CHAPTER 6

First Steps with R for Excel Users

In Chapter 1 you learned how to conduct exploratory data analysis in Excel. You may recall from that chapter that John Tukey is credited with popularizing the practice of EDA. Tukey's approach to data inspired the development of several statistical programming languages, including S at the legendary Bell Laboratories. In turn, S inspired R. Developed in the early 1990s by Ross Ihaka and Robert Gentleman, the name is a play both on its derivation from S and its cofounders' first names. R is open source and maintained by the R Foundation for Statistical Computing. Because it was built primarily for statistical computation and graphics, it's most popular among researchers, statisticians, and data scientists.

R was developed specifically with statistical analysis in mind.

Downloading R

To get started, navigate to the R Project's website (*https://r-project.org*). Click the link at the top of the page to download R. You will be asked to choose a mirror from the Comprehensive R Archive Network (CRAN). This is a network of servers that distributes R source code, packages, and documentation. Choose a mirror near you to download R for your operating system.

Getting Started with RStudio

You've now installed R, but we will make one more download to optimize our coding experience. In Chapter 5, you learned that when software is open source, anyone is free to build on, distribute, or contribute to it. For example, vendors are welcome to offer an *integrated development environment* (IDE) to interact with the code. The RStudio IDE combines tools for code editing, graphics, documentation, and more under a single interface. This has become the predominant IDE for R programming in its decade or so on the market, with users building everything from interactive dashboards (Shiny) to research reports (R Markdown) with its suite of products.

You may be wondering, *if RStudio is so great, why did we bother installing R?* These are in fact two distinct downloads: we downloaded R for the *code base*, and RStudio for an *IDE to work with the code*. This decoupling of applications may be unfamiliar to you as an Excel user, but it's quite common in the open source software world.

RStudio is a platform to *work with* R code, not the code base itself. First, download R from CRAN; then download RStudio.

To download RStudio, head to the download page (*https://oreil.ly/rfP1X*) of its website. You will see that RStudio is offered on a tiered pricing system; select the free RStudio Desktop. (RStudio is an excellent example of how to build a solid business on top of open source software.) You'll come to love RStudio, but it can be quite overwhelming at first with its many panes and features. To overcome this initial discomfort, we'll take a guided tour.

First, head to the home menu and select File → New File → R Script. You should now see something like Figure 6-1. There are lots of bells and whistles here; the idea of an IDE is to have all the tools needed for code development in one place. We'll cover the features in each of the four panes that you should know to get started.

Located in the lower left-hand corner of RStudio, the *console* is where commands are submitted to R to execute. Here you will see the `>` sign followed by a blinking cursor. You can type operations here and then press Enter to execute. Let's start with something very basic, like finding `1 + 1`, as in Figure 6-2.

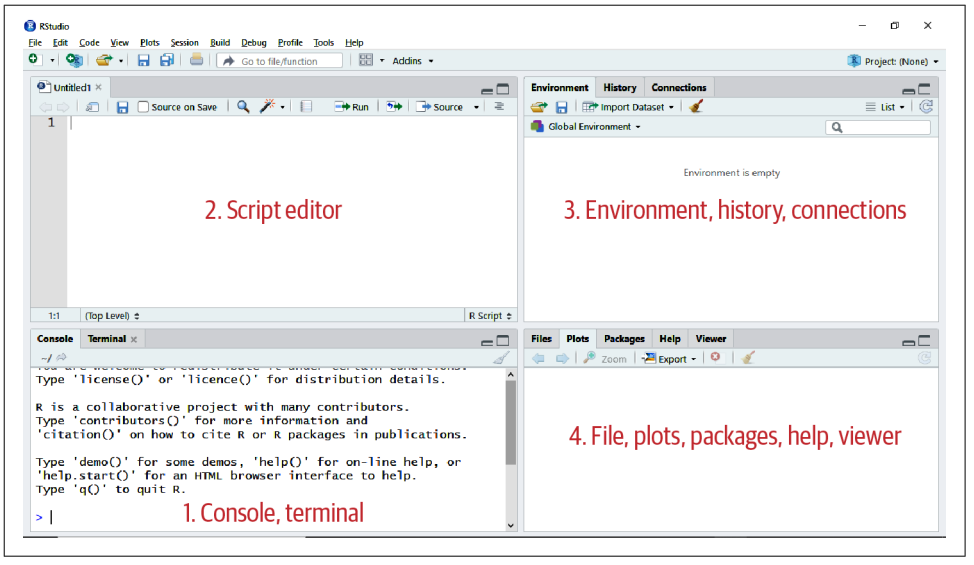

Figure 6-1. The RStudio IDE

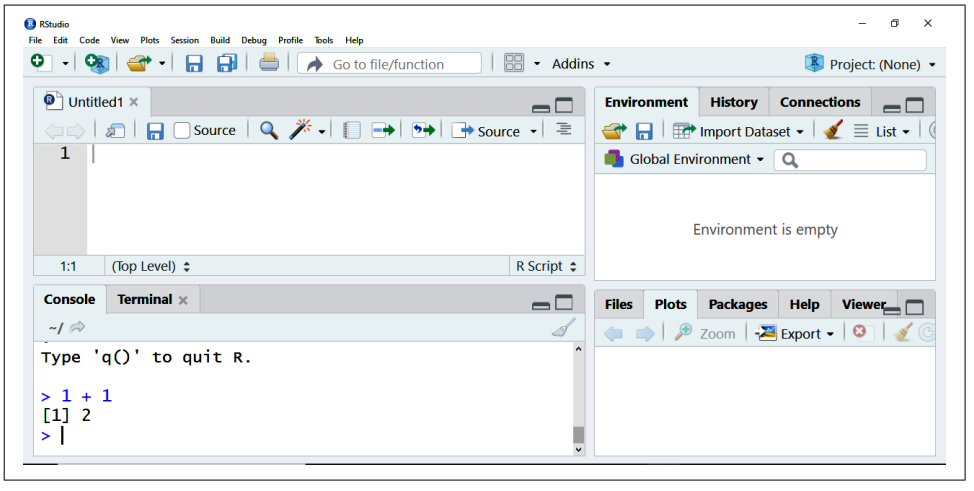

Figure 6-2. Coding in RStudio, starting with 1 + 1

You may have noticed that a `[1]` appears before your result of `2`. To understand what this means, type and execute `1:50` in the console. The `:` operator in R will produce all numbers in increments of 1 between a given range, akin to the fill handle in Excel. You should see something like this:

```
1:50
#> [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
#> [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
#> [47] 47 48 49 50
```

These bracketed labels indicate the numeric position of the first value for each line in the output.

While you can continue to work from here, it's often a good idea to first write your commands in a *script*, and then send them to the console. This way you can save a long-term record of the code you ran. The script editor is found in the pane immediately above the console. Enter a couple of lines of simple arithmetic there, as in Figure 6-3.

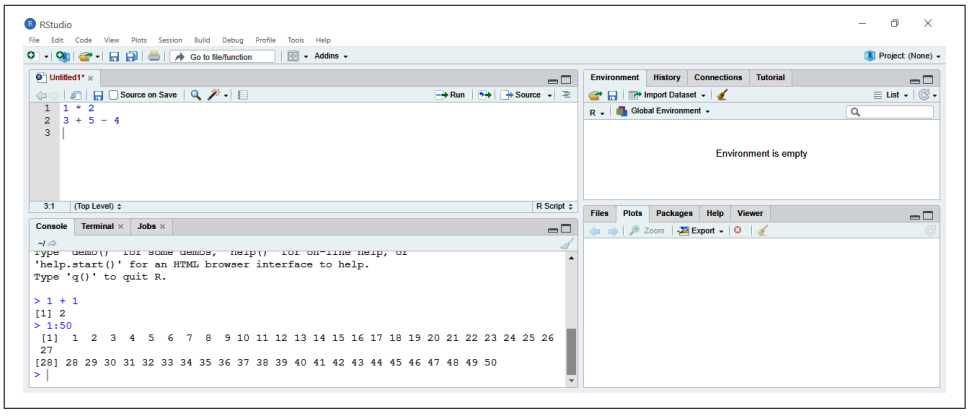

Figure 6-3. Working with the script editor in RStudio

Place your cursor in line 1, then hover over the icons at the top of the script editor until you find one that says "Run the current line or selection." Click that icon and two things will happen. First, the active line of code will be executed in the console. The cursor will also drop to the next line in the script editor. You can send multiple lines to the console at once by selecting them and clicking that icon. The keyboard shortcut for this operation is Ctrl + Enter for Windows, Cmd + Return for Mac. As an Excel user, you're probably a keyboard shortcut enthusiast; RStudio has an abundance of them, which can be viewed by selecting Tools → Keyboard Shortcuts Help.

Let's save our script. From the menu head to File → Save. Name the file *ch-6*. The file extension for R scripts is *.r*. The process of opening, saving, and closing R scripts may remind you of working with documents in a word processor; after all, they are both written records.

We'll now head to the lower-right pane. You will see five tabs here: Files, Plots, Packages, Help, Viewer. R provides plenty of help documentation, which can be viewed in this pane. For example, we can learn more about an R function with the `?` operator.

As an Excel user, you know all about functions such as `VLOOKUP()` or `SUMIF()`. Some R functions are quite similar to those of Excel; let's learn, for example, about R's square-root function, `sqrt()`. Enter the following code into a new line of your script and run it using either the menu icon or the keyboard shortcut:

```
?sqrt
```

A document titled "Miscellaneous Mathematical Functions" will appear in your Help window. This contains important information about the `sqrt()` function, the arguments it takes, and more. It also includes this example of the function in action:

```
require(stats) # for spline
require(graphics)
xx <- -9:9
plot(xx, sqrt(abs(xx)),  col = "red")
lines(spline(xx, sqrt(abs(xx)), n=101), col = "pink")
```

Don't worry about making sense of this code right now; just copy and paste the selection into your script, highlighting the complete selection, and run it. A plot will now appear as in Figure 6-4. I've resized my RStudio panes to make the plot larger. You will learn how to build R plots in Chapter 8.

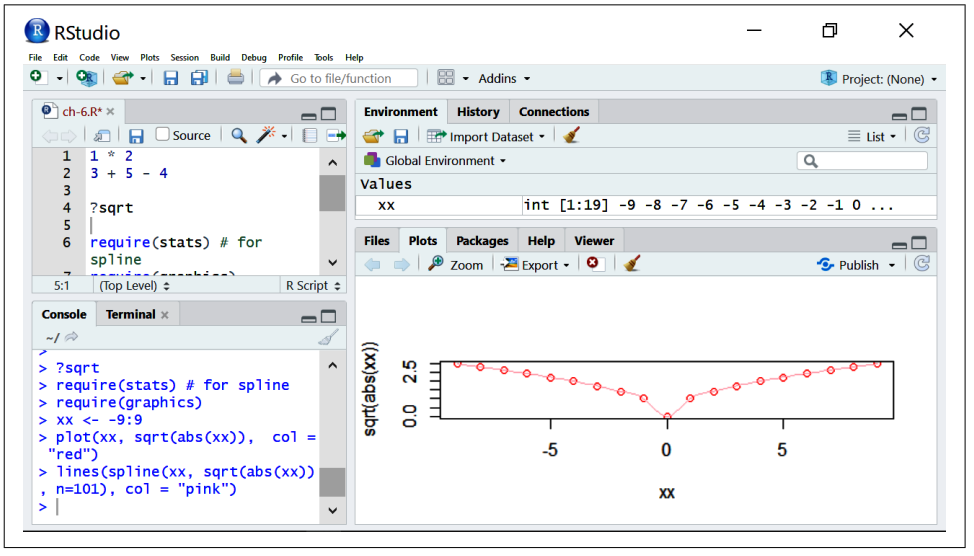

Figure 6-4. Your first R plot

Now, look to your upper-right pane: Environment, History, Connections. The Environment tab lists something called `xx` next to what looks to be some set of integers. What is this? As it turns out, *you* created this with the code I told you to run blindly from the `sqrt()` documentation. In fact, much of what we do in R will focus around what is shown here: an *object*.

As you likely noticed, there are several panes, icons, and menu options we overlooked in this tour of RStudio. It's such a feature-rich IDE: don't be afraid to explore, experiment, and search-engine your way to learning more. But for now, you know enough about getting around in RStudio to begin learning R programming proper. You've already seen that R can be used as a fancy calculator. Table 6-1 lists some common arithmetic operators in R.

Table 6-1. Common arithmetic operators in R

Operator	Description
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`^`	Exponent
`%%`	Modulo
`%/%`	Floor division

You may be less familiar with the last two operators in Table 6-1: the *modulo* returns the remainder of a division, and *floor division* rounds the division's result down to the nearest integer.

Like Excel, R follows the order of operations in arithmetic.

```
# Multiplication before addition
3 * 5 + 6
#> [1] 21

# Division before subtraction
2 / 2 - 7
#> [1] -6
```

What's the deal with the lines containing the hash (`#`) and text? Those are *cell comments* used to provide verbal instructions and reminders about the code. Comments help other users—and ourselves at a later date—understand and remember what the code is used for. R does not execute cell comments: this part of the script is for the programmer, not the computer. Though comments can be placed to the right of code, it's preferred to place them above:

```
1 * 2 # This comment is possible
#> [1] 2

# This comment is preferred
2 * 1
#> [1] 2
```

You don't need to use comments to explain *everything* about what your code is doing, but do explain your reasoning and assumptions. Think of it as, well, a commentary. I will continue to use comments in this book's examples where relevant and helpful.

Get into the habit of including comments to document your objectives, assumptions, and reasoning for writing the code.

As previously mentioned, functions are a large part of working in R, just as in Excel, and often look quite similar. For example, we can take the absolute value of –100:

```
# What is the absolute value of -100?
abs(-100)
#> [1] 100
```

However, there are some quite important differences for working with functions in R, as these errors indicate.

```
# These aren't going to work
ABS(-100)
#> Error in ABS(-100) : could not find function "ABS"
Abs(-100)
#> Error in Abs(-100) : could not find function "Abs"
```

In Excel, you can enter the `ABS()` function as lowercase `abs()` or proper case `Abs()` without a problem. In R, however, the `abs()` function *must* be lowercase. This is because R is *case-sensitive*. This is a major difference between Excel and R, and one that is sure to trip you up sooner or later.

R is a case-sensitive language: the `SQRT()` function is not the same as `sqrt()`.

Like in Excel, some R functions, like `sqrt()`, are meant to work with numbers; others, like `toupper()`, work with characters:

```
# Convert to upper case
toupper('I love R')
#> [1] "I LOVE R"
```

Let's look at another case where R behaves similarly to Excel, with one exception that will have huge implications: comparison operators. This is when we compare some relationship between two values, such as whether one is greater than the other.

```
# Is 3 greater than 4?
3 > 4
#> [1] FALSE
```

R will return a `TRUE` or `FALSE` as a result of any comparison operator, just as would Excel. Table 6-2 lists R's comparison operators.

Table 6-2. Comparison operators in R

Operator	Meaning
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`!=`	Not equal to
`==`	Equal to

Most of these probably look familiar to you, except… did you catch that last one? That's correct, you do not check whether two values are equal to one another in R with one equal sign, but with *two*. This is because a single equal sign in R is used to *assign objects*.

Objects Versus Variables

Stored objects are also sometimes referred to as *variables* because of their ability to be overwritten and change values. However, we've already been referring to *variables* in the statistical sense throughout this book. To avoid this confusing terminology, we will continue to refer to "objects" in the programming sense and "variables" in the statistical.

If you're not quite sure what the big deal is yet, bear with me for another example. Let's assign the absolute value of –100 to an object; we'll call it `my_first_object`.

```
# Assigning an object in R
my_first_object = abs(-100)
```

You can think of an object as a shoebox that we are putting a piece of information into. By using the `=` operator, we've stored the result of `abs(-100)` in a shoebox called `my_first_object`. We can open this shoebox by *printing* it. In R, you can simply do this by running the object's name:

```
# Printing an object in R
my_first_object
#> [1] 100
```

Another way to assign objects in R is with the <- operator. In fact, this is usually preferred to = in part to avoid the confusion between it and ==. Try assigning another object using this operator, then printing it. The keyboard shortcut is Alt+- (Alt +minus) on Windows, and Option-- (Option-minus) on Mac. You can get creative with your functions and operations, like I did:

```
my_second_object <- sqrt(abs(-5 ^ 2))
my_second_object
#> [1] 5
```

Object names in R must start with a letter or dot and should contain only letters, numbers, underscores, and periods. There are also a few off-limit keywords. That leaves a lot of margin for "creative" object naming. But good object names are indicative of the data they store, similar to how the label on a shoebox signals what kind of shoe is inside.

R and Programming Style Guides

Some individuals and organizations have consolidated programming conventions into "style guides," just as a newspaper might have a style guide for writing. These style guides cover which assignment operators to use, how to name objects, and more. One such R style guide has been developed by Google and is available online (*https://oreil.ly/fAeJi*).

Objects can contain different types or *modes* of data, just as you might have different categories of shoeboxes. Table 6-3 lists some common data types.

Table 6-3. Common data types in R

Data type	Example
Character	`'R'`, `'Mount'`, `'Hello, world'`
Numeric	`6.2`, `4.13`, `3`
Integer	`3L`, `-1L`, `12L`
Logical	`TRUE`, `FALSE`, `T`, `F`

Let's create some objects of different modes. First, character data is often enclosed in single quotations for legibility, but double quotes also work and can be particularly helpful if you want to include a single quote as part of the input.

```
my_char <- 'Hello, world'
my_other_char <- "We're able to code R!"
```

Numbers can be represented as decimals or whole numbers:

```
my_num <- 3
my_other_num <- 3.21
```

However, whole numbers can also be stored as a distinct integer data type. The `L` included in the input stands for *literal*; this term comes from computer science and is used to refer to notations for fixed values:

```
my_int <- 12L
```

`T` and `F` will by default evaluate as logical data to `TRUE` and `FALSE`, respectively:

```
my_logical <- FALSE
my_other_logical <- F
```

We can use the `str()` function to learn about the *structure* of an object, such as its type and the information contained inside:

```
str(my_char)
#> chr "Hello, world"
str(my_num)
#> num 3
str(my_int)
#> int 12
str(my_logical)
#> logi FALSE
```

Once assigned, we are free to use these objects in additional operations:

```
# Is my_num equal to 5.5?
my_num == 5.5
#> [1] FALSE

# Number of characters in my_char
nchar(my_char)
#> [1] 12
```

We can even use objects as input in assigning other objects, or reassign them:

```
my_other_num <- 2.2
my_num <- my_num/my_other_num
my_num
#> [1] 1.363636
```

"So what?" you may be asking. "I work with a lot of data, so how is assigning each number to its own object going to help me?" Fortunately, you'll see in Chapter 7 that it's possible to combine multiple values into one object, much as you might do with ranges and worksheets in Excel. But before that, let's change gears for a moment to learn about packages.

Packages in R

Imagine if you weren't able to download applications on your smartphone. You could make phone calls, browse the internet, and jot notes to yourself—still pretty handy. But the real power of a smartphone comes from its applications, or apps. R ships much like a "factory-default" smartphone: it's still quite useful, and you could accomplish nearly anything necessary with it if you were forced to. But it's often more efficient to do the R equivalent of installing an app: *installing a package*.

The factory-default version of R is called "base R." Packages, the "apps" of R, are shareable units of code that include functions, datasets, documentation, and more. These packages are built on top of base R to improve functionality and add new features.

Earlier, you downloaded base R from CRAN. This network also hosts over 10,000 packages that have been contributed by R's vast user base and vetted by CRAN volunteers. This is your "app store" for R, and to repurpose the famous slogan, "There's a package for that." While it's possible to download packages elsewhere, it's best as a beginner to stick with what's hosted on CRAN. To install a package from CRAN, you can run `install.packages()`.

CRAN Task Views

It can be difficult for a newcomer to identify the right R packages for their needs. Fortunately, the CRAN team provides something like "curated playlists" of packages for given use cases with CRAN Task Views (*https://oreil.ly/q31wg*). These are bundles of packages meant to assist with everything from econometrics to genetics and provide a great landscape of helpful R packages. As you continue to learn the language, you'll get more comfortable locating and sizing up the right package for your requirements.

We'll be using packages in this book to help us with tasks like data manipulation and visualization. In particular, we'll be using the `tidyverse`, which is actually a *collection* of packages designed to be used together. To install this collection, run the following in the console:

```
install.packages('tidyverse')
```

You've just installed a number of helpful packages; one of which, `dplyr` (usually pronounced *d-plier*), includes a function `arrange()`. Try opening the documentation for this function and you'll receive an error:

```
?arrange
#> No documentation for ‘arrange’ in specified packages and libraries:
#> you could try ‘??arrange’
```

To understand why R can't find this `tidyverse` function, go back to the smartphone analogy: even though you've installed an app, you still need to open it to use it. Same with R: we've installed the package with `install.packages()`, but now we need to call it into our session with `library()`:

```
# Call the tidyverse into our session
library(tidyverse)
#> -- Attaching packages ------------------------- tidyverse  1.3.0 --
#> v ggplot2 3.3.2     v purrr   0.3.4
#> v tibble  3.0.3     v dplyr   1.0.2
#> v tidyr   1.1.2     v stringr 1.4.0
#> v readr   1.3.1     v forcats 0.5.0
#> -- Conflicts ---------------------------- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

The packages of `tidyverse` are now available for the rest of your R session; you can now run the example without error.

Packages are *installed* once, but *called* for each session.

Upgrading R, RStudio, and R Packages

RStudio, R packages, and R itself are constantly improving, so it's a good idea to occasionally check for updates. To update RStudio, navigate to the menu and select Help → Check for Updates. If you're due for an update, RStudio will guide you through the steps.

To update all packages from CRAN, you can run this function and follow the prompted steps:

```
update.packages()
```

You can also update packages from the RStudio menu by going to Tools → Check for Package Updates. An Update Packages menu will appear; select all of the packages that you wish to update. You can also install packages via the Tools menu.

Upgrading R itself is unfortunately more involved. If you are on a Windows computer, you can use the `updateR()` function from the package `installr` and follow its instructions:

```
# Update R for Windows
install.packages('installr')
library(installr)
updateR()
```

For Mac, return to the CRAN website (*https://cran.r-project.org*) to install the latest version of R.

Conclusion

In this chapter, you learned how to work with objects and packages in R and got the hang of working with RStudio. You've learned a lot; I think it's time for a break. Go ahead and save your R script and close out of RStudio by selecting File → Quit Session. When you do so, you'll be asked: "Save workspace image to ~/.RData?" As a rule, *don't save your workspace image*. If you do, a copy of all saved objects will be saved so that they'll be available for your next session. While this *sounds* like a good idea, it can get cumbersome to store these objects and keep track of *why* you stored them in the first place.

Instead, rely on the R script itself to regenerate these objects in your next session. After all, the advantage of a programming language is that it's reproducible: no need to drag objects around with us if we can create them on demand.

Err on the side of *not* saving your workspace image; you should be able to re-create any objects from a previous session using your script.

To prevent RStudio from preserving your workspace between sessions, head to the home menu and go to Tools → Global Options. Under the General menu, change the two settings under Workspace as shown in Figure 6-5.

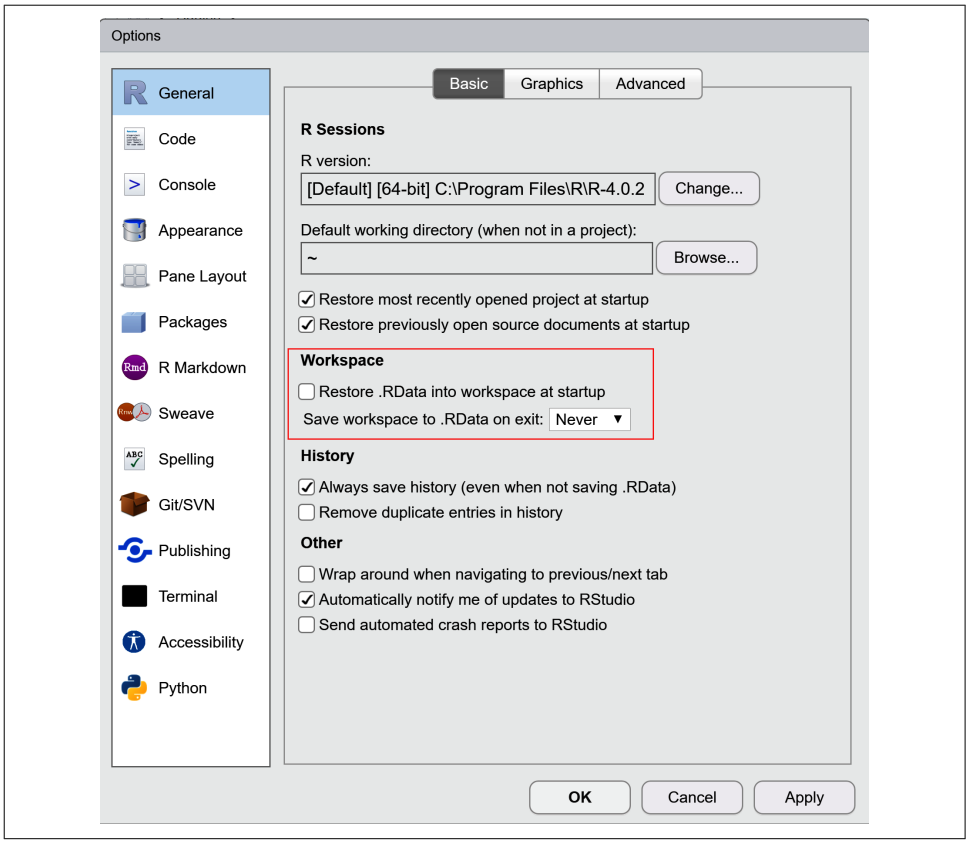

Figure 6-5. Customized workspace options in RStudio

Exercises

The following exercises provide additional practice and insight on working with objects, packages, and RStudio:

1. In addition to being a workhorse of a tool, RStudio provides endless appearance customizations. From the menu, select Tools → Global Options → Appearance and customize your editor's font and theme. For example, you may decide to use a "dark mode" theme.
2. Using a script in RStudio, do the following:
   - Assign the sum of 1 and 4 as `a`.
   - Assign the square root of `a` as `b`.
   - Assign `b` minus 1 as `d`.
   - What type of data is stored in `d`?
   - Is `d` greater than 2?
3. Install the `psych` package from CRAN, and load it into your session. Use comments to explain the differences between installing and loading a package.

Along with these exercises, I encourage you to begin using R immediately in your day-to-day work. For now, this may just involve using the application as a fancy calculator. But even this will help you get comfortable with using R and RStudio.