Ｒ入門

Andrew Ba Tran

目次

Syntax..1
ワーキングディレクトリ..2
ライブラリ..3
R コマンドの例..4
電卓..4
ワークスペース..4
スカラーとベクトル..5
関数..5
プロット..6
スクリプト..7
利用できないデータ..7
データ型..8
文字..8
日付..8
因子..9
整数と数字..10
データ型の変換..10
関数再考..11
演習..12

learn.r-journalism.com 第１章より

Syntax

Ｒを機能させるには、厳格な構文規則に従わなければなりません。Ｒは忖度できません。

- Ｒは（ＳＱＬとは異なり）大文字と小文字を区別し、（Ｃと違って）インタプリタ型言語です。
- プロンプト`>`からコマンドを入力できます。

- コメントの前には `#`を入れてください。
- コメントは内容が他の人にも分かるように使いましょう。
- このコースでも使います。
- ステートメントは、関数やオブジェクトの割り当てなどを記述するコード行です。
- （改行と同じく）コードの途中でエンターやセミコロンを入力すると、文が区切られます。

ワーキングディレクトリ

*ワーキングディレクトリ*は、現在作業しているコンピュータ上のフォルダです。あるファイルを開くようＲに要求すると、そのファイルのワーキングディレクトリを調べます。データファイルや図を保存するようＲに指示すると、ワーキングディレクトリに保存します。

作業を開始する前には、すべてのデータとスクリプトファイルが保存されている場所をワーキングディレクトリとして設定してください。

Tip:

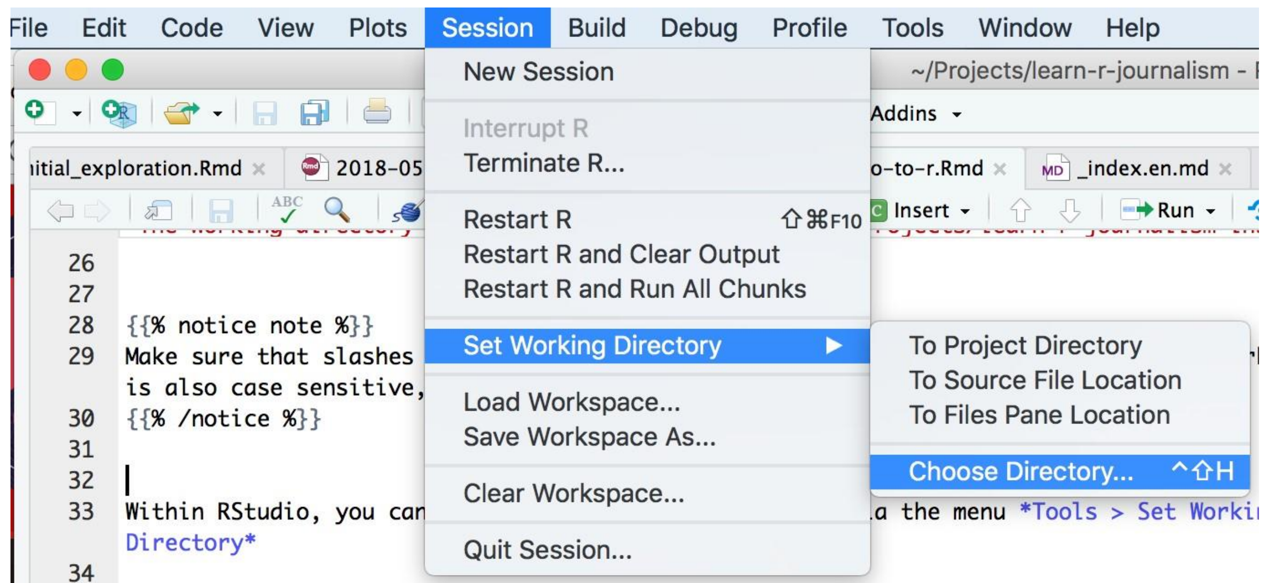
このクラスで、背景が黒い部分にコードがある場合は、**特に記載がない限りＲでそのコードを実行してください**。コンソールで実行することもできますが、履歴を追えるようにスクリプトを使うのが望ましいと思います。ビデオでは、コンソールとスクリプトの間を行き来してコーディングをします。これは、コードをちょっと試しているのか、それとも腰を据えて取り組むのかによって異なります。普通は、コンソールで実行したコードをスクリプトにコピー&ペーストすることになるでしょう。また、コマンドが失敗した場合の対処方法に関するヒントなど、追加の予備知識を提供しますので **コメントアウトされたコードは必ず読んでください。**

これは、ワーキングディレクトリを手動で設定する方法です。

```
# Mac なら setwd("~/projects/learn-r-journalism")

# その他の pc は setwd("C:/Documents/learn-r-journalism")
```

Note:スラッシュが普通のスラッシュであることを確かめ、そして引用符を忘れないようにしてください。RStudio では、ワーキングディレクトリをメニューから *Tools > Set Working Directory* という手順で設定できます。



上記のコマンド setwd()は絶対パスの設定例です。
問題ないように見えますが、自分のやり方とスクリプトを共有したいとか、別のコンピュータでコードを実行して保存したいと思っても、元のスクリプトが書かれたコンピュータの他には存在しないフォルダ構造を調べようとするので、**うまくいかない**でしょう。これは再現性の観点からすると理想的ではありません。

ワーキングディレクトリは難解な概念です。第 6 章で最善の手法を紹介しますが、差し当たりこのリンク(https://wangfanghelsinki.wordpress.com/2015/03/31/understanding-and-setting-rstudio-working-directory/)をご覧ください。

ライブラリ

R はさまざまなデータ統計分析を行うことができます。それらはいわゆるパッケージやライブラリとしてまとめられています。

R はライブラリを追加しなくても、多くの統計分析を行うことができます。これを base R と呼びます。

しかし、その他にも R の利用者は一般的な問題を解決するライブラリを作りました。R パッケージの利用者は、それぞれのプロジェクトに必要なライブラリだけをダウンロードします。

インストールされているすべてのパッケージのリストを取得するには、パッケージウィンドウに移動するか、コンソールに`library()` と入力します。パッケージウィンドウで、パッケージ名の欄にチェックが入っている場合、既にパッケージがロードされ、その中の関数を呼び出す準備ができています。

R の Web サイトには、より多くのパッケージがあります。パッケージをインストールして使いたい場合（たとえば "dplyr" なら）:

- パッケージをインストール: `install packages`をクリックし、パッケージウィンドウに `dplyr` と入力するか、コンソールに`install.packages("dplyr")`と打ち込んでください。
- パッケージのダウンロード: `dplyr`の前の欄にチェックを入れるか、コンソールに `library("dplyr")`と入力してください。

R コマンドの例

電卓

R は電卓として使えます。

`>`の後にコンソールウィンドウに式を入力するだけです。

Note: このセクションで、**##**が付いたコードは上の行のコードの出力を意味します。

```
10^2 + 26
```

```
## [1] 126
```

ワークスペース

数に名前を付けることもできます。

そうしておけば、後で使用できる、いわゆる変数になるのです。

```
a <- 4 a
```

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

`a`を計算に使うことができるようになりました。

```
a*5
```

```
## [1] 20
```

`a`を再び指定すると、割り当てが何もなかったため、以前に持っていた値は保持されていません。

```
a
```

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

前の値を使用して`a` に新しい値を代入することもできます。

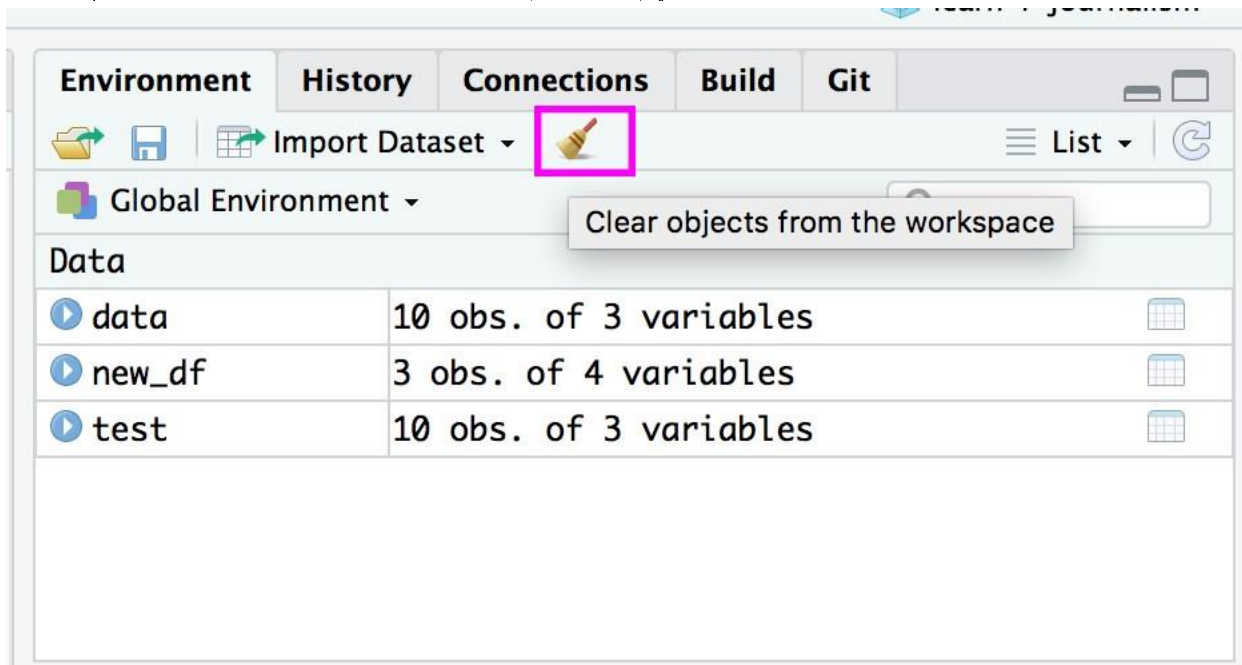
```
a <- a + 10 a
```

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

R のメモリからすべての変数を削除するには、次のように入力します。

```
rm(list=ls())
```

または、ワークスペースの "clear all" をクリックします。



スカラーとベクトル

多くのプログラムと同様に、R は数を*スカラー* (大きさのみを持つ 0 次元の量)、*ベクトル* (数列、配列とも呼びます) 、*行列* (今は深入りしません)に分類します。先ほど定義した`a` はスカラーです。

3、4、5 という数からなるベクトルを定義するには、連結（して貼り合わせる）の略である `c()` という関数が必要です。

```
b=c(3,4,5) b
```

[1] 3 4 5

関数

上記の例で挙げたベクトル `b` のすべての要素の平均を計算したい場合は、次のように入力します。

```
(3+4+5)/3
```

[1] 4

しかし、長いベクトルになると、これは非常に煩雑です。

関数はデータを処理するためのもので、R は関数で構築されています。`median()`や`summary()`のように、もともと R に付属している関数もあれば、作成したパッケージの一部として提供されている関数もあります。

平均を計算する関数は、次のようにタイプします。

```
mean(x=b)
```

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

（）には*引数*を入れます。
引数は関数に追加の情報を与えるものです。ここでは、引数*x*は、平均値を計算しようとしている数値の集合（ベクトル）を表します（すなわち b)。

引数の名前が不要な場合もあります。

```
mean(b)
```

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

これでも大丈夫です。

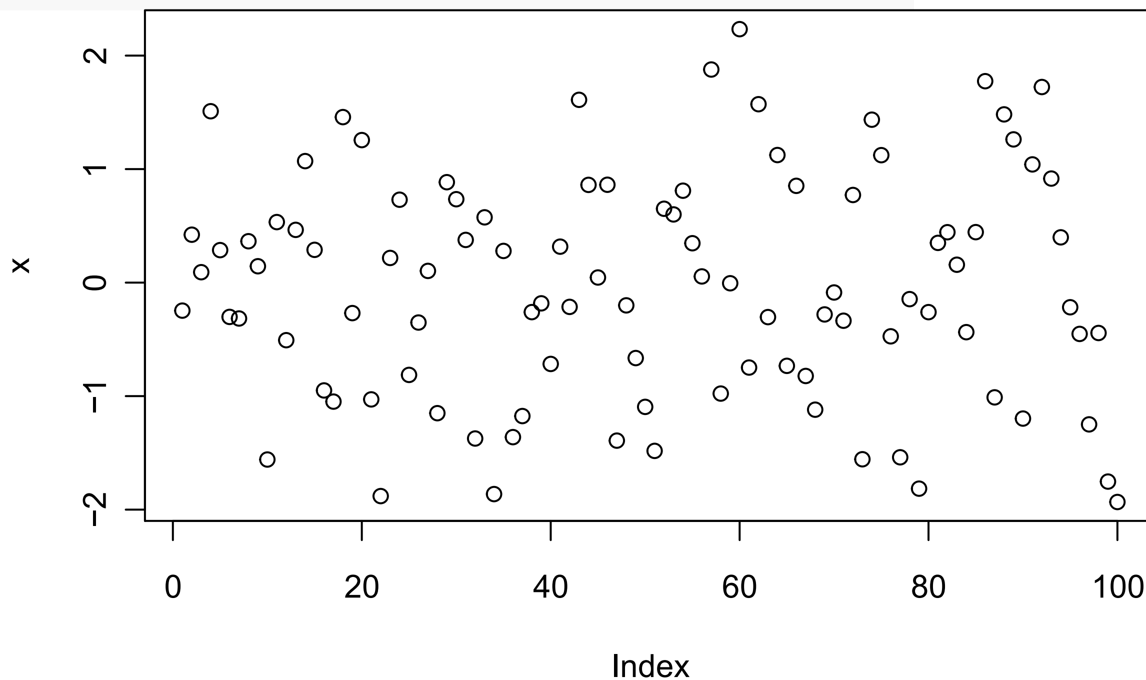
プロット

R は瞬時に簡単なグラフィックを作ることができます。

```
# rnorm() は、ランダムな分布からサンプルを生成する関数です。

distribution x <- rnorm(100)

# plot() は plot(x) をチャートにする関数です。
```



- 1 行目では、100 個の乱数がベクトルとして変数 x に割り当てられます。
- l 行目では、全ての値をプロットウィンドウに描画します。

スクリプト

R はコマンドラインベースの環境を使用するインタプリタです。つまり、マウスやメニューを使用するのではなく、コマンドを入力する必要があるのです。 これには、必ずしもコマンドを再入力する必要がないという利点があります。

コマンドはファイル、いわゆる*スクリプト*に保存することができます。これらのスクリプトは通常、**script.R** のように.**R** という拡張子が付いたファイル名を持ちます。

File > New または *File > Open file...*からこれらのファイルを編集することができます。
行を選択して **CTRL+ENTER** か **CMD+ENTER** 、スクリプトエディタウィンドウの上部にある *Run* をクリックすると、コードの一部を実行（コンソールウィンドウに送信）できます。何も選択しなければ、R はカーソルがある行を実行します。
いつでも`source()`を使えばスクリプト全体を実行できます。
例えば、保存済みの **script.R** が作業ディレクトリのルートディレクトリにあり、このスクリプト全体を実行する場合には

```
source("script.R")
```

エディタウィンドウで *Run all* を押すか、**CTRL+SHIFT+S** または *CMD+SHIFT+S* と入力しても構いません。

利用できないデータ

実際にデータを扱うとき、計測が失敗したか人的エラーが発生したために、しばしば値が欠落していることがあります。

データが*利用できない*場合には、数字の代わりに`NA`と表記する必要があります。

```
j <- c(1,2,NA)
```

厳密にいえば、不完全なデータセットの統計をとることは不可能です。

あなたが測定していなかった週末の間に最大の値を発生させる出来事があったかもしれません。したがって、R は `j` の最大値が何か分からないと返すでしょう。

```
max(j)
```

[1] NA

欠けているデータに構わず計算したい場合は、引数 `na.rm=TRUE` を追加します(要するに`NA`を除外しますか? Yes ということです)。

```
max(j, na.rm=T)
```

[1] 2

Warning: NA はあらゆる計算に影響します。

```
sum(j)
```

[1] NA

```
# compared to sum(j,
na.rm=T)
```

[1] 3

NA を処理するためのいくつかのリンクをご紹介します。

データ型

これまでは数値を使ってきました。しかし、私たちが扱うデータには、文字や **TRUE** または **FALSE** といったブール値、日付などの文字列のように、他のものもあるのです。.

文字

```
m <- "apples"

m
```

[1] "apples"

文字列であることを R に認識させるには、引用符の間にテキストを入力する必要があります。そうしないと、R は同じ名前の定義済み変数を探してしまいます。

```
n <- pears
```

Error in eval(expr, envir, enclos): object 'pears' not found

文字を使って計算することはできません。

```
m + 2
```

Error in m + 2: non-numeric argument to binary operator

日付

日付と時間は複雑な概念です。R は、3 時が 2 時 59 分の後に来ること、そして 2 月は数年に一度 29 日あることを知っていなければなりません。

日付と時刻の組み合わせであることを R に伝える基本的な方法は`strptime()`関数を使うことです。

```
date1 <- strptime(c("20100225230000", "20100226000000", "20100226010000"), format="%Y%m%d%H%M%S") date1
```

[1] "2010-02-25 23:00:00 EST" "2010-02-26 00:00:00 EST"
[3] "2010-02-26 01:00:00 EST"

関数`c()`によってベクトルが作成され、引用符の間の数字は文字列として扱われます。 これは`strptime()`関数を使うために必要な操作です。

文字列の読み方を規定する引数 **format** が後に続いています。この例では、年(%Y)、月(%M)、秒(%S)の順に並んでいます。フォーマットがストリングに対応していれば、全てを指定する必要はありません。

このコースでは **lubridate** パッケージを使用して手間をかけずに日付を処理する方法を使用します。

.

```
# lubridate パッケージをインストールしていない場合は、以下の行のコメントを外して実行してください。

# install.packages("lubridate") library(lubridate)
```

```
##
## Attaching package: 'lubridate'
## The following object is masked from 'package:base':
##
##     date
```

```
date1 <- ymd_hms(c("20100225230000", "20100226000000", "20100226010000"))
```

`ymd_hms()` は、ストリング内の年、月、日と時、分、秒を変換します。[第 3 章](http://learn.r-journalism.com/en/wrangling/dates/dates/)でより詳しく説明します。

因子

これは複雑な概念です。

* （性別などの）名義尺度や（順位などの）順序尺度を併せて質的データといい、このデータ構造を処理する方法を R に指示します。

* とても重要な概念にも関わらず、しばしば誤解されています。

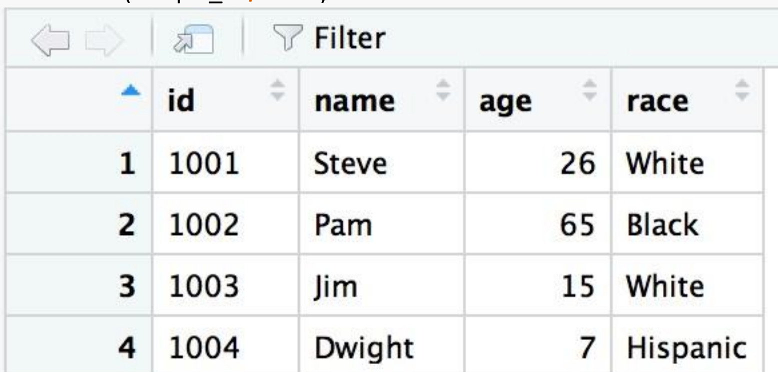
* 質的変数または順序尺度の変数は通常、因子として格納されます。

たとえば、人種を「白人」、「黒人」、「ヒスパニック」と入力するとしましょう。

スプレッドシートからそのデータをインポートするとき、R はふつうこれを**因子**として処理します。

以下のコードを実行して、新しいオブジェクト **sample_df** というデータフレームを作ります。

```
sample_df <- data.frame(id=c(1001,1002,1003,1004), name=c("Steve", "Pam", "Jim", "Dwight") age=c(26, 65, 15, 7), ))
                                                                                    race=c("White", "Black", "White", "Hispanic"
sample_df$race <- factor(sample_df$race) sample_df$id <-
factor(sample_df$id) sample_df$name <-
as.character(sample_df$name)
```



	id	name	age	race
1	1001	Steve	26	White
2	1002	Pam	65	Black
3	1003	Jim	15	White
4	1004	Dwight	7	Hispanic

作成したデータフレームの構造を見てみましょう。

```
str(sample_df)
```

```
## 'data.frame':	4 obs. of 4 variables:
## $ id : Factor w/ 4 levels "1001","1002",..: 1 2 3 4
## $ name: chr "Steve" "Pam" "Jim" "Dwight"
## $ age : num 26 65 15 7
## $ race: Factor w/ 3 levels "Black","Hispanic",..: 3 1 3 2
```

R は**人種を**三つのレベルがある因子型変数だと認識しています。

```
levels(sample_df$race)
```

```
## [1] "Black"	"Hispanic" "White"
```

これは R がこの統計を三つのレベルでグループ化することを示しています。

```
summary(sample_df$race)
```

```
##	Black Hispanic	White
##	1	1	2
```

R の内部では、文字列をアルファベット順に整数値 1、2、および 3 に割り当てます。1=黒人、2=ヒスパニック、3=白人ということになります。これを知ることがなぜ重要なのでしょうか。

ジャーナリストは因子を理解することにあまり関心がなく、しばしば因子を文字列や文字に置き換えています。しかし、線形回帰モデルを作成したいという段階にまで到達したら、因子を選択肢の一つとして使えることが有利になります。

Tip: このような R の最も理解しがたい点は、R が統計学者によって、統計学者のために作成されたことに起因するものです。R は大きく成長し、利用者のコミュニティはデータを慣れ親しんだ方法で扱うように進化させてきました。しかし、理解しがたい点のいくつかはしぶとく生き残っているのです。

整数と数字

計算に使用できるのは整数または 10 進数です。R は数字が混在するユニットを適切に扱うことができません。たとえば、「4in」は 4 ではなく文字列として扱われます。

データ型の変換

以下の点に注意してください。

- 因子を文字列に変換することができます。

```
sample_df$race
```

```
## [1] White	Black	White	Hispanic
## Levels: Black Hispanic White
```

```
as.character(sample_df$race)
```

```
## [1] "White"	"Black"	"White"	"Hispanic"
```

- 逆に文字列を因子に変換することもできます。

```
sample_df$name
```

```
## [1] "Steve" "Pam"	"Jim"	"Dwight"
```

```
factor(sample_df$name)
```

[1] Steve Pam Jim Dwight
Levels: Dwight Jim Pam Steve

- 因子を数値に変換することは**できません**。

```
sample_df$id
```

[1] 1001 1002 1003 1004 ## Levels:
1001 1002 1003 1004

```
as.numeric(sample_df$id)
```

[1] 1 2 3 4

R は**因子**を**整数**として格納するからです。数値に変換するにはまず文字に変換しなければなりません。

そうすれば、ネストが可能になります。

```
sample_df$id
```

[1] 1001 1002 1003 1004
Levels: 1001 1002 1003 1004

```
as.numeric(as.character(sample_df$id))
```

[1] 1001 1002 1003 1004

Tip: 次のセクションは完全に理解しなくても大丈夫です。だいぶ高度な内容です。**演習**まで飛ばしてもかまいません。

関数再考

プロセスを単純化するために、コードを関数に保存することもできます。

```
percent_change <- function(first_number, second_number) { pc <-
  (second_number-first_number)/first_number*100
  return(pc)
} percent_change(100,150)
```

[1] 50

上記のコードを実行すると、こんなことが起きます:

* **percent_change** は関数の名前で、`function()`関数がこれに割り当てられています。

* **first_number** と **second_number** という二つの変数が必要です。

* 新たなオブジェクト`pc`は、渡された 2 つの変数からパーセント変化を計算して作成されます。

* `return()`は`percent_change` に計算結果を返します。こうしてできた関数は[パッケージ](https://github.com/andrewbtran/muckrakr)として保存できます。

関数について大事なことはそれらが人間によってプログラムされているということです。

この関数を作成できたのは、入力が二つしかなく、それぞれが数値で、順序が決まっていることを知っているからです。R の関数でエラーが発生した場合は、その関数に渡している少なくとも一つの変数に問題がある可能性があります。

`percent_change()`関数に文字列を渡すとこうなります。

.

```
percent_change("what", "happens")
```

Error in second_number - first_number: non-numeric argument to binary operator

This is the point:

優れた R のプログラマーは、あらかじめ不適切な入力によるエラーを予想して、一般的なエラー表示の代わりに役に立つ提案を出力するでしょう。特定のエラーはあまり明白ではありません。エラーを解釈する**あなた** は関数が正しく機能するために数字が必要なことを知っています。しかし新しいユーザーはそれを知らないかもしれません。これは、あなたが関数が機能しないときに感じることと同じです。

エラーを Google で確認するか、コードを覗いて、エラーの原因を確認できるかどうか、または関数に引数を渡したらエラーが発生する可能性があるかどうかを確認します。

私たちがコーディングして他の人と共有しているとき、他の人が使うであろうすべての方法を予測することはできません。関数の作成者にメッセージを送るか、パッケージを書いた場合は他の人からのフィードバックを歓迎すべきです。

この仕組みが R コミュニティに参加することをとても素晴らしいものにしています。私たちは改善を追求しているのです。

演習

[演習](http://code.r-journalism.com/chapter-1/#section-intro-to-r) で知識を身につけましょう。

エクササイズアプリの実行方法に関する説明は、このセクションの [紹介ページ](http://learn.r-journalism.com/en/how_to_use_r/) にあります。