单引号和双引号都可以用来生成字符向量。

```
"hello, world!"
```

```
## [1] "hello, world!"
```

```
c("Hello", "World")
```

```
## [1] "Hello" "World"
```

```
c("Hello", "World") == "Hello, World"
```

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

要想字符串中出现单引号或双引号，需要用转义符 \ 来做转义，或者单双引号错开，用函数 cat() 生成字符串：

```
cat("Is \"You\" a Chinese name?")
```

```
# Is "You" a Chinese name?
```

```
cat('Is "You" a Chinese name?')
```

```
# Is "You" a Chinese name?
```

```
'Is "You" a Chinese name?'
```

```
# [1] "Is \"You\" a Chinese name?"
```

> R 中还有不常用的复数向量、原（raw）向量。

4. 访问向量子集

即访问向量的一些特定元素或者某个子集。注意，R 中的索引是从 1 开始的。

使用元素的位置来访问：

```
v1 = c(1, 2, 3, 4)
v1[2]                # 第 2 个元素
v1[2:4]              # 第 2-4 个元素
v1[-3]               # 除了第 3 个之外的元素
```

也可以放任意位置的数值向量，但是注意不能既放正数又放负数：

```
v1[c(1,3)]
v1[c(1, 2, -3)]     # 报错
```

访问不存在的位置也是可以的，返回 NA：

```
v1[3:6]
```

使用逻辑向量来访问，输入与向量相同长度的逻辑向量，以此决定每一个元素是否要被获取：

```
v1[c(TRUE, FALSE, TRUE, FALSE)]
```

这可以引申为“根据条件访问向量子集”：

```
v1[v1 <= 2]        # 同 v1[which(v1 <= 2)] 或 subset(v1, v1<=2)
v1[v1 ^ 2 - v1 >= 2]
which.max(v1)      # 返回向量 v1 中最大值所在的位置
which.min(v1)      # 返回向量 v1 中最小值所在的位置
```

5. 对向量子集赋值，替换相应元素

对向量子集赋值，就是先访问到向量子集，再赋值。

```
v1[2] = 0
v1[2:4] = c(0, 1, 3)
v1[c(TRUE, FALSE, TRUE, FALSE)] = c(3, 2)
v1[v1 <= 2] <- 0
```

注意，若对不存在的位置赋值，前面将用 NA 补齐：

```
v1[10] <- 8
v1
```

6. 对向量元素命名

可以在创建向量的同时对其每个元素命名：

```
x <- c(a = 1, b = 2, c = 3)
x
```

```
## a b c
## 1 2 3
```

命名后，就可以通过名字来访问向量元素：

```
x[c("a", "c")]
x[c("a", "a", "c")]    # 重复访问也是可以的
x["d"]                 # 访问不存在的名字
```

获取向量元素的名字：

```
names(x)
```

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

更改向量元素的名字：

```
names(x) <- c("x", "y", "z")
x["z"]
```

```
## z
## 3
```

移除向量元素的名字：

```
names(x) <- NULL
x
```

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

[ ] 与 [[ ]] 的区别

[ ] 可以提取对象的子集，[[ ]] 可以提取对象中的元素。

二者的区别：以向量为例，可以将一个向量比作 10 盒糖果，你可以使用 [ ] 获取其中的 3 盒糖果，使用 [[ ]] 打开盒子并从中取出一颗糖果。

对于未对元素命名的向量，使用 [ ] 和 [[ ]] 取出一个元素会产生相同的结果。但已对元素命名的向量，二者会产生不同的结果：

```
x <- c(a = 1, b = 2, c = 3)
x["a"]          # 取出标签为"a" 的糖果盒
```

```
## a
## 1
```

```
x[["a"]]        # 取出标签为"a" 的糖果盒里的糖果
```

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

由于 [[ ]] 只能用于提取出一个元素，因此不适用于提取多个元素的情况，所以 [[ ]] 不能用于负整数，因为负整数意味着提取除特定位置之外的所有元素。使用含有不存在的位置或名称来创建向量子集时将会产生缺失值。但当使用 [[ ]] 提取一个位置超出范围或者对应名称不存在的元素时，该命令将会无法运行并产生错误信息。

以下三个语句会报错：

```
x[[c(1, 2)]]
x[[-1]]
x[["d"]]
```

7. 对向量排序

向量排序函数 sort()，基本格式为：

```
sort(x, decreasing, na.last, ...)
```

- x：为排序对象（数值型或字符型）；
- decreasing：默认为 FALSE 即升序，TURE 为降序；
- na.last：默认为 FALSE，若为 TRUE，则将向量中的 NA 值放到序列末尾。

函数 order()，返回元素排好序的索引，以其结果作为索引访问元素，正好是排好序的元素。

函数 rank()，返回值是该向量中对应元素的“排名”。

```
x = c(1,5,8,2,9,7,4)
sort(x)
```

```
## [1] 1 2 4 5 7 8 9
```

```r
order(x)      # 默认升序，排名第 2 的元素在原向量的第 4 个位置
```

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

```r
x[order(x)]  # 同 sort(x)
```

```
## [1] 1 2 4 5 7 8 9
```

```r
rank(x)       # 默认升序，第 2 个元素排名第 4 位
```

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

还有函数 `rev()`，可将序列进行反转，即 `1,2,3` 变成 `3,2,1`。

1.2.2 矩阵（二维数据）

矩阵是一个用两个维度表示和访问的向量。因此，适用于向量的性质和方法大多也适用于矩阵：矩阵也要求元素是同一类型，数值矩阵、逻辑矩阵等。

1. 创建矩阵

函数 `matrix()` 将一个向量创建为矩阵，其基本格式为：

```r
matrix(x, nrow, ncol, byrow, dimnames, ...)
```

- `x`：为数据向量作为矩阵的元素；
- `nrow`：设定行数；
- `ncol`：设定列数；
- `byrow`：设置是否按行填充，默认为 `FALSE`（按列填充）；
- `dimnames`：用字符型向量表示矩阵的行名和列名。

```r
matrix(c(1, 2, 3,
         4, 5, 6,
         7, 8, 9), nrow = 3, byrow = FALSE)
```

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

```r
matrix(c(1, 2, 3,
         4, 5, 6,
         7, 8, 9), nrow = 3, byrow = TRUE)
```

```
##      [,1] [,2] [,3]
```

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

对矩阵的行列命名：

```
matrix(1:9, nrow = 3, byrow = TRUE,
       dimnames = list(c("r1","r2","r3"), c("c1","c2","c3")))
```

```
##    c1 c2 c3
## r1  1  2  3
## r2  4  5  6
## r3  7  8  9
```

也可以创建后再命名：

```
m1 = matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), ncol = 3)
rownames(m1) = c("r1", "r2", "r3")
colnames(m1) = c("c1", "c2", "c3")
m1
```

```
##    c1 c2 c3
## r1  1  4  7
## r2  2  5  8
## r3  3  6  9
```

特殊矩阵：

```
diag(1:4, nrow = 4)      # 对角矩阵
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    0    2    0    0
## [3,]    0    0    3    0
## [4,]    0    0    0    4
```

函数 `as.vector()`，可将矩阵转化为向量，元素按列读取。

2. 访问矩阵子集

矩阵是用两个维度表示和访问的向量，可以用一个二维存取器 `[ , ]` 来访问，这类似于构建向量子集时用的一维存取器 `[]`。

可以为每个维度提供一个向量来确定一个矩阵的子集。方括号中的第 1 个参数是行选择器，第 2 个参数是列选择器。与构建向量子集一样，可以在两个维度中使用数

值向量、逻辑向量和字符向量。

```
m1[1,2]               # 提取第 1 行，第 2 列的单个元素
m1[1:2, 2:4]          # 提取第 1 至 2 行，第 2 至 4 列的元素
m1[c("r1","r3"), c("c1","c3")]  # 提取行名为 r1 和 r3，列名为 c1 和 c3 的元素
```

若一个维度空缺，则选出该维度的所有元素：

```
m1[1,]                # 提取第 1 行，所有列元素
m1[,2:4]              # 提取所有行，第 2 至 4 列的元素
```

负数表示在构建矩阵子集时可排除该位置，这和向量中的用法一致：

```
m1[-1,]               # 提取除了第 1 行之外的所有元素
m1[,-c(2,4)]          # 提取除了第 2 和 4 列之外的所有元素
```

注意，矩阵是一个用两个维度表示和访问的向量，但它本质上仍然是一个向量。因此，向量的一维存取器也可以用来构建矩阵子集：

```
m1[3:7]
```

```
## [1] 3 4 5 6 7
```

由于向量只包含相同类型的元素，矩阵也是如此。所以它们的操作方式也相似。若输入一个不等式，则返回同样大小的逻辑矩阵：

```
m1 > 3
```

```
##       c1   c2   c3
## r1 FALSE TRUE TRUE
## r2 FALSE TRUE TRUE
## r3 FALSE TRUE TRUE
```

根据它就可以选择矩阵元素或赋值：

```
m1[m1 > 3]    # 注意选出来的结果是向量
```

```
## [1] 4 5 6 7 8 9
```

矩阵运算

- `A+B`，`A-B`，`A*B`，`A/B`：矩阵四则运算，要求矩阵同型，类似 Matlab 中的点运算，分别将对应位置的元素做四则运行；
- `A %*% B`：矩阵乘法，要求 A 的列数 = B 的行数。

1.2.3 多维数组（多维数据）

向量/矩阵向更高维度的自然推广。具体来说，多维数组就是一个维度更高（通常大于 2）、可访问的向量。数组也要求元素是同一类型。

1. 创建多维数组

函数 `array()` 将一个向量创建为多维数组，基本格式为：

```
array(x, dim, dimnames, ...)
```

- `x`：为数据向量作为多维数组的元素；
- `dim`：设置多维数组各维度的维数；
- `dimnames`：设置多维数组各维度的名称。

```
a1 = array(1:24, dim = c(3, 4, 2))
a1
```

```
## , , 1
##
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
##
## , , 2
##
##      [,1] [,2] [,3] [,4]
## [1,]   13   16   19   22
## [2,]   14   17   20   23
## [3,]   15   18   21   24
```

也可以在创建数组时对每个维度进行命名：

```
a1 = array(1:24, dim = c(3, 4, 2),
           dimnames=list(c("r1","r2","r3"),
                         c("c1","c2","c3","c4"), c("k1","k2")))
```

或者创建之后再命名[4]

```
a1 = array(1:24, dim = c(3, 4, 2))
dimnames(a1) = list(c("r1","r2","r3"),
```

[4] `list` 是创建列表（见下节）。

```
                    c("c1","c2","c3","c4"), c("k1","k2"))
```

2. 访问多维数组子集

第 3 个维度姑且称为“页”

```
a1[2,4,2]           # 提取第 2 行，第 4 列，第 2 页的元素
a1["r2","c4","k2"]  # 提取第 r2 行，第 c4 列，第 k2 页的元素
a1[1,2:4,1:2]       # 提取第 1 行，第 2 至 4 列，第 1 至 2 页的元素
a1[,,2]             # 提取第 2 页的所有元素
dim(a1)             # 返回多维数组 a1 的各维度的维数
```

> 在想象多维数组时，为了便于形象地理解，可以将其维度依次想象为与“书”相关的概念：行、列、页、本、层、架、室.....

本节部分内容参阅 (任坤2017) 和 (Hadley Wickham 2017).

1.3 数据结构 ii：列表、数据框、因子

1.3.1 列表（list）

列表，可以包含不同类型的对象，甚至可以包括其他列表。列表的灵活性使得它非常有用。

例如，用 R 拟合一个线性回归模型，其返回结果就是一个列表，其中包含了线性回归的详细结果，如线性回归系数（数值向量）、残差（数值向量）、QR 分解（包含一个矩阵和其他对象的列表）等。因为这些结果全都被打包到一个列表中，就可以很方便地提取所需信息，而无需每次调用不同的函数。

列表最大的好处就是，它能够将多个不同类型的对象打包到一起，使得可以根据位置和名字访问它们。

1. 创建列表

可以用函数 `list()` 创建列表。不同类型的对象可以放入同一个列表中。

例如，创建了一个列表，包含 3 个成分：一个单元素的数值向量、一个两元素的逻辑向量和一个长度为 3 的字符向量：

```
l0 = list(1, c(TRUE, FALSE), c("a", "b", "c"))
l0
```

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

```
##
## [[2]]
## [1]  TRUE FALSE
##
## [[3]]
## [1] "a" "b" "c"
```

可以在创建列表时，为列表的每个成分指定名字：

```
l1 = list(A = 1, B = c(TRUE, FALSE), C = c("a", "b", "c"))
l1
```

```
## $A
## [1] 1
##
## $B
## [1]  TRUE FALSE
##
## $C
## [1] "a" "b" "c"
```

也可以创建列表后再对列表成分命名或修改名字：

```
names(l1) = NULL       # 移除列表成分的名字
names(l1) = c("x","y","z")
```

2. 从列表中提取成分的内容

提取列表中成分下的内容，最常用的方法是用 `$`，通过成分名字来提取该成分下的内容：

```
l1$y
l1$m                   # 访问不存在的成分 m，将会返回 NULL
```

也可以用 `[[n]]` 来提取列表第 `n` 个成分的内容，`n` 也可以换成成分的名字：

```
l1[[2]]                # 同 l1[["y"]]
```

用 `[[]]` 提取列表中某个成分的内容更加灵活，可用在函数调用中，通过参数来传递成分名字：

```
p = "y"                # 想要提取其内容的成分名字
l1[[p]]
```

3. 提取列表子集

经常也需要从列表中提取多个成分及其内容，由这些成分组成的列表构成了原列表的一个子集。

就像提取向量和矩阵的子集一样，提取一个列表子集是用 `[]`，可以取出列表中的一些成分，作为一个新的列表。

`[]` 中可以用字符向量表示成分名字，用数值向量表示成分位置，或用逻辑向量指定是否选择，来取出列表成分。

```
l1["x"]                    # 同 l1[1]
l1[c("x", "z")]            # 同 l1[c(1, 3)], l1[c(TRUE, FALSE, TRUE)]
```

> 用 `[]` 提取若干成分时，返回列表的子集，还是一个列表；用 `[[ ]]` 提取单个成分的元素，返回的是对应成分的元素。
> 总之，`[]` 提取对象的子集，类型仍是该对象；`[[]]` 提取对象的内容（下一级元素）。

4. 对列表的成分赋值

即先访问（提取）到列表的成分，再赋以相应的值。注意，若给一个不存在的成分赋值，列表会自动地在对应名称或位置下增加一个新成分。

```
l1$x = 0   # 将列表的成分 x 赋值为 0
```

也可以同时给多个列表成分赋值：

```
l1[c("x", "y")] = list(x = "new value for y", y = c(3, 1))
```

若要移除列表中的某些成分，只需赋值为 `NULL`：

```
l1[c("z", "m")] = NULL
```

5. 列表函数

用函数 `as.list()` 可将向量转换成列表：

```
l2 = as.list(c(a = 1, b = 2))
l2
```

```
## $a
## [1] 1
##
## $b
## [1] 2
```

用去列表化函数 `unlist()`，可将一个列表打破成分界限，强制转换成一个向量[5]：

```
unlist(l2)
```

```
## a b
## 1 2
```

`tidyverse` 系列中的 `purrr` 包为方便操作列表，提供了一系列列表相关的函数，建议读者查阅使用：

- `pluck()`：同 [[提取列表中的元素
- `keep()`: 保留满足条件的元素
- `dicard()`: 删除满足条件的元素
- `compact()`: 删除列表中的空元素
- `append()`：在列表末尾增加元素
- `flatten()`: 摊平列表（只摊平一层）

1.3.2 数据框（数据表）

R 语言中做统计分析的样本数据，都是按数据框类型操作的。

数据框是指有若干行和列的数据集，它与矩阵类似，但并不要求所有列都是相同的类型；本质上讲，数据框就是一个列表，它的每个成分都是一个向量，并且长度相同，以表格的形式展现。总之，**数据框是由列向量组成、有着矩阵形式的列表**。

数据框与最常见的数据表是一致的：每一列代表一个变量属性，每一行代表一条样本数据：

姓名	性别	年龄	专业
Ken	Male	24	Finance
Ashley	Female	25	Statistics
Jennifer	Female	23	Computer Science

图 1.7: 数据表样式

R 中自带的数据框是 `data.frame`，建议改用更现代的数据框：`tibble`[6]。

Hadley 在 `tibble` 包中引入一种 `tibble` 数据框，以代替 `data.frame`；而且 `tidyverse` 包都是基于 `tibble` 数据框。

`tibble` 对比 `data.frame` 的优势：

- `tibble()` 比 `data.frame()` 做的更少：不改变输入变量的类型（**R 4.0.0 之前默认将字符串转化为因子！**），不会改变变量名，不会创建行名；

[5]若列表的成分具有不同类型，则自动向下兼容到统一类型。

[6]读者若习惯用 R 自带的 `data.frame`, 只需要换个名字，将 `tibble` 改为 `data.frame` 即可。

- tibble 对象的列名可以是 R 中的“非法名”：非字母开头、包含空格，但定义和使用变量时都需要用倒引号 ‘ 括起来；
- tibble 在输出时不自动显示所有行，避免大数据框时显示很多内容；
- 用 [] 选取列子集时，即使只选取一列，返回结果仍是 tibble，而不自动简化为向量。

1. 创建数据框

用 tibble() 根据若干列向量创建 tibble：

```
library(tibble)  # 或 tidyverse

persons <- tibble(
  Name = c("Ken", "Ashley", "Jennifer"),
  Gender = c("Male", "Female", "Female"),
  Age = c(24, 25, 23),
  Major = c("Finance", "Statistics", "Computer Science")
)
persons
```

```
## # A tibble: 3 x 4
##   Name     Gender   Age Major
##   <chr>    <chr>  <dbl> <chr>
## 1 Ken      Male      24 Finance
## 2 Ashley   Female    25 Statistics
## 3 Jennifer Female    23 Computer Science
```

用 tribble() 按行录入数据式创建 tibble：

```
tribble(
  ~Name, ~Gender, ~Age, ~Major,
  "Ken", "Male", 24, "Finance",
  "Ashley", "Female", 25, "Statistics",
  "Jennifer", "Female", 23, "Computer Science"
)
```

用 as_tibble() 将 data.frame, matrix, 各成分等长度的 list, 转换为 tibble。

数据框既是列表的特例，也是矩阵的推广，因此访问这两类对象的方式都适用于数据框。例如与矩阵类似，对数据框的行列重新命名：

```
df = tibble(id = 1:4,
            level = c(0, 2, 1, -1),
```

```
            score = c(0.5, 0.2, 0.1, 0.5))
names(df) = c("id", "x", "y")
df
```

```
## # A tibble: 4 x 3
##      id     x     y
##   <int> <dbl> <dbl>
## 1     1     0   0.5
## 2     2     2   0.2
## 3     3     1   0.1
## 4     4    -1   0.5
```

2. 提取数据框的元素、子集

数据框是由列向量组成、有着矩阵形式的列表，所以可以用两种操作方式来访问数据框的元素和子集。

（1）以列表方式提取数据框的元素、子集

若把数据框看作是由向量组成的列表，则可以沿用列表的操作方式来提取元素或构建子集。例如，可以用 `$` 按列名来提取某一列的值，或者用 `[[]]` 按照位置或列名提取。

例如，提取列名为 `x` 列的值，得到向量：

```
df$x                  # 同 df[["x"]], df[[2]]
```

```
## [1]  0  2  1 -1
```

以列表形式构建子集完全适用于数据框，同时也会生成一个新的数据框。提取子集的操作符 `[]` 允许用数值向量表示列的位置，用字符向量表示列名，或用逻辑向量指定是否选择。

例如，提取数据框的一列或多列，得到子数据框：

```
df[1]                 # 提取第 1 列，同 df["id"]
```

```
## # A tibble: 4 x 1
##      id
##   <int>
## 1     1
## 2     2
## 3     3
## 4     4
```

```
df[1:2]              # 同 df[c("id", "x")], df[c(TRUE, TRUE, FALSE)]
```

```
## # A tibble: 4 x 2
##      id     x
##   <int> <dbl>
## 1     1     0
## 2     2     2
## 3     3     1
## 4     4    -1
```

（2）以矩阵方式提取数据框的元素、子集

以列表形式操作并不支持行选择。以矩阵形式操作更加灵活，若将数据框看作矩阵，其二维形式的存取器可以很容易地获取一个子集的元素，同时支持列选择和行选择。

换句话说，可以使用 `[i, j]` 指定行或列来提取数据框子集，`[ , ]` 内可以是数值向量、字符向量或者逻辑向量。

若行选择器为空，则只选择列（所有行）：

```
df[, "x"]
```

```
## # A tibble: 4 x 1
##       x
##   <dbl>
## 1     0
## 2     2
## 3     1
## 4    -1
```

```
df[, c("x", "y")]   # 同 df[, 2:3]
```

```
## # A tibble: 4 x 2
##       x     y
##   <dbl> <dbl>
## 1     0   0.5
## 2     2   0.2
## 3     1   0.1
## 4    -1   0.5
```

若列选择器为空，则只选择行（所有列）：

```
df[c(1,3),]
```

```
## # A tibble: 2 x 3
##      id     x     y
##   <int> <dbl> <dbl>
## 1     1     0   0.5
## 2     3     1   0.1
```

同时选择行和列：

```
df[1:3, c("id", "y")]
```

```
## # A tibble: 3 x 2
##      id     y
##   <int> <dbl>
## 1     1   0.5
## 2     2   0.2
## 3     3   0.1
```

根据条件筛选数据。例如用 `y >= 0.5` 筛选 `df` 的行，并选择 `id` 和 `y` 两列：

```
df[df$y >= 0.5, c("id","y")]
```

```
## # A tibble: 2 x 2
##      id     y
##   <int> <dbl>
## 1     1   0.5
## 2     4   0.5
```

按行名属于集合 `{x, y, w}` 来筛选 `df` 的行，并选择 `x` 和 `y` 两列：

```
ind = names(df) %in% c("x", "y", "w")
df[1:2, ind]
```

```
## # A tibble: 2 x 2
##       x     y
##   <dbl> <dbl>
## 1     0   0.5
## 2     2   0.2
```

3. 给数据框赋值

给数据框赋值，就是选择要赋值的位置，再准备好同样大小且格式匹配的数据，赋值给那些位置即可，所以同样有列表方式和矩阵方式。

（1）以列表方式给数据框赋值

用 $ 或 [[ ]] 对数据框的 1 列赋值

```
df$y = c(0.6,0.3,0.2,0.4)    # 同 d[["y"]] = c(0.4,0.5,0.2,0.8)
```

利用现有列，创建 (计算) 新列：

```
df$z = df$x + df$y
df
```

```
## # A tibble: 4 x 4
##      id     x     y     z
##   <int> <dbl> <dbl> <dbl>
## 1     1     0   0.5   0.5
## 2     2     2   0.2   2.2
## 3     3     1   0.1   1.1
## 4     4    -1   0.5  -0.5
```

```
df$z = as.character(df$z)    # 转换列的类型
df
```

```
## # A tibble: 4 x 4
##      id     x     y z
##   <int> <dbl> <dbl> <chr>
## 1     1     0   0.5 0.5
## 2     2     2   0.2 2.2
## 3     3     1   0.1 1.1
## 4     4    -1   0.5 -0.5
```

用 [] 可以对数据框的 1 列或多列进行赋值：

```
df["y"] = c(0.8,0.5,0.2,0.4)
df[c("x", "y")] = list(level = c(1,2,1,0),
                       score = c(0.1,0.2,0.3,0.4))
```

（2）以矩阵方式给数据框赋值

以列表方式对数据框进行赋值时，也是只能访问列。若需要更加灵活地进行赋值操作，可以以矩阵方式进行。

```
df[1:3,"y"] = c(-1,0,1)
df[1:2,c("x","y")] = list(level = c(0,0),
                          score = c(0.9,1.0))
```

4. 一些有用函数

函数 `str()` 或 `glimpse()` 作用在 R 对象上，显示该对象的结构：

```
str(persons)
```

```
## tibble [3 x 4] (S3: tbl_df/tbl/data.frame)
##  $ Name  : chr [1:3] "Ken" "Ashley" "Jennifer"
##  $ Gender: chr [1:3] "Male" "Female" "Female"
##  $ Age   : num [1:3] 24 25 23
##  $ Major : chr [1:3] "Finance" "Statistics" "Computer Science"
```

`summary()` 作用在数据框/列表上，将生成各列/成分的汇总信息：

```
summary(persons)
```

```
##      Name              Gender               Age
##  Length:3           Length:3           Min.   :23.0
##  Class :character   Class :character   1st Qu.:23.5
##  Mode  :character   Mode  :character   Median :24.0
##                                        Mean   :24.0
##                                        3rd Qu.:24.5
##                                        Max.   :25.0
##     Major
##  Length:3
##  Class :character
##  Mode  :character
##
##
##
```

经常需要将多个数据框（或矩阵）按行或按列进行合并。用函数 `rbind()`，增加行（样本数据），要求宽度（列数）相同；用函数 `cbind()`，增加列（属性变量），要求高度（行数）相同。

例如，向数据框 `persons` 数据框中添加一个人的新记录：

```
rbind(persons,
      tibble(Name = "John", Gender = "Male",
             Age = 25, Major = "Statistics"))
```

```
## # A tibble: 4 x 4
##   Name     Gender   Age Major
##   <chr>    <chr>  <dbl> <chr>
```

```
## 1 Ken      Male      24 Finance
## 2 Ashley   Female    25 Statistics
## 3 Jennifer Female    23 Computer Science
## 4 John     Male      25 Statistics
```

向 persons 数据框中添加两个新列表示每个人是否已注册和其手头的项目数量：

```
cbind(persons, Registered = c(TRUE, TRUE, FALSE),
               Projects = c(3, 2, 3))
```

```
##       Name Gender Age            Major Registered Projects
## 1      Ken   Male  24          Finance       TRUE        3
## 2   Ashley Female  25       Statistics       TRUE        2
## 3 Jennifer Female  23 Computer Science      FALSE        3
```

> bind() 和 cbind() 不会修改原始数据，而是生成一个添加了行或列的新数据框。更建议用 dplyr 包中的 bind_rows() 和 bind_cols()。

函数 expand.grid() 可生成多个属性水平值所有组合（笛卡尔积）的数据框：

```
expand.grid(type=c("A", "B"), class=c("M", "L", "XL"))
```

```
##   type class
## 1    A     M
## 2    B     M
## 3    A     L
## 4    B     L
## 5    A    XL
## 6    B    XL
```

1.3.3 因子（factor）

变量分为离散/分类型、连续/数值型（通常的数值变量，可带小数位）；分类型变量又分为：名义型（无顺序好坏之分的分类变量，如性别）、有序型（有顺序好坏之分的分类变量，如疗效）。

名义型和有序型的分类变量，在 R 语言中称为因子，因子本质上是一个带有水平（level）属性的整数向量，其中“水平”是指事前确定可能取值的有限集合。

因子提供了一个简单且紧凑的形式来处理分类数据，因子用水平来表示所有可能的取值，例如，性别有两个水平：男、女。

1. 创建因子

函数 factor() 用来创建因子，基本格式为：

```
factor(x, levels, labels, ordered, ...)
```

- x：为创建因子的数据向量；
- levels：指定因子的各水平值，默认为 x 中不重复的所有值；
- labels：设置各水平名称（前缀），与水平一一对应；
- ordered：设置是否对因子水平排序，默认 FALSE 为无序因子，TRUE 为有序因子；

该函数还包含参数 exclude：指定有哪些水平是不需要的（设为 NA）；nmax 设定水平数的上限。

```
x = c(" 男", " 女", " 男", " 男", " 女")
sex = factor(x)
sex
```

```
## [1] 男 女 男 男 女
## Levels: 男 女
```

```
levels(sex)                    # 访问因子水平
```

```
## [1] "男" "女"
```

```
levels(sex) = c("M", "F")      # 修改因子水平
sex
```

```
## [1] M F M M F
## Levels: M F
```

函数 gl() 用来生成有规律的水平值组合因子。对于多因素试验设计，用该函数可以生成多个因素完全组合，基本格式为：

```
gl(n, k, length, labels, ordered, ...)
```

- n：为因子水平个数；
- k：为同一因子水平连续重复次数；
- length：为总的元素个数，默认为 n * k，若不够则自动重复；
- labels：设置因子水平值；
- ordered：设置是否为有序，默认为 FALSE。

```
tibble(
  Sex = gl(2, 3, length=12, labels=c(" 男", " 女")),
  Class = gl(3, 2, length=12, labels=c(" 甲", " 乙", " 丙")),
```

```
  Score = gl(4, 3, length=12, labels=c(" 优", " 良", " 中", " 及格"))
)
```

```
## # A tibble: 12 x 3
##    Sex   Class Score
##    <fct> <fct> <fct>
##  1 男    甲    优
##  2 男    甲    优
##  3 男    乙    优
##  4 女    乙    良
##  5 女    丙    良
##  6 女    丙    良
##  7 男    甲    中
##  8 男    甲    中
##  9 男    乙    中
## 10 女    乙    及格
## 11 女    丙    及格
## 12 女    丙    及格
```

2. 使用因子

R 中因子是以整数型向量存储的，每个因子水平对应一个整数型的数。对字符型向量创建的因子，可以指定因子水平顺序，否则默认会按照字母顺序，再对应到整数型向量。

> 不能直接将因子数据当字符型操作，需要用 `as.character()` 转化。

考虑用一个变量存储字符串型的月份：

```
x1 = c("Dec", "Apr", "Jan", "Mar")
```

这有两个问题：

- 实际只需要 12 个月份值，对拼写错误也无能为力；
- 不会按照想要的方式排序：

```
sort(x1)
```

```
## [1] "Apr" "Dec" "Jan" "Mar"
```

若改用因子型就能避免上述问题。创建因子型，首先要创建一个有效的“水平值”向量：

```
month_levels = c("Jan", "Feb", "Mar", "Apr", "May", "Jun",
                 "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")
```

再来创建因子：

```
y1 = factor(x1, levels = month_levels)
y1
```

```
## [1] Dec Apr Jan Mar
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

```
sort(y1)
```

```
## [1] Jan Mar Apr Dec
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

并且任何不在水平集中的值都将转化为 NA，相当于“识错”：

```
x2 = c("Dec", "Apr", "Jam", "Mar")
y2 = factor(x2, levels = month_levels)
y2
```

```
# [1] Dec  Apr  <NA> Mar
# Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

有时候你可能更希望让水平的顺序与其在数据集中首次出现的次序相匹配，设置参数 levels = unique(x)：

```
f1 = factor(x1, levels = unique(x1))
f1
```

```
## [1] Dec Apr Jan Mar
## Levels: Dec Apr Jan Mar
```

3. 有用函数

> 函数 table()，可以统计因子各水平的出现次数（频数），也可以统计向量中每个不同元素的出现次数，返回结果为命名向量。

```
table(sex)
```

```
## sex
## M F
## 3 2
```

函数 cut()，用来做连续变量离散化：将数值向量切分为若干区间段，返回因子。基本格式为：

```
cut(x, breaks, labels, ...)
```

- x：为要切分的数值向量；
- breaks：切分的界限值构成的向量，或表示切分段数的整数。

该函数还包含参数 right 设置区间段是否左开右闭，include.lowest 设置是否包含下界，ordered_result 设置是否对结果因子排序。

```
Age = c(23,15,36,47,65,53)
cut(Age, breaks = c(0,18,45,100),
    labels = c("Young","Middle","Old"))
```

```
## [1] Middle Young  Middle Old    Old    Old
## Levels: Young Middle Old
```

tidyverse 系列中的 facats 包是专门为处理因子型数据而设计的，提供了一系列方便的函数，建议读者查阅使用：

- fct_count()：计算因子各水平频数、占比，可按频数排序
- fct_c()：合并多个因子的水平
- 改变因子水平的顺序：
  - fct_relevel()：手动对水平值重新排序
  - fct_infreq()：按高频优先排序
  - fct_inorder()：按水平值出现的顺序
  - fct_rev()：将顺序反转
  - fct_reorder()：根据其它变量或函数结果排序（绘图时有用）
- 修改水平：
  - fct_recode()：对水平值逐个重编码
  - fct_collapse()：推倒手动合并部分水平
  - fct_lump_*()：将多个频数小的水平合并为其它
  - fct_other()：将保留之外或丢弃的水平合并为其它
- 增加或删除水平：
  - fct_drop()：删除若干水平
  - fct_expand：增加若干水平
  - fct_explicit_na()：为 NA 设置水平

本节部分内容参阅 (Hadley Wickham 2017), (任坤2017), RStudio Cheatsheets: Factors with forcats.

1.4 数据结构 iii：字符串、日期时间

1.4.1 字符串

字符串是用双引号或单引号括起来的若干字符，建议用双引号，除非字符串中包含双引号。字符串构成的向量，简称为字符向量。

字符串处理不是 R 语言中的主要功能，但也是必不可少的，数据清洗、可视化等操作都会用到。

tidyverse 系列中的 stringr 包提供了一系列接口一致的、简单易用的字符串操作函数，足以代替 R 自带字符串函数。这些函数都是向量化的，即作用在字符向量上，对字符向量中的每个字符串做某种操作。

```
library(stringr)
```

1. 字符串的长度（包含字符个数）

```
str_length(c("a", "R for data science", NA))
```

```
## [1]  1 18 NA
```

```
str_pad(c("a", "ab", "abc"), 3)       # 填充到长度为 3
```

```
## [1] "  a" " ab" "abc"
```

```
str_trunc("R for data science", 10)   # 截断到长度为 10
```

```
## [1] "R for d..."
```

```
str_trim(c("a  ", "b  ", "a b"))      # 移除空格
```

```
## [1] "a"   "b"   "a b"
```

后三个函数都包含参数 side=c("both", "left", "right")) 设定操作的方向。

2. 字符串合并

```
str_c(..., sep = "", collapse = NULL)
```

- sep：设置间隔符，默认为空字符；-collapse：指定间隔符，将字符向量推倒合并为一个字符串。

```
str_c("x", 1:3, sep = "")  # 同 paste0("x", 1:3), paste("x", 1:3, sep="")
```

```
## [1] "x1" "x2" "x3"
```

```r
str_c("x", 1:3, collapse = "_")
```

```
## [1] "x1_x2_x3"
```

注：1:3 自动向下兼容以适应字符串运算，效果同 c("1","2","3")

将字符串重复 n 次，基本格式为：

```r
str_dup(string, times)
```

- string：为要重复的字符向量；
- times：为重复的次数。

```r
str_dup(c("A","B"), 3)
```

```
## [1] "AAA" "BBB"
```

```r
str_dup(c("A","B"), c(3,2))
```

```
## [1] "AAA" "BB"
```

3. 字符串拆分

```r
str_split(string, pattern)          # 返回列表
str_split_fixed(string, pattern, n) # 返回矩阵，n 控制返回的列数
```

- string：为要拆分的字符串；
- pattern：指定拆分的分隔符，可以是正则表达式。

```r
x = "10,8,7"
str_split(x, ",")
```

```
## [[1]]
## [1] "10" "8"  "7"
```

```r
str_split_fixed(x, ",", n = 2)
```

```
##      [,1] [,2]
## [1,] "10" "8,7"
```

4. 字符串格式化输出

只要在字符串内用 {变量名}，则函数 str_glue() 和 str_glue_data 就可以将字符串中的变量名替换成变量值，后者的参数 .x 支持引入数据框、列表等。

```r
str_glue("Pi = {pi}")
```

```
## Pi = 3.14159265358979
```

```
name = " 李明"
tele = "13912345678"
str_glue(" 姓名：{name}", " 电话号码：{tele}", .sep="；")
```

```
## 姓名： 李明；电话号码：13912345678
```

5. 字符串排序

```
str_sort(x, decreasing, locale, ...)
str_order(x, decreasing, locale, ...)
```

默认 decreasing = FALSE 表示升序，前者返回排好序的元素，后者返回排好序的索引；参数 locale 可设定语言，默认为 "en" 英语。

```
x = c("banana", "apple", "pear")
str_sort(x)
```

```
## [1] "apple"  "banana" "pear"
```

```
str_order(x)
```

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

```
str_sort(c(" 香蕉", " 苹果", " 梨"), locale = "ch")
```

```
## [1] "梨"   "苹果" "香蕉"
```

6. 检测匹配

str_detect(string, pattern, negate=FALSE) —— 检测是否存在匹配
str_which(string, pattern, negate=FALSE) —— 查找匹配的索引
str_count(string, pattern) —— 计算匹配的次数
str_locate(string, pattern) —— 定位匹配的位置
str_starts(string, pattern) —— 检测是否以 pattern 开头
str_ends(string, pattern) —— 检测是否以 pattern 结尾

- string：为要检测的字符串；
- pattern：为匹配的模式，可以是正则表达式；
- negate：默认为 FALSE 表示正常匹配，若为 TRUE 则反匹配（找不匹配）。

```
x
```

```
## [1] "banana" "apple"  "pear"
```

```
str_detect(x, "p")
```

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

```
str_which(x, "p")
```

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

```
str_count(x, "p")
```

```
## [1] 0 2 1
```

```
str_locate(x, "a.")   # 正则表达式，. 匹配任一字符
```

```
##      start end
## [1,]     2   3
## [2,]     1   2
## [3,]     3   4
```

7. 提取字符串子集

根据指定的起始和终止位置提取子字符串，基本格式为：

```
str_sub(string, start = 1, end = -1)
```

```
str_sub(x, 1, 3)
```

```
## [1] "ban" "app" "pea"
```

```
str_sub(x, 1, 5)     # 若长度不够，则尽可能多地提取
```

```
## [1] "banan" "apple" "pear"
```

```
str_sub(x, -3, -1)
```

```
## [1] "ana" "ple" "ear"
```

提取字符向量中匹配的字符串，基本格式为：

```
str_subset(string, pattern, negate=FALSE)
```

若 negate = TURE, 则返回不匹配的字符串。

```
str_subset(x, "p")
```

```
## [1] "apple" "pear"
```

8. 提取匹配的内容

```
str_extract(string, pattern)
str_match(string, pattern)
```

- str_extract() 只提取匹配的内容；
- str_match() 提取匹配的内容以及各个分组捕获，返回矩阵，每行对应于字符向量中的一个字符串，每行的第一个元素是匹配内容，其它元素是各个分组捕获，没有匹配则为 NA

```
x <- c("1978-2000", "2011-2020-2099")
pat <- "\\b(19|20)([0-9]{2})\\b"      # 正则表达式
str_extract(x, pat)
```

```
## [1] "1978" "2011"
```

```
str_match(x, pat)
```

```
##      [,1]   [,2] [,3]
## [1,] "1978" "19" "78"
## [2,] "2011" "20" "11"
```

9. 修改字符串

用新字符串赋值给 str_sub() 提取的子字符串；

做字符替换，基本格式为：

```
str_replace(string, pattern, replacement)
```

- pattern：要替换的子字符串或模式；
- replacement：要替换为的新字符串。

```
x
```

```
## [1] "1978-2000"      "2011-2020-2099"
```

```
str_replace(x, "-", "/")
```

```
## [1] "1978/2000"      "2011/2020-2099"
```

10. 其它函数

- 转化大小写
  - str_to_upper()：转换为大写；
  - str_to_lower()：转换为小写；

- `str_to_title()`: 转换标题格式（单词首字母大写）

```
str_to_lower("I love r language.")
```

```
## [1] "i love r language."
```

```
str_to_upper("I love r language.")
```

```
## [1] "I LOVE R LANGUAGE."
```

```
str_to_title("I love r language.")
```

```
## [1] "I Love R Language."
```

- `str_conv(string, encoding)`: 转化字符串的字符编码
- `str_view(string, pattern, match)`: 在 Viewer 窗口输出（正则表达式）模式匹配结果
- `word(string, start, end, sep = " ")`: 从英文句子中提取单词
- `str_wrap(string, width = 80, indent = 0, exdent = 0)`: 调整段落格式

关于 stringr 包的注

以上查找匹配的各个函数，只是查找第一个匹配，要想查找所有匹配，各个函数都有另一版本：加后缀 `_all`，例如 `str_extract_all()`
以上各个函数中的参数 `pattern` 都支持用 **正则表达式**（Regular Expression）表示模式。
正则表达式，是根据字符串规律按一定法则，简洁表达一组字符串的表达式。正则表达式是表示一组字符串的特征（或模式），正则表达式是处理文本数据非常强大的工具，将在下一节专门来讲解。

1.4.2 日期时间

日期时间值通常以字符串形式传入 R 中，然后转化为以数值形式存储的日期时间变量。

R 的内部日期是以 1970 年 1 月 1 日至今的天数来存储，内部时间则是以 1970 年 1 月 1 日至今的秒数来存储。

tidyverse 系列的 `lubridate` 包提供了更加方便的函数，生成、转换、管理日期时间数据，足以代替 R 自带的日期时间函数。

```
library(lubridate)
```

1. 识别日期时间

```
today()
```

```
## [1] "2020-09-08"
```

```
now()
```

```
## [1] "2020-09-08 20:54:40 CST"
```

```
as_datetime(today())   # 日期型转日期时间型
```

```
## [1] "2020-09-08 UTC"
```

```
as_date(now())         # 日期时间型转日期型
```

```
## [1] "2020-09-08"
```

无论年月日/时分秒按什么顺序及以什么间隔符分隔，总能正确地识别成日期时间值：

```
ymd("2020/03~01")
```

```
## [1] "2020-03-01"
```

```
myd("03202001")
```

```
## [1] "2020-03-01"
```

```
dmy("03012020")
```

```
## [1] "2020-01-03"
```

```
ymd_hm("2020/03~011213")
```

```
## [1] "2020-03-01 12:13:00 UTC"
```

注：根据需要可以 ymd_h/myd_hm/dmy_hms 任意组合；可以用参数 tz ="…" 指定时区。

也可以用 make_date() 和 make_datetime() 从日期时间组件创建日期时间：

```
make_date(2020, 8, 27)
```

```
## [1] "2020-08-27"
```

```
make_datetime(2020, 8, 27, 21, 27, 15)
```

```
## [1] "2020-08-27 21:27:15 UTC"
```

2. 格式化输出日期时间

用 `format()` 函数

```
d = make_date(2020, 3, 5)
format(d, '%Y/%m/%d')
```

```
## [1] "2020/03/05"
```

用 `stamp()` 函数，按给定模板格式输出

```
t = make_datetime(2020, 3, 5, 21, 7, 15)
fmt = stamp("Created on Sunday, Jan 1, 1999 3:34 pm")
fmt(t)
```

```
## [1] "Created on Sunday, 03 05, 2020 21:07 下午"
```

3. 提取日期时间数据的组件

日期时间数据中的“年、月、日、周、时、分、秒”等，称为其组件。

日期时间格式	含义	示例
%d	数字表示的日期（00~31）	01~31
%a	缩写的星期名	Mon
%A	非缩写的星期名	Monday
%w	数字表示的星期几	0~6（0 为周日）
%m	数字表示的月份（00~12）	00~12
%b	缩写的月份	Jan
%B	非缩写的月份	January
%y	二位数的年份	16
%Y	四位数的年份	2016
%H	24 小时制小时（00~23）	00~23
%I	12 小时制小时（01~12）	01~12
%p	AM/PM 指示	AM/PM
%M	十进制分钟（00~60）	00~60
%S	十进制秒（00~60）	00~60

图 1.8: R 中的日期时间组件

```
t = ymd_hms("2020/08/27 21:30:27")
t
```

```
## [1] "2020-08-27 21:30:27 UTC"
```

```
year(t)
```

```
## [1] 2020
```

```
quarter(t)              # 第几季度
```

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

```
month(t)
```

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

```
day(t)
```

```
## [1] 27
```

```
yday(t)                 # 当年的第几天
```

```
## [1] 240
```

```
hour(t)
```

```
## [1] 21
```

```
minute(t)
```

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

```
second(t)
```

```
## [1] 27
```

```
weekdays(t)
```

```
## [1] "星期四"
```

```
wday(t)                 # 数值表示本周第几天，默认周日是第 1 天
```

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

```
wday(t,label = TRUE)    # 字符因子型表示本周第几天
```

```
## [1] 周四
## Levels: 周日 < 周一 < 周二 < 周三 < 周四 < 周五 < 周六
```

```
week(t)                 # 当年第几周
```

```
## [1] 35
```

```
tz(t)                   # 时区
```

```
## [1] "UTC"
```

用 `with_tz()` 将时间数据转换为另一个时区的同一时间；`force_tz()` 将时间数据的时区强制转换为另一个时区：

```
with_tz(t, tz = "America/New_York")
```

```
## [1] "2020-08-27 17:30:27 EDT"
```

```
force_tz(t, tz = "America/New_York")
```

```
## [1] "2020-08-27 21:30:27 EDT"
```

还可以模糊提取（取整）到不同时间单位：

```
round_date(t, unit="hour")      # 四舍五入取整到小时
```

```
## [1] "2020-08-27 22:00:00 UTC"
```

注：类似地，向下取整: `floor_date()`; 向上取整: `ceiling_date()`

`rollback(dates, roll_to_first=FALSE, preserve_hms=TRUE)`：回滚到上月最后一天或本月第一天

4. 时间段数据

- `interval()`: 计算两个时间点的时间间隔，返回时间段数据

```
begin = ymd_hm("2019-08-10 14:00")
end = ymd_hm("2020-03-05 18:15")
gap = interval(begin, end)
gap
```

```
## [1] 2019-08-10 14:00:00 UTC--2020-03-05 18:15:00 UTC
```

```
time_length(gap, "day")     # 计算时间段的长度为多少天
```

```
## [1] 208.1771
```

```
time_length(gap, "minute")  # 计算时间段的长度为多少分钟
```

```
## [1] 299775
```

```
t %within% gap              # 判断 t 是否属于该时间段
```

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

- `duration()`: 以数值 + 时间单位存储时段的长度

```
duration(100, units = "day")
```

```
## [1] "8640000s (~14.29 weeks)"
```

```
int = as.duration(gap)
int
```

```
## [1] "17986500s (~29.74 weeks)"
```

- period(): 基本同 duration()

二者区别：duration 是基于数值线，不考虑闰年和闰秒；period 是基于时间线，考虑闰年和闰秒。

比如，duration 中的 1 年总是 365 天，而 period 的平年 365 天闰年 366 天。

- 固定单位的时间段

period 时间段：years(), months(), weeks(), days(), hours(), minutes(), seconds();

duration 时间段：dyears(), dmonths(), dweeks(), ddays(), dhours(), dminutes(), dseconds().

```
dyears(1)
```

```
## [1] "31557600s (~1 years)"
```

```
years(1)
```

```
## [1] "1y 0m 0d 0H 0M 0S"
```

5. 日期的时间的计算

时间点 + 时间段生成一个新的时间点：

```
t + int
```

```
## [1] "2021-03-24 01:45:27 UTC"
```

```
leap_year(2020)              # 判断是否闰年
```

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

```
ymd(20190305) + years(1)     # 加 period 的一年
```

```
## [1] "2020-03-05"
```

```
ymd(20190305) + dyears(1)    # 加 duration 的一年, 365 天
```

```
## [1] "2020-03-04 06:00:00 UTC"
```

```
t + weeks(1:3)
```

```
## [1] "2020-09-03 21:30:27 UTC" "2020-09-10 21:30:27 UTC"
## [3] "2020-09-17 21:30:27 UTC"
```

除法运算：

```
gap / ddays(1)              # 除法运算，同 time_length(gap,'day')
```

```
## [1] 208.1771
```

```
gap %/% ddays(1)            # 整除
```

```
## [1] 208
```

```
gap %% ddays(1)             # 余数
```

```
## [1] 2020-03-05 14:00:00 UTC--2020-03-05 18:15:00 UTC
```

```
as.period(gap %% ddays(1))
```

```
## [1] "4H 15M 0S"
```

月份加运算: %m+%，表示日期按月数增加。例如，生成每月同一天的日期数据：

```
date = as_date("2019-01-01")
date %m+% months(0:11)
```

```
##  [1] "2019-01-01" "2019-02-01" "2019-03-01" "2019-04-01" "2019-05-01"
##  [6] "2019-06-01" "2019-07-01" "2019-08-01" "2019-09-01" "2019-10-01"
## [11] "2019-11-01" "2019-12-01"
```

prety_dates() 生成近似的时间刻度：

```
x = seq.Date(as_date("2019-08-02"), by = "year", length.out = 2)
pretty_dates(x, 12)
```

```
##  [1] "2019-08-01 UTC" "2019-09-01 UTC" "2019-10-01 UTC"
##  [4] "2019-11-01 UTC" "2019-12-01 UTC" "2020-01-01 UTC"
##  [7] "2020-02-01 UTC" "2020-03-01 UTC" "2020-04-01 UTC"
## [10] "2020-05-01 UTC" "2020-06-01 UTC" "2020-07-01 UTC"
## [13] "2020-08-01 UTC" "2020-09-01 UTC"
```

1.4.3 时间序列

为了研究某一事件的规律，依据时间发生的顺序将事件在多个时刻的数值记录下来，就构成了一个时间序列，用 $\{Y_t\}$ 表示。

例如，国家或地区的年度财政收入，股票市场的每日波动，气象变化，工厂按小时观测的产量等等。另外，随温度、高度等变化而变化的离散序列，也可以看作时间序列。

ts 数据类型是专门为时间序列设计的，一个时间序列数据，其实就是一个数值型向量，且每个数都有一个时刻与之对应。

用 ts() 函数生成时间序列，基本格式为：

```
ts(data, start=1, end, frequency=1, ...)
```

- data：为数值向量或矩阵；
- start：设置起始时刻；
- end：设置结束时刻；
- frequency：设置时间频率，默认为 1，表示一年有 1 个数据。

```
ts(data = 1:10, start = 2010, end = 2019)      # 年度数据
```

```
## Time Series:
## Start = 2010
## End = 2019
## Frequency = 1
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
ts(data = 1:10, start = 2010, frequency = 4)  # 季度数据
```

```
##      Qtr1 Qtr2 Qtr3 Qtr4
## 2010    1    2    3    4
## 2011    5    6    7    8
## 2012    9   10
```

同理，月度数据则 frequency = 12，周度数则为 frequency = 52，日度数据则为 frequency = 365。

时间序列领域的最新包，可参阅：

- tidyverts: https://github.com/tidyverts
- business-science: https://github.com/business-science

本节部分内容参阅 (Hadley Wickham 2017), RStudio Cheatsheets: String manipulation with stringr, RStudio Cheatsheets: Dates and Times with lubridate.

1.5 正则表达式

正则表达式，是根据字符串规律按一定法则，简洁表达一组字符串的表达式。正则表达式通常就是从貌似无规律的字符串中发现规律性，进而概括性地表达它们所共有的规律或模式，以方便地操作处理它们，这是真正的**化繁为简，以简御繁**的典范。

几乎所有的高级编程语言都支持正则表达式，正则表达式广泛应用于文本挖掘、数据预处理，例如：

- 检查文本中是否含有指定的特征词
- 找出文本中匹配特征词的位置
- 从文本中提取信息
- 修改文本

正则表达式包括：只能匹配自身的普通字符（如英文字母、数字、标点等）和被转义了的特殊字符（称为‘‘元字符’’）。

1.5.1 基本语法

1. 常用的元字符

符号	描述
.	匹配除换行符“/n”以外的任意字符
\\	转义字符，匹配元字符时，使用“\\元字符”
|	表示或者，即 | 前后的表达式任选一个
^	匹配字符串的开始
$	匹配字符串的结束
( )	提取匹配的字符串，即括号内的看成一个整体，即指定子表达式
[ ]	可匹配方括号内任意一个字符
{}	前面的字符或表达式的重复次数：{n}表示重复 n 次；{n,}重复 n 次到更多次；{n, m}表示重复 n 次到 m 次
*	前面的字符或表达式重复 0 次或更多次
+	前面的字符或表达式重复 1 次或更多次
?	前面的字符或表达式重复 0 次或 1 次

图 1.9: 常用的元字符（表）

其它语言中的转义字符一般是 \\；在多行模式下，^ 和 $ 就表示行的开始和结束。

```
# 创建多行模式的正则表达式
pat = regex("^\\(.+?\\)$", multiline = TRUE)
```

2. 特殊字符类与反义

符号	描述
\\d 与 \\D	匹配 1 位数字字符，匹配非数字字符
\\s 与 \\S	匹配空白符，匹配非空白符
\\w 与 \\W	匹配字母或数字或下划线或汉字，匹配非\w 字符
\\b 与 \\B	匹配单词的开始或结束的位置，匹配非\b 的位置
\\h 与 \\H	匹配水平间隔，匹配非水平间隔
\\v 与 \\V	匹配垂直间隔，匹配非垂直间隔
[^...]	匹配除了...以外的任意字符

图 1.10: 特殊字符类与反义（表）

- \\S+: 匹配不包含空白符的字符串
- \\d: 匹配数字，同 [0-9]
- [a-zA-Z0-9]: 匹配字母和数字
- [\u4e00-\u9fa5] 匹配汉字
- [^aeiou]: 匹配除 aeiou 之外的任意字符，即匹配辅音字母

3. POSIX 字符类

符号	描述
[[:lower:]]	小写字母
[[:upper:]]	大写字母
[[:alpha:]]	大小写字母
[[:digit:]]	数字 0-9
[[:alnum:]]	字母和数字
[[:blank:]]	空白符：空格、制表符、换行符、中文全角空格等
[[:cntrl:]]	控制字符
[[:punct:]]	标点符号：！”＃％＆’（）＊＋－．／：；等
[[:space:]]	空格字符：空格，制表符，垂直制表符，回车，换行符，换页符
[[:xdigit:]]	十六进制数字：0-9 A-F a-f
[[:print:]]	控制字符：[[:alpha:]], [[:punct:]], [[:space:]]
[[:graph:]]	图形化字符：[[:alpha:]], [[:punct:]]

图 1.11: POSIX 字符类（表）

4. 运算优先级

圆括号括起来的表达式最优先，其次是表示重复次数的操作（即 * + { }）；再次是连接运算（即几个字符放在一起，如 abc）；最后是或者运算（|）。

另外，正则表达式还有若干高级用法，常用的有**零宽断言**和**分组捕获**，将在下面实例中进行演示。

1.5.2 若干实例

以上正则表达式语法组合起来使用，就能产生非常强大的匹配效果，对于匹配到的内容，根据需要可以提取它们，可以替换它们。

正则表达式与 stringr 包连用

若只是调试和查看正则表达式的匹配效果，可用 `str_view()` 及其 `_all` 后缀版本，将在 RStudio 的 Viewer 窗口显示匹配结果，在原字符向量中高亮显示匹配内容，非常直观。

若要提取正则表达式匹配到的内容，则用 `str_extract()` 及其 `_all` 后缀版本。

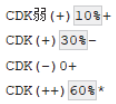
若要替换正则表达式匹配到的内容，则用 `str_replace()` 及其 `_all` 后缀版本。

使用正则表达式关键是，能够从貌似没有规律的字符串中发现规律性，再将规律性用正则表达式语法表示出来。下面看几个正则表达式比较实用的实例。

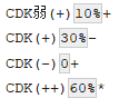
例 1.2 直接匹配

适合想要匹配的内容具有一定规律性，该规律性可用正则表达式表示出来。比如，数据中包含字母、符号、数值，我们想提取其中的数值，按正则表达式语法规则直接把要提取的部分表示出来：

```
x = c("CDK 弱 (+)10%+", "CDK(+)30%-", "CDK(-)0+", "CDK(++)60%*")
str_view(x, "\\d+%")
```




```
str_view(x, "\\d+%?")
```



\\d 表示匹配一位数字，+ 表示前面数字重复 1 次或多次，接着 % 原样匹配 %. 若后面不加 ? 则必须匹配到 % 才会成功，故第 3 个字符串就不能成功匹配；若后面加上 ? 则表示匹配前面的 % 0 次或 1 次，从而能成功匹配第 3 个字符串。

例 1.3 （零宽断言）匹配两个标志之间的内容

适合想要匹配的内容没有规律性，但该内容位于两个有规律性的标志之间，标志也可以是开始和结束。

通常想要匹配的内容不包含两边的 ‘‘标志’’，这就需要用**零宽断言**。简单来说，就是一种引导语法告诉既要匹配到 ‘‘标志’’，但又不包含 ‘‘标志’’。左边标志的引导语法是 (?<= 标志)，右边标志的引导语法是 (?= 标志)，而真正要匹配的内容放在它们中间。

比如，来自问卷星 ‘‘来自 IP’’ 数据，想要提取 IP、省份。

```
x = c("175.10.237.40(湖南-长沙)", "114.243.12.168(北京-北京)",
      "125.211.78.251(黑龙江-哈尔滨)")
# 提取省份
str_extract(x, "\\(.*-")            # 对比，不用零宽断言
```

```
## [1] "(湖南-"   "(北京-"   "(黑龙江-"
```

```
str_extract(x, "(?<=\\().*(?=-)")  # 用零宽断言
```

```
## [1] "湖南"   "北京"   "黑龙江"
```

```
# 提取 IP
# str_extract(x, "\\d.*\\d")        # 直接匹配
str_extract(x, "^.*(?=\\()")        # 用零宽断言
```

```
## [1] "175.10.237.40"  "114.243.12.168" "125.211.78.251"
```

省份位于两个标志 “(” 和 “-” 之间，但又不包含该标志，这就需要用到零宽断言。IP 位于两个标志 ‘‘开始’’ 和 “(” 之间，左边用开始符号 ^，右边用零宽断言。

再比如，用零宽断言提取专业（位于“级”和数字之间）：

```
x = c("18 级能源动力工程 2 班", "19 级统计学 1 班")
str_extract(x, "(?<= 级).*?(?=[0-9])")
```

```
## [1] "能源动力工程" "统计学"
```

关于懒惰匹配

正则表达式正常都是贪婪匹配，即重复直到文本中能匹配的最长范围，例如匹配小括号：

```
str_extract("(1st) other (2nd)", "\\(.+\\)")
```

```
## [1] "(1st) other (2nd)"
```

若想只匹配到第 1 个右小括号，则需要懒惰匹配，在重复匹配后面加上 ? 即可：

```
str_extract("(1st) other (2nd)", "\\(.+?\\)")
```

```
## [1] "(1st)"
```

例 1.4 分组捕获

正则表达式中可以用圆括号来分组，作用是

- 确定优先规则
- 组成一个整体
- 拆分出整个匹配中的部分内容（称为捕获）
- 捕获内容供后续引用或者替换。

比如，来自瓜子二手车的数据：若型号是中文，则品牌与型号中间有空格；若型号为英文或数字，则品牌与型号中间没有空格。

若用正则表达式匹配“字母或数字”并分组，然后捕获该分组并用添加空格替换：

```
x = c(" 宝马 X3 2016 款", " 大众 速腾 2017 款", " 宝马 3 系 2012 款")
str_replace(x, "([a-zA-Z0-9])", " \\1")
```

```
## [1] "宝马 X3 2016款"    "大众 速腾  2017款" "宝马 3系 2012款"
```

后续再用空格分割列即可。更多分组的引用还有 \\2, \\3, ...

最后，再推荐一个来自 Github 可以推断正则表达式的包 inferregex：

```
library(inferregex)
infer_regex("abcd-9999-ab9")$regex
```

```
## [1] "^[a-z]{4}-\\d{4}-[a-z]{2}\\d$"
```

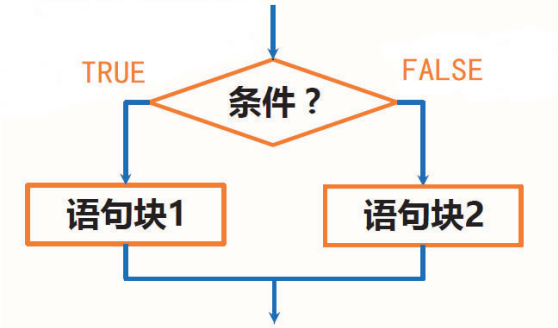
本节部分内容参阅 (Hadley Wickham 2017), (李东风2020), 嵩天, Python 网络爬虫与信息提取, deerchao 博客园, 正则表达式 30 分钟入门教程

1.6 控制结构

编程中的控制结构，是指分支结构和循环结构。

1.6.1 分支结构

正常程序结构与一步一步解决问题是一致的，即顺序结构，过程中可能需要对不同情形选择走不同的支路，即分支结构，是用条件语句做判断以实现分支：



R 语言中的条件语句的一般格式为：

1. 一个分支

```
if(条件) {
  执行体
}
```

2. 两个分支

```
if(条件) {
  执行体 1
} else {
  执行体 2
}
```

例如，实现计算 $|x|$:

```
if(x < 0) {
  y = -x
} else {
```

```
  y = x
}
```

3. 多个分支

```
if(条件 1) {
  执行体 1
} else if(条件 2) {
  执行体 2
} else {
  执行体 n
}
```

中间可以有任意多个 `else if` 块。多个分支的意思是，若满足“条件 1”，则执行“执行体 1”；其它的若满足“条件 2”，则执行“执行体 2”；……其它的，执行“执行体 n”。

> **特别注意**：分支的本意就是，不同分支之间不存在交叉（重叠）。

例 1.5 实现将百分制分数转化为五级制分数

```
if(score >= 90) {
  res = " 优"
} else if(score >= 80) {
  res = " 良"
} else if(score >= 70) {
  res = " 中"
} else if(score >= 60) {
  res = " 及格"
} else {
  res = " 不及格"
}
```

注意，若先写 `>=60`，结果就不对了。

关于“条件”

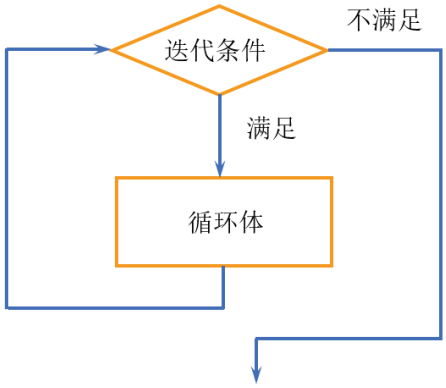
- “条件”是用逻辑表达式表示，必须是返回一个逻辑值 TRUE 或 FALSE；
- 多个逻辑表达式，可以通过逻辑运算符组合以表示复杂条件；
- 多个逻辑值的逻辑向量，可以借助函数 `any()` 和 `all()` 得到一个逻辑值；
- 函数 `ifelse()` 可简化代码，仍以计算 $|x|$ 为例：

```
ifelse(x < 0, -x, x)
```

1.6.2 循环结构

编程中减少代码重复的两个工具，一是循环，一是函数。

循环，用来处理对多个同类输入做相同事情（即迭代），如对向量的每个元素做相同操作，对数据框不同列做相同操作、对不同数据集做相同操作等。



R 语言有三种方式实现循环：

- for 循环、while 循环、repeat 循环
- apply 函数族
- purrr 泛函式编程

关于跳出循环：

- 用关键字 next 跳出本次循环，进入下次循环
- 用关键词 break 跳出循环

两点说明

关于“for 循环运行速度慢”的说法，实际上已经过时了，现在的 R、Matlab 等软件经过多年的内部优化已经不慢了，之所以表现出来慢，是因为你没有注意两个关键点：

- 提前为保存循环结果分配存储空间；
- 为循环体中涉及到的数据选择合适的数据结构。

apply 函数族和 purrr 泛函式编程能够更加高效简洁地实现一般的 for 循环、while 循环，但这不代表 for 循环、while 循环就没用了，它们可以在更高的层次使用（相对于在逐元素级别使用）。

1. for 循环

（1）基本 for 循环

```
library(tidyverse)
df = as_tibble(iris[,1:4])
```

用 “复制-粘贴” 法，计算前 4 列的均值：

```
mean(df[[1]])
```

```
## [1] 5.843333
```

```
mean(df[[2]])
```

```
## [1] 3.057333
```

```
mean(df[[3]])
```

```
## [1] 3.758
```

```
mean(df[[4]])
```

```
## [1] 1.199333
```

为了避免 “粘贴-复制多于两次”，改用 `for` 循环实现：

```
output = vector("double", 4)              # 1. 输出
for (i in 1:4) {                          # 2. 迭代器
  output[i] = mean(df[[i]])               # 3. 循环体
}
output
```

```
## [1] 5.843333 3.057333 3.758000 1.199333
```

for 循环有三个组件：

(i) **输出**：`output = vector("double", 4)`

在循环开始之前，最好为输出分配足够的存储空间，这样效率更高：若每循环一次，就用 `c()` 合并一次，效率会很低下。

通常是用 `vector()` 函数创建一个给定长度的空向量，它有两个参数：向量类型 (logical, integer, double, character 等)、向量长度。

(ii) **迭代器**：`i in 1:4`

确定怎么循环：每次 `for` 循环将对 `i` 赋一个 `1:4` 中的值，可将 `i` 理解为代词 `it`.

有时候会用 `1:length(df)`, 但更安全的做法是用 `seq_along(df)` ，它能保证即使不小心遇到长度为 0 的向量时，仍能正确工作。

(iii) **循环体**：`output[i] = mean(df[[i]])`

即执行具体操作的代码，它将重复执行，每次对不同的 `i` 值。

- 第 1 次迭代将执行：`output[1] = mean(df[[1]])`
- 第 2 次迭代将执行：`output[2] = mean(df[[2]])`
- ……

(2) for 循环的几种常用操作

(i) 循环模式

- 根据数值索引：`for(i in seq_along(xs))`，迭代中使用 `x[i]`.
- 根据元素值：`for(x in xs)`，迭代中使用 `x`.
- 根据名字：`for(nm in names(xs))`，迭代中使用 `x[nm]`.

若要创建命名的输出，可按如下方式命名结果向量：

```
results = vector("list", length(x))
names(results) = names(x)
```

用数值索引迭代是最常用的形式，因为名字和元素都可以根据索引提取：

```
for (i in seq_along(x)) {
    name <- names(x)[i]
    value <- x[i]
}
```

(ii) 将每次循环得到的结果合并为一个整体对象

这种情形，在 `for` 循环中经常遇到。此时要尽量避免 ‘‘每循环一次，就做一次拼接’’，这样效率很低。更好的做法是：先将结果保存为列表，等循环结束再将列表 `unlist()` 或 `purrr::flatten_dbl()` 成一个向量。

另外两种类似的情形是：

- 生成一个长字符串。不是用 `str_c()` 函数将上一次的迭代结果拼接到一起，而是将结果保存为字符向量，再用函数 `str_c(output, collapse= " ")` 合并为一个单独的字符串；
- 生成一个大的数据框。不是依次用 `rbind()` 函数合并每次迭代的结果，而是将结果保存为列表，再用 `dplyr::bind_rows(output)` 函数合并成一个单独的数据框。

所以，遇到上述模式时，要先转化为更复杂的结果对象，最后再做一步合并。

2. while 循环

适用于迭代次数未知。

`while` 循环更简单，因为它只包含两个组件：条件、循环体：

```
while (condition) {
  # 循环体
}
```

While 循环是比 for 循环更一般的循环，因为 for 循环总可以改写为 while 循环，但 while 循环不一定能改写为 for 循环：

```
for (i in seq_along(x)) {
  # 循环体
}
# 等价于
i <- 1
while (i <= length(x)) {
  # 循环体
  i <- i + 1
}
```

下面用 while 循环实现：抛一枚硬币直到连续出现 3 次“正面”，需要的次数：

```
flip = function() sample(c(" 正面", " 反面"), 1)

flips = 0
nheads = 0

while (nheads < 3) {
  if (flip() == " 正面") {
    nheads <- nheads + 1
  } else {
    nheads <- 0
  }
  flips <- flips + 1
}

flips
```

```
## [1] 11
```

while 循环并不常用，但在模拟时常用，特别是预先不知道迭代次数的情形。

3. repeat 循环

重复执行循环体，直到满足退出条件：

```
repeat{
  # 循环体
  if(退出条件) break
}
```

注意，repeat 循环至少会执行一次。

repeat 循环等价于：

```
while (TRUE) {
  # 循环体
  if(退出条件) break
}
```

例如，用如下泰勒公式近似计算 e:

$$e = 1 + \sum_{k=1}^{\infty} \frac{1}{k!}$$

```
s = 1.0
x = 1
k = 0

repeat{
  k = k + 1
  x = x / k
  s = s + x
  if(x < 1e-10) break
}

stringr::str_glue(" 迭代 {k} 次，得到 e = {s}")
```

```
## 迭代 14 次，得到 e = 2.71828182845823
```

4. apply 函数族

更建议弃用 apply 函数族，直接用 purrr::map 系列。

(1) apply()

apply() 函数是最常用的代替 for 循环的函数。可以对矩阵、数据框、多维数组，按行或列进行循环计算，即将行或列的元素逐个传递给函数 FUN 进行计算。其基本格式为：

```
apply(x, MARGIN, FUN, ...)
```

- x：为数据对象（矩阵、多维数组、数据框）；
- MARGIN：1 表示按行，2 表示按列；
- FUN：表示要作用的函数。

```
x = matrix(1:6, ncol = 3)
apply(x, 1, mean)           # 按行求均值
```

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

```
apply(x, 2, mean)           # 按列求均值
```

```
## [1] 1.5 3.5 5.5
```

```
apply(df, 2, mean)          # 对前文 df 计算各列的均值
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

(2) tapply()

该函数可以按照因子分组计算分组统计：

```
height = c(165, 170, 168, 172, 159)
sex = factor(c(" 男", " 女", " 男", " 男", " 女"))
tapply(height, sex, mean)
```

```
##       男       女
## 168.3333 164.5000
```

注意, height 与 sex 是等长的，对应元素分别为同一人的身高和性别，tapply() 函数分男女两组计算了身高平均值。

(3) lapply()

lapply() 函数是一个最基础循环操作函数，用来对 vector、list、data.frame 逐元、逐成分、逐列分别应用函数 FUN，并返回和 x 长度相同的 list 对象。其基本格式为：

```
lapply(x, FUN, ...)
```

- x：为数据对象（列表、数据框、向量）；
- FUN：表示要作用的函数。

```
lapply(df, mean)       # 对前文 df 计算各列的均值
```

```
# $Sepal.Length
# [1] 5.843333
#
# $Sepal.Width
# [1] 3.057333
#
# $Petal.Length
# [1] 3.758
#
# $Petal.Width
# [1] 1.199333
```

（4）sapply() {.unnumbered}

sapply() 函数是 lapply() 的简化版本，多了一个参数 simplify，若 simplify=FALSE，则同 lapply()，若为 TRUE，则将输出的 list 简化为向量或矩阵。其基本格式为：

```
sapply(x, FUN, simplify = TRUE, ...)
```

```
sapply(df, mean)       # 对前文 df 计算各列的均值
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

5. purrr 泛函式编程

相对于 apply 族，purrr 泛函式编程提供了更多的一致性、规范性和便利性，更容易记住和使用。

（1）先来理解几个概念

循环迭代

就是将函数依次应用（映射）到序列的每一个元素上，做相同的操作

序列，是由一系列可以根据位置索引的元素构成，元素可以很复杂和不同类型。原子向量和列表都是序列。

泛函式编程

函数的函数称为泛函，在编程中表示函数作用在函数上，或者说函数包含其它函数作为参数。

循环迭代，本质上就是将一个函数依次应用（映射）到序列的每一个元素上。表示出来不就是泛函式：`map(x, f)`。

`purrr` 泛函式编程解决循环迭代问题的逻辑是：

针对序列每个单独的元素，怎么处理它得到正确的结果，将之定义为函数，再 `map` 到序列中的每一个元素。

循环迭代返回类型的控制

`map` 系列函数都有后缀形式，以决定循环迭代之后返回的数据类型，这是 `purrr` 比 `apply` 函数族更先进和便利的一大优势。常用后缀如下：

- `map_chr(.x, .f)`: 返回字符型向量
- `map_lgl(.x, .f)`: 返回逻辑型向量
- `map_dbl(.x, .f)`: 返回实数型向量
- `map_int(.x, .f)`: 返回整数型向量
- `map_dfr(.x, .f)`: 返回数据框列表，再 `bind_rows` 按行合并为一个数据框
- `map_dfc(.x, .f)`: 返回数据框列表，再 `bind_cols` 按列合并为一个数据框

purrr 风格公式

在序列上做循环迭代（应用函数），经常需要自定义函数，但有些简单的函数也用 `function` 定义一番，毕竟是麻烦和啰嗦。所以，`purrr` 包提供了对 `purrr` 风格公式（匿名函数）的支持。

熟悉其它语言的匿名函数的话，很自然地就能习惯。

前面说了，`purrr` 包实现迭代循环是用 `map(x, f)`，`f` 是要应用的函数，想用匿名函数来写它，它要应用在序列 `x` 上，就是要和序列 `x` 相关联，那么就限定用序列参数名关联好了，即将该序列参数名作为匿名函数的参数使用：

- 一元函数：序列参数是 `.x` 比如，`f(x) = x^2 + 1`, 其 `purrr` 风格公式就写为：`~ .x ^ 2 + 1`
- 二元函数：序列参数是 `.x`, `.y` 比如，`f(x, y) = x^2 - 3 y`, 其 `purrr` 风格公式就写为：`~ .x ^ 2 - 3 * .y`

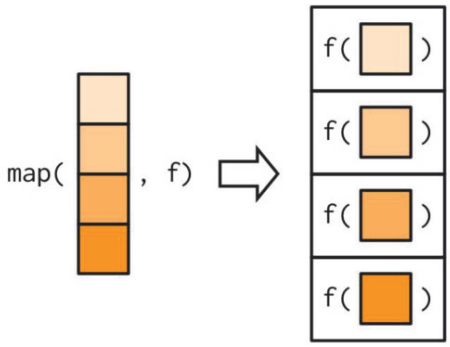
- 多元函数：序列参数是 `..1, ..2, ..3` 等比如，`f(x, y, z) = ln(x + y + z)`，其 purrr 风格公式就写为：`~ log(..1 + ..2 + ..3)`

> 所有序列参数，可以用 `...` 代替，比如，`sum(...)` 同 `sum(..1, ..2, ..3)`

（2）map()：依次应用一元函数到一个序列的每个元素

```
map(.x, .f, ...)
map_*(.x, .f, ...)
```

`.x` 为序列
`.f` 为要应用的一元函数，或 purrr 风格公式（匿名函数）
`...` 可设置函数.f 的其它参数



`map()` 返回结果列表，基本同 `lapply()`。例如，计算前文 `df`，每列的均值，即依次将 `mean()` 函数，应用到第 1 列，第 2 列，...；并控制返回结果为 `double` 向量：

```
map(df, mean)
```

```
## $Sepal.Length
## [1] 5.843333
##
## $Sepal.Width
## [1] 3.057333
##
## $Petal.Length
## [1] 3.758
##
## $Petal.Width
## [1] 1.199333
```

说明：`df` 是数据框 (特殊的列表)，作为序列其元素依次是：`df[[1]], df[[2]], .....` 所以，`map(df, mean)` 相当于依次计算：`mean(df[[1]]), mean(df[[2]]), .....`

返回结果是 `double` 型数值，所以更好的做法是，控制返回类型为数值向量，只

需：

```
map_dbl(df, mean)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##     5.843333     3.057333     3.758000     1.199333
```

另外，mean() 函数还有其它参数，如 na.rm，若上述计算过程需要设置忽略缺失值，只需：

```
map_dbl(df, mean, na.rm = TRUE)        # 数据不含 NA，故结果同上（略）
map_dbl(df, ~mean(.x, na.rm = TRUE))   # purrr 风格公式写法
```

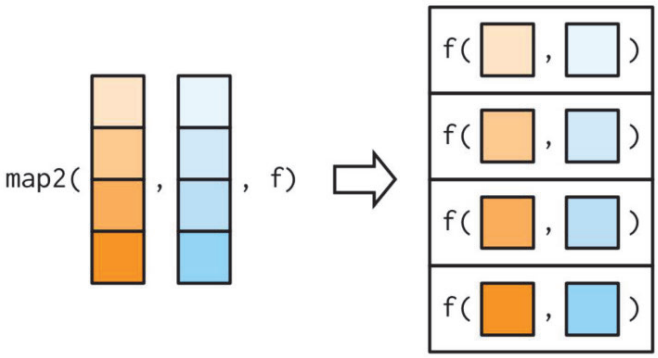
有了 map() 函数，对于自定义只接受标量的一元函数，比如 f(x)，想要让它支持接受向量作为输入，根本不需要改造原函数，只需：

```
map_*(xs, f)                           # xs 表示若干个 x 构成的序列
```

(3) map2()：依次应用二元函数到两个序列的每对元素

```
map2(.x, .y .f, ...)
map2_*(.x, .y, .f, ...)
```

.x 为序列 1
.y 为序列 2
.f 为要应用的二元函数，或 purrr 风格公式（匿名函数）
... 可设置函数 .f 的其它参数



例如，根据身高、体重数据计算 BMI 指数

```
height = c(1.58, 1.76, 1.64)
weight = c(52, 73, 68)

cal_BMI = function(h, w) w / h ^ 2      # 定义计算 BMI 的函数
map2_dbl(height, weight, cal_BMI)
```

```
## [1] 20.83000 23.56663 25.28257
```

说明：序列 1 其元素为：`height[[1]], height[[2]], ......`

序列 2 其元素为：`weight[[1]], weight[[2]], ......`

所以，`map2_dbl(height, weight, cal_BMI)` 相当于依次计算：

`cal_BMI(height[[1]], weight[[1]]), cal_BMI(height[[2]], weight[[2]]), ......`

更简洁的 purrr 风格公式写法（省了自定义函数）：

```
map2_dbl(height, weight, ~ .y / .x^2)
```

同样，有了 `map2()` 函数，对于自定义只接受标量的二元函数，比如 `f(x, y)`，想要让它支持接受向量作为输入，根本不需要改造原函数，只需：

```
map2_*(xs, ys, f)              # xs, ys 分别表示若干个 x, y 构成的序列
```

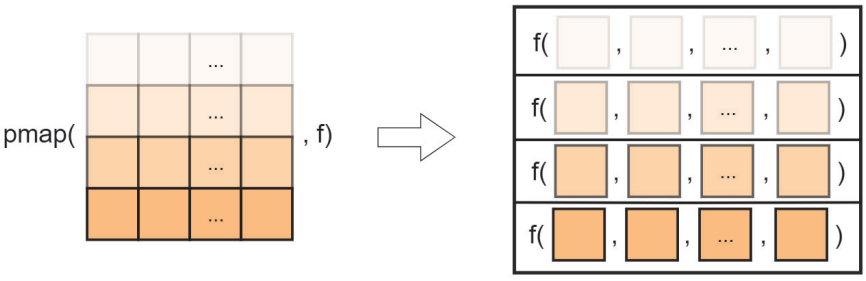
(4) pmap(): 应用多元函数到多个序列的每组元素，可以实现对数据框逐行迭代

多个序列得长度相同，长度相同的列表，不就是数据框吗。所以，`pmap()` 的多元迭代就是依次在数据框的每一行上进行迭代！

```
pmap(.l, .f, ...)
pmap_*(.l, .f, ...)
```

- `.l` 为数据框，
- `.f` 为要应用的多元函数
- `...` 可设置函数.f 的其它参数

注：`.f` 是几元函数，对应数据框 `.l` 有几列，`.f` 将依次在数据框 `.l` 的每一行上进行迭代。



例如，分别生成不同数量不同均值、标准差的正态分布随机数。

```
df = tibble(
  n = c(1, 3, 5),
  mean = c(5, 10, -3),
```

```
  sd = c(1, 5, 10)
)
df
```

```
## # A tibble: 3 x 3
##       n  mean    sd
##   <dbl> <dbl> <dbl>
## 1     1     5     1
## 2     3    10     5
## 3     5    -3    10
```

```
set.seed(123)
pmap(df, rnorm)
```

```
## [[1]]
## [1] 4.439524
##
## [[2]]
## [1]  8.849113 17.793542 10.352542
##
## [[3]]
## [1]  -1.707123  14.150650   1.609162 -15.650612  -9.868529
```

说明：这里的 `rnorm(n, mean, sd)` 是三元函数，`pmap(df, rnorm)` 相当于将三元函数 `rnorm()` 依次应用到数据框 `df` 的每一行上，即依次执行：

```
rnorm(1, 5, 1), rnorm(3, 10, 5), rnorm(5, -3, 10)
```

特别注意，这里 `df` 中的列名，必须与 `rnorm()` 函数的参数名相同（列序随便）。若要避免这种局限，可以使用 `purrr` 风格公式写法：

```
pmap(df, ~ rnorm(..1, ..2, ..3))    # 或者简写为
pmap(df, ~ rnorm(...))
```

`pmap_*()` 提供了一种行化操作数据框的办法。

```
pmap_dbl(df, ~ mean(c(...)))        # 按行求均值
```

```
## [1] 2.333333 6.000000 4.000000
```

```
pmap_chr(df, str_c, sep = "-")      # 将各行拼接在一起
```

```
## [1] "1-5-1"   "3-10-5"  "5--3-10"
```

其它 purrr 函数

- `imap_*(.x, .f)`: 带索引的 `map_*()` 系列，迭代的时候既迭代元素，也迭代元素的索引（位置或名字），purrr 风格公式中用.y 表示索引；
- `invoke_map_*(.f, .x, ...)`: 将多个函数依次应用到序列，相当于依次执行：`.f[[1]](.x, ...)`, `.f[[2]](.x, ...)`, ……
- `walk` 系列：`walk(.l, .f, ...)`, `walk2(.l, .f, ...)`, `pwalk(.l, .f, ...)`
  - 将函数依次作用到序列上，不返回结果。有些批量操作是没有或不关心返回结果的，例如批量保存到文件比如批量保存到文件等。
- `reduce()`: 可先对序列前两个元素应用函数，再对结果与第 3 个元素应用函数，再对结果与第 4 个元素应用函数，……直到所有的元都被“reduced”
  - `reduce(1:100, sum)` 是对 1:100 求累加和；
  - `reduce()` 可用于批量数据连接
- `accumulate()`: 与 `reduce()` 作用方式相同，不同之处是：`reduce()` 只返回最终的结果，而 `accumulate()` 会返回所有中间结果。

本节部分内容参阅 (Hadley Wickham 2017), Hadley Wickham. The joy of functional programming, 2019, Charlotte Wickham. Solving Iteration Problems With Purrr, 2017, Hendrik van Broekhuizen. purrr beyond map, functional programming in R, 2020, RStudio Cheatsheets: Apply Functions with purrr.

1.7 自定义函数

编程中的函数，是用来实现某个功能，其一般形式为：

```
(返回值 1, ..., 返回值 m) = 函数名 (输入 1, ..., 输入 n)
```

你只要把输入给它，它就能在内部进行相应处理，把你想要的返回值给你。

这些输入和返回值，在函数定义时，都要有固定的类型（模具）限制，叫做形参(形式上的参数)；在函数调用时，必须给它对应类型的具体数值，才能真正的去做处理，这叫做实参（实际的参数)。

所以，定义函数就好比创造一个模具，调用函数就好比用模具批量生成产品。

使用函数最大的好处，就是将实现某个功能，封装成模具，从而可以反复使用。这就避免了写大量重复的代码，程序的可读性也大大加强。

以前文的**百分制分数转化为五级制分数**为例，如果来一个百分制分数，就这样转化一次，10 个学生分数，就得写 100 多行代码。所以有必要封装成一个函数。

1.7.1 自定义函数

1. 自定义函数的一般语法

R 中，自定义函数的一般格式为：

```
函数名 = function(输入 1, ..., 输入 n) {
  函数体
  return(返回值)
}
```

> 注意，return 并不是必须的，默认函数体最后一行的值作为返回值，也就是说“return(返回值)”完全可以换成“返回值”。

2. 怎么自定义一个函数

我们想要自定义一个函数，能够实现**把百分制分数转化为五级制分数**的功能。

基于前面函数的理解，

第一步，分析输入和输出，设计函数外形

- 输入有几个，分别是什么，适合用什么数据类型存放；
- 输出有几个，分别是什么，适合用什么数据类型存放。

本问题，输入有 1 个：百分制分数，数值型；输出有 1 个：五级制分数，字符串

- 然后就可以设计自定义函数的外形：

```
Score_Conv = function(score) {
# 实现将一个百分制分数转化为五级分数
# 输入参数：score 为数值型，百分制分数
# 返回值：res 为字符串型，五级分数
...
}
```

> 函数名和变量可以随便起名，但是建议用有含义的单词。另外，为函数增加注释是一个好习惯。这些都是为了代码的可读性。

第二步，梳理功能的实现过程

前言中在谈如何自己写代码时讲到：“分解问题 + 实例梳理 + 翻译及调试”，完全适用于这里，不再赘述。

拿一组（本例只有一个）具体的形参的值作为输入，比如 76 分，分析怎么到达对应的五级分数“良”。这依赖于对五级分数界限的选取，选定之后做分支判断即可实现，即前文的条件语句中的示例那样。

复杂的功能，就需要更耐心的梳理和思考甚至借助一些算法，当然也离不开逐代码片段的调试。

```
score = 76
if(score >= 90) {
  res = " 优"
} else if(score >= 80) {
  res = " 良"
} else if(score >= 70) {
  res = " 中"
} else if(score >= 60) {
  res = " 及格"
} else {
  res = " 不及格"
}
res
```

```
## [1] "中"
```

拿一组具体的形参值作为输入，通过逐步调试，得到正确的返回值结果，这一步骤非常关键和有必要。

第三步，将第二步的代码封装到函数体

```
Score_Conv = function(score) {
  if(score >= 90) {
    res = " 优"
  } else if(score >= 80) {
    res = " 良"
  } else if(score >= 70) {
    res = " 中"
  } else if(score >= 60) {
    res = " 及格"
  } else {
    res = " 不及格"
  }
  res
}
```

基本就是原样作为函数体放入函数，原来的变量赋值语句不需要了，只需要形参。

3. 调用函数

要调用自定义函数，必须要先加载到当前变量窗口（内存），有两种方法：

- 需要选中并执行函数代码，或者
- 将函数保存为同名的 Score_Conv.R 文件，注意勾选 “Source on save” 再保存，然后执行 `source("Score_Conv.R", encoding="UTF-8")`

然后就可以调用函数了，给它一个实参 76，输出结果为 ‘‘中’’：

```
Score_Conv(76)
```

```
## [1] "中"
```

关于函数传递参数

要调用一个函数，比如 `f(x, y)`，首先要清楚其形参 `x, y` 所要求的类型，假设 `x` 要求是数值向量，`y` 要求是单个逻辑值。

那么，要调用该函数，首先需要准备与形参类型相符的实参（同名异名均可），比如

```
a = c(3.56, 2.1)
b = FALSE
```

再调用函数：

```
f(a, b)              # 同直接给值：f(c(3.56,2.1), FALSE)
```

调用函数时若不指定参数名，则默认是根据位置关联形参，即以 `x = a, y = b` 的方式进入函数体。

调用函数时若指定参数名，则根据参数名关联形参，位置不再重要，比如

```
f(y = b, x = a)      # 效果同上
```

4. 向量化改进

我们希望自定义函数，也能处理向量输入，即输入多个百分制分数，能一下都转化为五级分数。这也是所谓的 ‘‘向量化编程’’ 思维，就是要习惯用向量（矩阵）去思考、去表达。

方法一：修改自定义函数

将输入参数设计为数值向量，函数体也要相应的修改，借助循环依次处理向量中的每个元素，就相当于再套一层 `for` 循环。

```
Score_Conv2 = function(score) {
  n = length(score)
  res = vector("character", n)
```

```
  for(i in 1:n) {
    if(score[i] >= 90) {
      res[i] = " 优"
    } else if(score[i] >= 80) {
      res[i] = " 良"
    } else if(score[i] >= 70) {
      res[i] = " 中"
    } else if(score[i] >= 60) {
      res[i] = " 及格"
    } else {
      res[i] = " 不及格"
    }
  }
  res
}

# 测试函数
scores = c(35, 67, 100)
Score_Conv2(scores)
```

```
## [1] "不及格" "及格"   "优"
```

方法二：借助 apply 族或 map 系列函数

简单的循环语句，基本都可以改用 apply 族或 map 系列函数实现，其作用相当于是依次 ‘‘应用’’ 某函数，到序列的每个元素上。

也就是说，不需要修改原函数，直接就能实现向量化操作：

```
scores = c(35, 67, 100)
map_chr(scores, Score_Conv)
```

```
## [1] "不及格" "及格"   "优"
```

5. 处理多个返回值

若自定义函数需要有多个返回值，R 的处理方法是，将多个返回值放入一个列表(或数据框)，再返回一个列表。

例如，自定义函数，实现计算一个数值向量的均值和标准差：

```
MeanStd = function(x)
{
```

```
  mu = mean(x)
  std  = sqrt(sum((x-mu)^2) / (length(x)-1))
  list(mu=mu, std=std)
}

# 测试函数
x = c(2, 6, 4, 9, 12)
MeanStd(x)
```

```
## $mu
## [1] 6.6
##
## $std
## [1] 3.974921
```

6. 默认参数值

有时候需要为输入参数设置默认值。

以前面的计算数值向量的均值和标准差的函数为例。我们知道，标准差的计算公式有两种形式，一种是除以 n, 一种是除以 $n-1$.

此时，没有必要写两个版本的函数，只需要再增加一个指示参数，将用的多的版本设为默认即可。

```
MeanStd2 = function(x, type = 1) {
  mu = mean(x)
  if(type == 1) {
    std  = sqrt(sum((x - mu) ^ 2) / (length(x) - 1))
  } else {
    std  = sqrt(sum((x - mu) ^ 2) / length(x))
  }
  list(mu = mu, std = std)
}

# 测试函数
x = c(2, 6, 4, 9, 12)
# MeanStd2(x)                          # 同 MeanStd(x)
MeanStd2(x, 2)
```

```
## $mu
```

```
## [1] 6.6
##
## $std
## [1] 3.555278
```

7. ... 参数

一般函数参数只接受一个对象，即使不指定参数名，也会按位置对应参数。例如

```
my_sum = function(x, y) {
    sum(x, y)
}
my_sum(1, 2)
```

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

但是，如果想对 3 个数加和，怎么办？直接 `my_sum(1, 2, 3)` 会报错。

`...` 是一个特殊参数，可以接受任意多个对象，并作为一个列表传递它们：

```
dots_sum = function(...) {
    sum(...)
}
dots_sum(1)
```

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

```
dots_sum(1, 2, 3, 4, 5)
```

```
## [1] 15
```

几乎所有 R 自带函数都在用 `...` 这样传递参数。若参数 `...` 后面还有其它参数，为了避免歧义调用函数时需要对其随后参数命名。

1.7.2 R 自带函数

除了自定义函数，还可以使用现成的函数：

- 来自 R base：可直接使用
- 来自各种扩展包：需载入包，或加上包名前缀：包名`::`函数名`()`

这些函数的使用，可以通过 `?` 函数名查阅其帮助，以及查阅包页面的 Reference manual 和 Vignettes（若有）。

下面对常用的 R 自带函数做分类总结。

1. 基本数学函数

```
round(x, digits)      # 四舍五入，保留 n 位小数
signif(x, digits)     # 四舍五入，保留 n 位有效数字
ceiling(x)            # 向上取整，例如 ceiling(pi) 为 4
floor(x)              # 向下取整，例如 floor(pi) 为 3
sign(x)               # 符号函数
abs(x)                # 取绝对值
sqrt(x)               # 求平方根
exp(x)                # e 的 x 次幂
log(x, base)          # 对 x 取以... 为底的对数，默认以 e 为底
log2(x)               # 对 x 取以 2 为底的对数
log10(x)              # 对 x 取以 10 为底的对数
Re(z)                 # 返回复数 z 的实部
Im(z)                 # 返回复数 z 的虚部
Mod(z)                # 求复数 z 的模
Arg(z)                # 求复数 z 的辐角
Conj(z)               # 求复数 z 的共轭复数
```

2. 三角函数与双曲函数

```
sin(x)                # 正弦函数
cos(x)                # 余弦函数
tan(x)                # 正切函数
asin(x)               # 反正弦函数
acos(x)               # 反余弦函数
atan(x)               # 反正切函数
sinh(x)               # 双曲正弦函数
cosh(x)               # 双曲余弦函数
tanh(x)               # 双曲正切函数
asinh(x)              # 反双曲正弦函数
acosh(x)              # 反双曲余弦函数
atanh(x)              # 反双曲正切函数
```

3. 矩阵函数

```
nrow(A)               # 返回矩阵 A 的行数
ncol(A)               # 返回矩阵 A 的列数
```

```
dim(A)                  # 返回矩阵 x 的维数（几行 × 几列）
colSums(A)              # 对矩阵 A 的各列求和
rowSums(A)              # 对矩阵 A 的各行求和
colMeans(A)             # 对矩阵 A 的各列求均值
rowMeans(A)             # 对矩阵 A 的各行求均值
t(A)                    # 对矩阵 A 转置
det(A)                  # 计算方阵 A 的行列式
crossprod(A, B)         # 计算矩阵 A 与 B 的内积，t(A) %*% B
outer(A, B)             # 计算矩阵的外积（叉积），A %o% B
diag(x)                 # 取矩阵对角线元素，或根据向量生成对角矩阵
diag(n)                 # 生成 n 阶单位矩阵
solve(A)                # 求逆矩阵（要求矩阵可逆）
solve(A, B)             # 解线性方程组 AX=B
ginv(A)                 # 求矩阵 A 的广义逆（Moore-Penrose 逆）
eigen()                 # 返回矩阵的特征值与特征向量（列）
kronecker(A, B)         # 计算矩阵 A 与 B 的 Kronecker 积
svd(A)                  # 对矩阵 A 做奇异值分解，A=UDV'
qr(A)                   # 对矩阵 A 做 QR 分解：A=QR，Q 为酉矩阵，R 为阶梯形矩阵
chol(A)                 # 对正定矩阵 A 做 Choleski 分解，A=P'P，P 为上三角矩阵
A[upper.tri(A)]         # 提取矩阵 A 的上三角矩阵
A[lower.tri(A)]         # 提取矩阵 A 的下三角矩阵
```

4. 概率函数

```
factorial(n)            # 计算 n 的阶乘
choose(n, k)            # 计算组合数
gamma(x)                # Gamma 函数
beta(a, b)              # beta 函数
combn(x, m)             # 生成 x 中任取 m 个元的所有组合，x 为向量或整数 n
```

```
combn(4, 2)
```

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

```
combn(c(" 甲"," 乙"," 丙"," 丁"), 2)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,] "甲" "甲" "甲" "乙" "乙" "丙"
```

```
## [2,] "乙" "丙" "丁" "丙" "丁" "丁"
```

R 中，常用的概率函数有密度函数、分布函数、分位数函数、生成随机数函数，其写法为：

- d = 密度函数（density）
- p = 分布函数（distribution）
- q = 分位数函数（quantile）
- r = 生成随机数（random）

上述 4 个字母 + 分布缩写，就构成通常的概率函数，例如：

```
dnorm(3, 0, 2)                    # 正态分布 N(0, 4) 在 3 处的密度值
```

```
## [1] 0.0647588
```

```
pnorm(1:3, 1, 2)                  # N(1,4) 分布在 1,2,3 处的分布函数值
```

```
## [1] 0.5000000 0.6914625 0.8413447
```

```
# 命中率为 0.02，独立射击 400 次，至少击中两次的概率
1 - sum(pbinom(0:1, 400, 0.02))
```

```
## [1] 0.9968561
```

```
pnorm(2, 1, 2) - pnorm(0, 1, 2)   # X~N(1, 4)，求 P{0<X<=2}
```

```
## [1] 0.3829249
```

```
qnorm(1-0.025,0,1)                # N(0,1) 的上 0.025 分位数
```

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

生成随机数[7]：

```
set.seed(123)          # 设置随机种子，以重现随机结果
rnorm(5, 0, 1)         # 生成 5 个服从 N(0,1) 分布的随机数
```

```
## [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774
```

[7]自然界中的随机现象是真正随机发生不可重现的，计算机中模拟随机现象，包括生成随机数、随机抽样并不是真正的随机，而是可以重现的。通过设置为相同的起始种子值就可以重现，故称为“伪随机”。

表 常用分布的缩写

分布名称	缩写	参数及默认值
二项分布	binom	size, prob
多项分布	multinom	size, prob
负二项分布	nbinom	size, prob
几何分布	geom	prob
超几何分布	hyper	m, n, k
泊松分布	pois	lambda
均匀分布	unif	min=0, max=1
指数分布	exp	rate=1
正态分布	norm	mean=0, std=1
对数正态分布	lnorm	meanlog=0, stdlog=1
t 分布	t	df
卡方分布	chisq	df
F 分布	f	df1, df2
Wilcoxon 符号秩分布	signrank	n
Wilcoxon 秩和分布	wilcox	m, n
柯西分布	cauchy	location=0, scale=1
Logistic 分布	logis	location=0, scale=1
Weibull 分布	weibull	shape, scale=1
Gamma 分布	gamma	shape, scale=1
Beta 分布	beta	shape1, shape2

随机抽样：

sample() 函数，用来从向量中重复或非重复地随机抽样，基本格式为：

```
sample(x, size, replace = FALSE, prob)
```

- x：向量或整数；
- size：设置抽样次数；
- replace：设置是否重复抽样；
- prob：设定抽样权重。

```
set.seed(2020)
sample(c(" 正"," 反"), 10, replace=TRUE)  # 模拟抛 10 次硬币
```

```
##  [1] "反" "反" "正" "反" "反" "正" "正" "反" "反" "反"
```

```
sample(1:10, 10, replace=FALSE)          # 随机生成 1~10 的某排列
```

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

5. 统计函数

```
min(x)              # 求最小值
cummin(x)           # 求累计最小值
max(x)              # 求最大值
cummax(x)           # 求累计最大值
range(x)            # 求 x 的范围：[最小值, 最大值]（向量）
sum(x)              # 求和
cumsum(x)           # 求累计和
prod(x)             # 求积
cumprod(x)          # 求累计积
mean(x)             # 求平均值
median(x)           # 求中位数
quantile(x, pr)     # 求分位数，x 为数值向量，pr 为概率值
sd(x)               # 求标准差
var(x)              # 求方差
cov(x)              # 求协方差
cor(x)              # 求相关系数
scale(x, center=TRUE, scale=FALSE)  # 对数据做中心化：减去均值
scale(x, center=TRUE, scale=TRUE)   # 对数据做标准化
```

自定义极差标准化函数：

```
rescale = function(x, type=1) {
  # type=1 正向指标，type=2 负向指标
  rng = range(x, na.rm = TRUE)
  if (type == 1) {
    (x - rng[1]) / (rng[2] - rng[1])
  } else {
    (rng[2] - x) / (rng[2] - rng[1])
  }
}

x = c(1, 2, 3, NA, 5)
rescale(x)
```

```
## [1] 0.00 0.25 0.50   NA 1.00
```

```
rescale(x, 2)
```

```
## [1] 1.00 0.75 0.50   NA 0.00
```

6. 时间序列函数

`lag()` 函数，用来计算时间序列的滞后，基本格式为：

```
lag(x, k, ...)
```

- `x`：为数值向量/矩阵或一元/多元时间序列；
- `k`：为滞后阶数，默认为 1.

`diff()` 函数，用来计算时间序列的差分，基本格式为：

```
diff(x, lag = 1, difference = 1, ...)
```

- `x`：为数值向量/矩阵；
- `lag`：为滞后阶数，默认为 1；
- `difference`：为差分阶数，默认为 1.

Y_t 的 j 阶滞后为 Y_{t-j}：

```
x = ts(1:8, frequency = 4, start = 2015)
x
```

```
##      Qtr1 Qtr2 Qtr3 Qtr4
## 2015    1    2    3    4
## 2016    5    6    7    8
```

```
stats::lag(x, 4)       # 避免被 dplyr::lag() 覆盖
```

```
##      Qtr1 Qtr2 Qtr3 Qtr4
## 2014    1    2    3    4
## 2015    5    6    7    8
```

Y_t 的一阶差分为 $\Delta Y_t = Y_t - Y_{t-1}$，二阶差分为 $\Delta^2 Y_t = \Delta Y_t - \Delta Y_{t-1}, \ldots$

```
x = c(1, 3, 6, 8, 10)
x
```

```
## [1]  1  3  6  8 10
```

```
diff(x, differences = 1)
```

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

```
diff(x, differences = 2)
```

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

```
diff(x, lag = 2, differences = 1)
```

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

7. 其它函数

```
unique(x, ...)              # 返回唯一值，即去掉重复元素或观测
duplicated(x, ...)          # 判断元素或观测是否重复（多余），返回逻辑值向量
anyDuplicated(x, ...)       # 返回重复元素或观测的索引
rle(x)                      # 统计向量中连续相同值的长度
inverse.rle(x)              # rle() 的反向版本，x 为 list(lengths, values)
```

本节部分内容参阅 (Hadley Wickham 2017), (张良均，谢佳标，杨坦，肖刚2016).