

Tutorial 0a:

Introduction to R Basics

Tutorial 0a: Introduction to R Basics

This tutorial will go over some basics about **programming in R**.

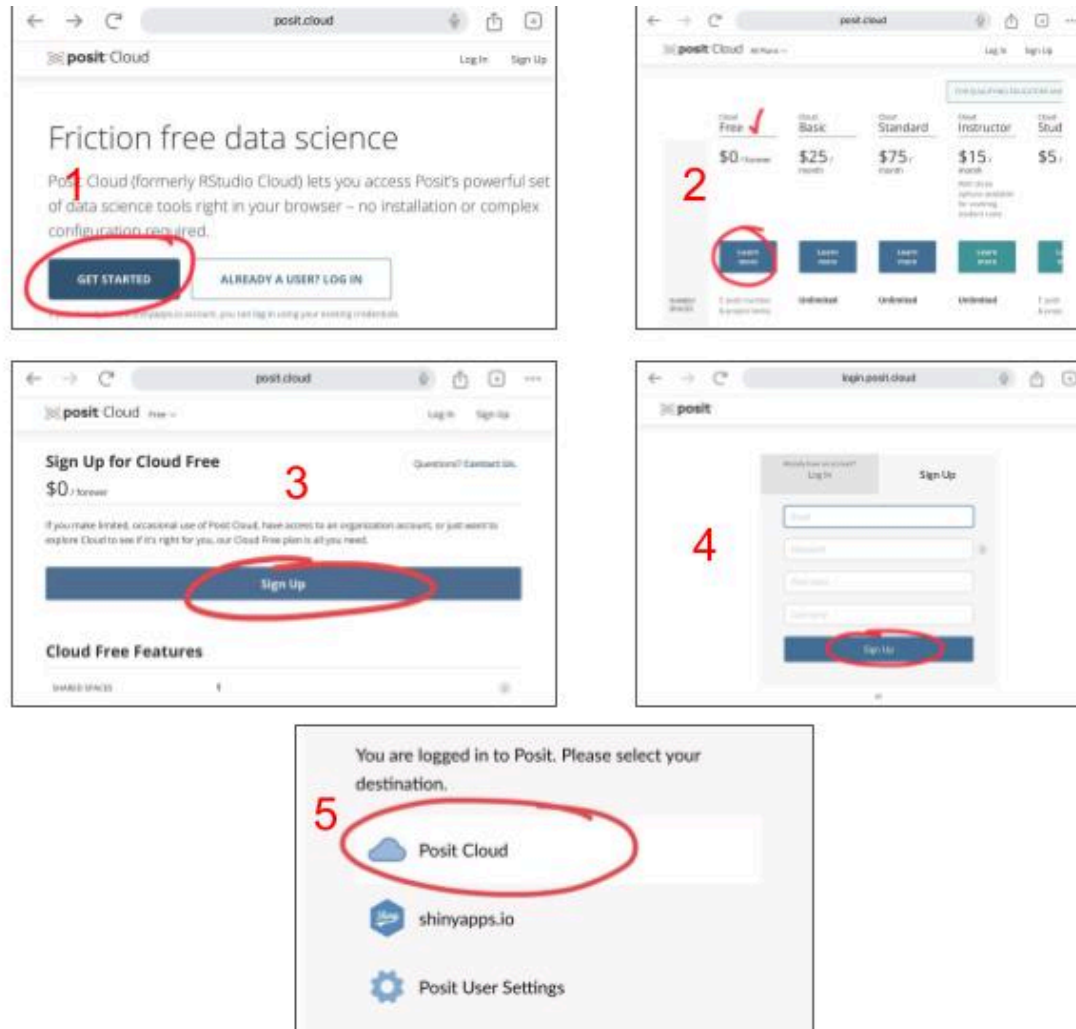
We hope that you'll get a feel for **what R can do** as well as learn **where you can learn more** to use it on your own (great resources are listed at the end).

Getting Started with R

- R is a programming language **developed to analyze data**.
- **Built-in tools** for common analysis tasks.
- Countless additional tools, called **packages**, have been developed by the R community and can be downloaded.
- Anyone can use the language to write their own code to perform **new tasks and analyses**.
- R is widely-used, free, and open-source with great community support.

Posit Cloud (or RStudio, if you have it)

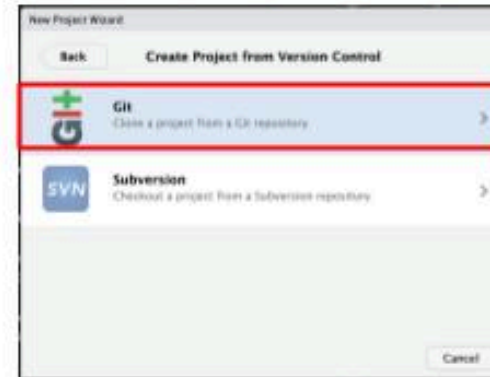
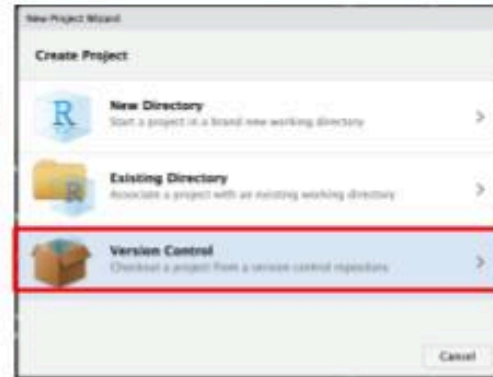
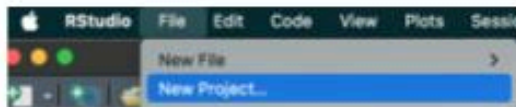
If you haven't already installed R and RStudio on your computer, we recommend using Posit Cloud for today.



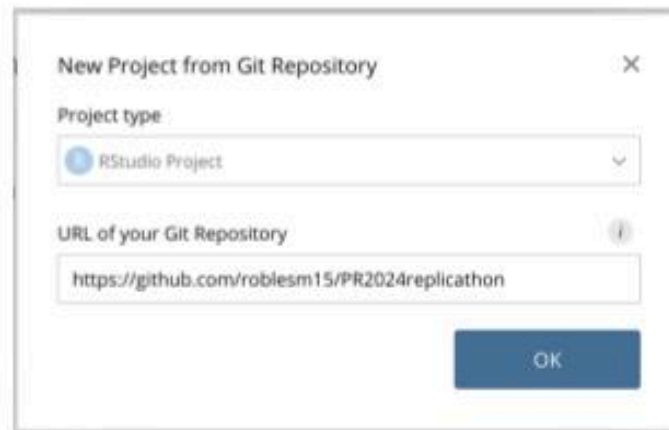
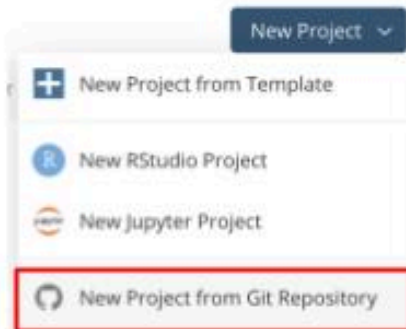
Project Setup

URL: <https://github.com/roblesm15/PR2024replicathon>

In RStudio

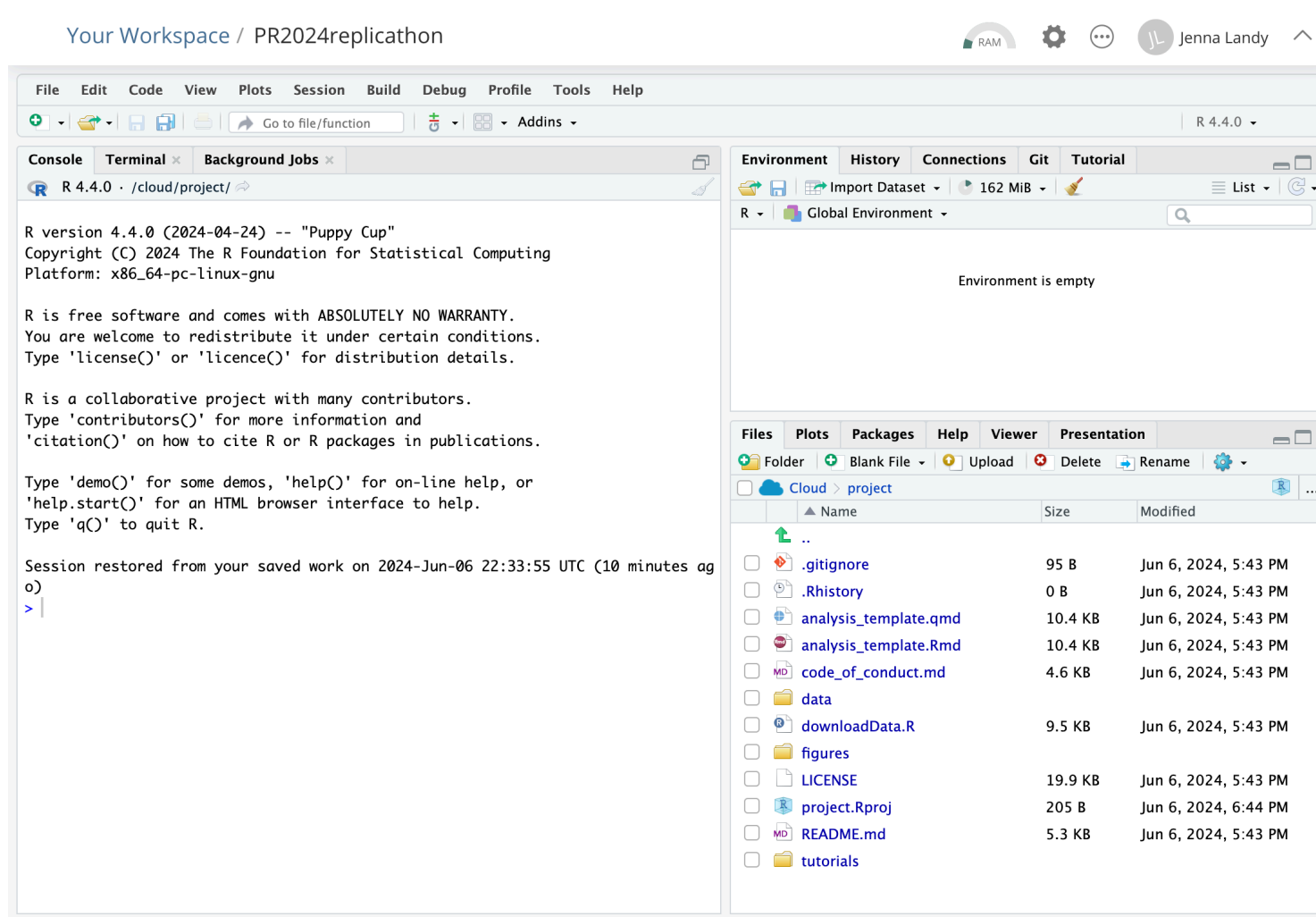


On Posit Cloud



The Console

Now you are ready to start working with data! On your computer, identify the **console**, the large pane on the left.



The Console

When you type a line of code into the console and hit enter the command gets *executed*. For example, try using R as a calculator by typing:

```
1 2 + 3
```

```
[1] 5
```

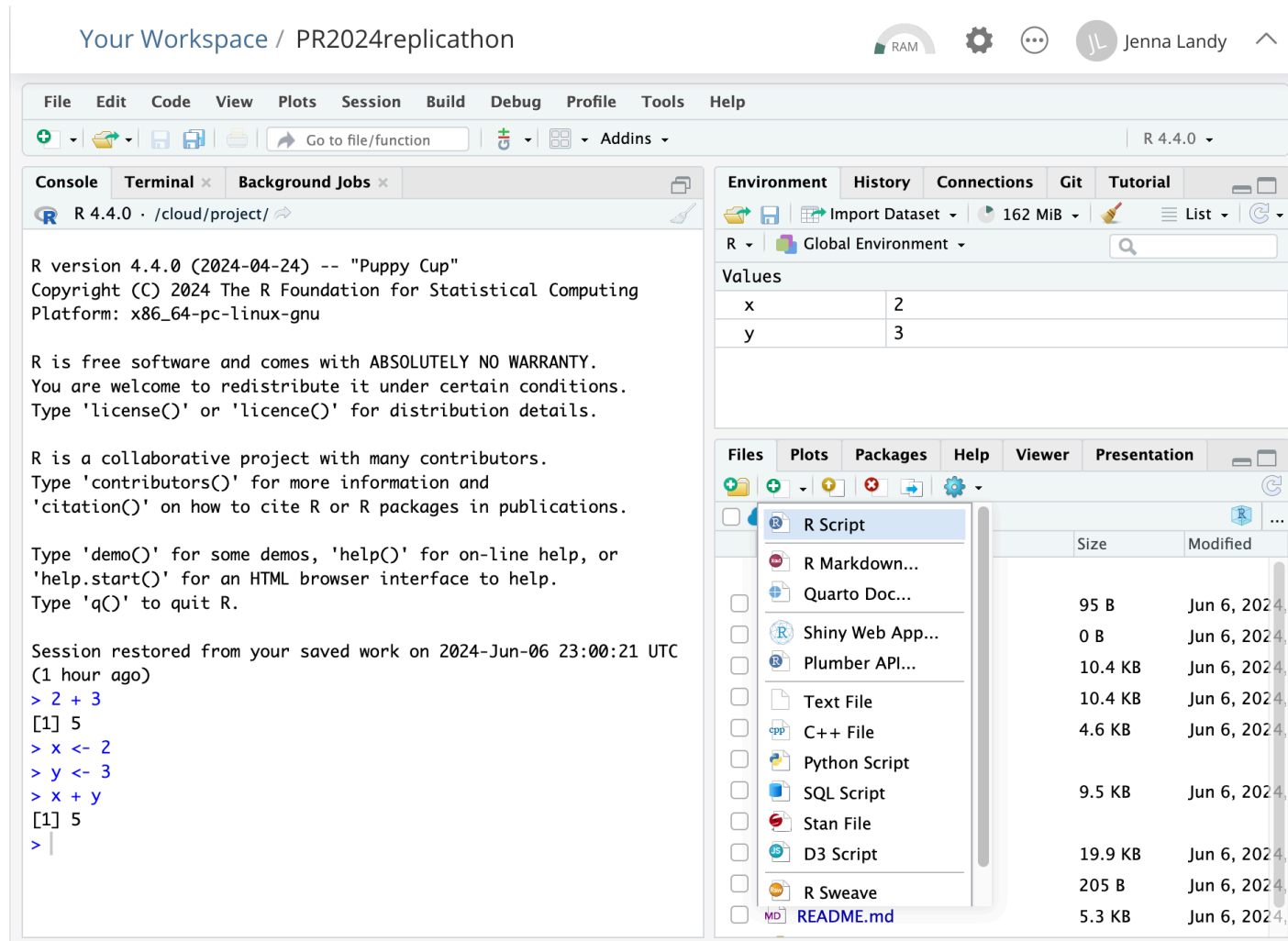
We can also assign values to variables. Try the following:

```
1 x <- 2  
2 y <- 3  
3 x + y
```

```
[1] 5
```

File Browser

The bottom right pane is the **file browser**. Instead of running code in the console, you can save code in a file to be able to revisit it in the future.



The screenshot displays the RStudio IDE interface. The top menu bar includes File, Edit, Code, View, Plots, Session, Build, Debug, Profile, Tools, and Help. The top toolbar contains icons for creating a new file, opening a file, saving, and navigating to a file/function. The top right shows the RAM usage, settings, and the user's name, Jenna Landy.

The main workspace is divided into four panes:

- Console:** Displays the R version (4.4.0) and the session restored from saved work on 2024-Jun-06 23:00:21 UTC (1 hour ago). The console shows the following code and output:

```
> 2 + 3
[1] 5
> x <- 2
> y <- 3
> x + y
[1] 5
> |
```
- Environment:** Shows the Global Environment with variables x and y. The values are 2 and 3, respectively.
- Files:** A file browser pane showing a list of files in the current directory. The files are: R Script, R Markdown..., Quarto Doc..., Shiny Web App..., Plumber API..., Text File, C++ File, Python Script, SQL Script, Stan File, D3 Script, R Sweave, and README.md. The files are listed with their sizes and modification dates (all from Jun 6, 2024).
- Plots:** Empty.

R Scripts

Your Workspace / PR2024replicathon

RAM ⚙️ ⋮ JL Jenna Landy ^

File Edit Code View Plots Session Build Debug Profile Tools Help

Go to file/function

Addins

R 4.4.0

my_script.R

Run

Source

```
1 2 + 3
2
3 x <- 2
4 y <- 3
5 x + y
```

5:6 (Top Level) R Script

Console Terminal Background Jobs

R 4.4.0 · /cloud/project/

Session restored from your saved work on 2024-Jun-06 23:00:21 UTC (1 hour ago)

> 2 + 3
[1] 5
> x <- 2
> y <- 3
> x + y
[1] 5
> source("/cloud/project/my_script.R")
>

Environment History Connections Git Tutorial

Import Dataset

168 MiB

List

R Global Environment

Values

x	2
y	3

Files Plots Packages Help Viewer Presentation

Cloud > project

	Name	Size	Modified
	..		
	.gitignore	95 B	Jun 6, 2024
	.Rhistory	0 B	Jun 6, 2024
	analysis_template.qmd	10.4 KB	Jun 6, 2024
	analysis_template.Rmd	10.4 KB	Jun 6, 2024
	code_of_conduct.md	4.6 KB	Jun 6, 2024
	data		
	downloadData.R	9.5 KB	Jun 6, 2024
	figures		
	LICENSE	19.9 KB	Jun 6, 2024
	my_script.R	26 B	Jun 6, 2024
	project.Rproj	205 B	Jun 6, 2024

Quarto Documents

my_script.R x my_quarto_file.qmd x

Source

Visual

Outline

```
1
2 Add two numbers:
3 ```{r}
4 2 + 3
5 ```
6
7 Set two numbers to variables and then add them:
8 ```{r}
9 x <- 2
10 y <- 3
11 x + y
12 ```
13
```

7:48

(Top Level)

Quarto

Console

Terminal x

Background Jobs x

R 4.4.0 · /cloud/project/

Session restored from your saved work on 2024-Jun-06 23:00:21 UTC (1 hour ago)

```
> 2 + 3
[1] 5
> x <- 2
> y <- 3
> x + y
[1] 5
> source("/cloud/project/my_script.R")
>
```

Environment

History

Connections

Git

Tutorial

R Global Environment

Values

x	2
y	3

Files

Plots

Packages

Help

Viewer

Presentation

Cloud > project

	Name	Size	Modified
	..		
	.gitignore	95 B	Jun 6, 2024
	.Rhistory	0 B	Jun 6, 2024
	analysis_template.qmd	10.4 KB	Jun 6, 2024
	analysis_template.Rmd	10.4 KB	Jun 6, 2024
	code_of_conduct.md	4.6 KB	Jun 6, 2024
	data		
	downloadData.R	9.5 KB	Jun 6, 2024
	figures		
	LICENSE	19.9 KB	Jun 6, 2024
	my_quarto_file.qmd	116 B	Jun 6, 2024
	my_script.R	26 B	Jun 6, 2024

The R Environment

When you download R from CRAN you get what we call *base R*.

- *functions* that are considered fundamental for data analysis.
- *example datasets*, useful as examples when we are learning to use the available functions.

You can see all the available dataset by executing the function `data` like this:

```
1 data()
```

The R Environment

Functions are objects!

- Two parenthesis () let R know that we want the function to be **executed**.
- Without them, R **shows us the code** for the function.

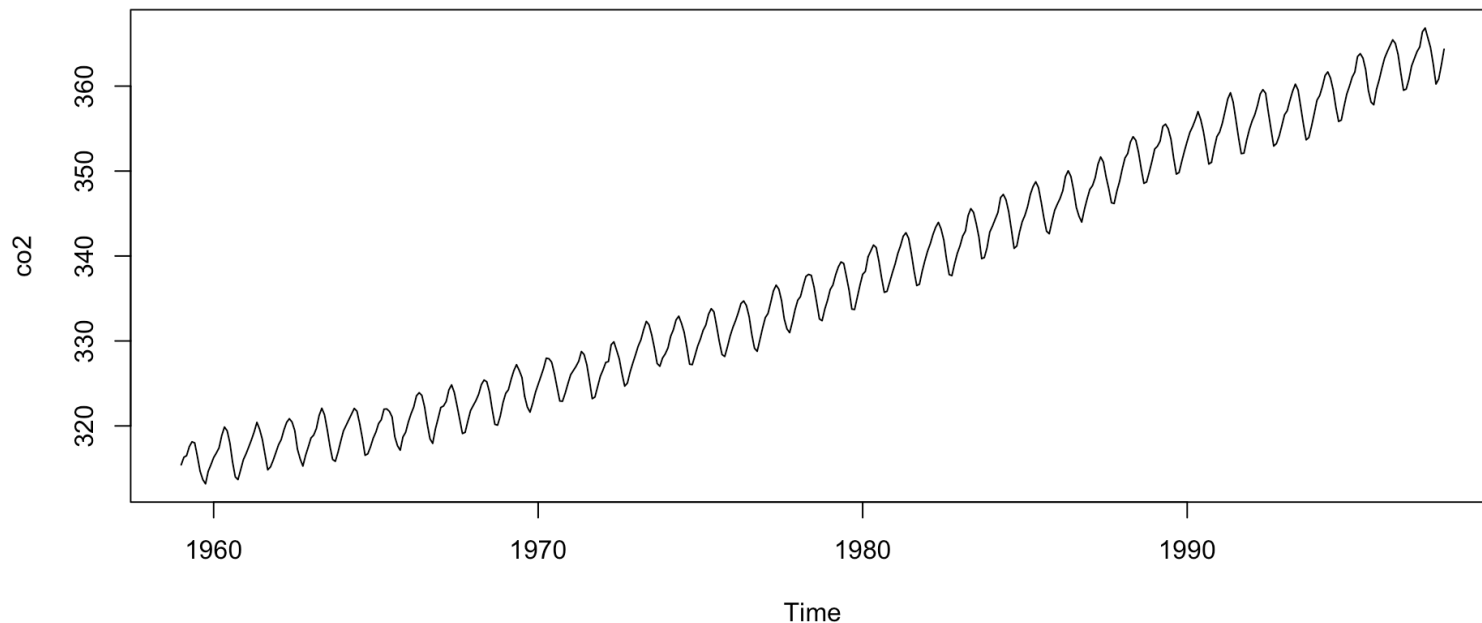
Type the following and note the difference:

```
1 data
```

The R Environment

To see functions at work, we'll use the `co2` dataset to illustrate the function `plot`, one of the base functions.

```
1 plot(co2)
```



This plot shows Mauna Loa Atmospheric CO2 Concentration over time.

R Packages

- R's base functionality is bare bones.
- Data science applications are broad, the statistical toolbox is extensive, and most users need only a small fraction of all the available functionality.
- Specific functionality is available *on demand*, just like apps for smartphones. R does this using *packages*, also called *libraries*.

Using R Packages

Some packages are included with the base download. This includes, for example, the [survival](#) package for survival analysis in R. To bring that functionality to your current session we type:

```
1 library(survival)
```

Most packages are stored on one of three repositories:

- CRAN with [over 14,000 packages](#) - these are vetted.
- [Bioconductor project](#) with [over 1,700 packages](#) for biological data analysis in R - these are more vetted.
- GitHub with new packages every day - these are unvetted.

Installing R Packages

Packages on CRAN can be installed using the `install.packages` function. You can easily install CRAN packages from within R if you know the name of the packages. As an example, if you want to install the package `dplyr`, you would use:

```
1 install.packages("dplyr")
```

This step **only needs to be carried out once** on your machine. We can then load the package into our R sessions using the `library` function:

```
1 library(dplyr)
```

This step **will need to be carried out during every new R session** before using the package. If you ever try to load a package with the `library` command and get an error, it probably means you need to install it first. Note that there are reasons to reinstall packages that already exist in your library (e.g., to receive updated versions of packages).

Getting Help

A key feature you need to know about R is that you can get help for a function using `help` or `?`, like this:

```
1 ?install.packages
2 help("install.packages")
```

These pages are quite detailed about what the function expects as input and what the function produces as output. They also include helpful examples of how to use the function at the end.

Comments

The hash character represents comments, so text following these characters is not interpreted:

```
1 ## This is just a comment; nothing happens after execution.
```

When writing your own R scripts, it is strongly recommended that you write out comments (or include text if using an quarto document) that explain what each section of code is doing. This is very helpful both for collaborators, and for your future self who may have to review, run, or edit your code.

General Programming Principles

Although there are different styles and languages of programming, in essence a piece of code is just a very detailed set of instructions. Each language has its own set of rules and syntax. According to Wikipedia, syntax is

“the set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language.”

Here are some general tips and pitfalls to avoid that will be useful when writing R code

General Programming Principles

1. Case matters: variable names, keywords, functions, and package names are all case-sensitive

```
1 x <- 2
2 X + 8
```

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

2. Avoid using spaces: variable names cannot contain spaces

```
1 my variable <- 10
```

Error: <text>:1:4: unexpected symbol

```
1: my variable
    ^
```

3. Use comments liberally: your future self and others will thank you

```
1 # define scalar variables x and y
2 x <- 2
3 y <- 3
4
5 # add variables x and y
6 x + y
```

```
[1] 5
```

General Programming Principles

4. Pay attention to classes: character strings, numerics, factors, matrices, lists, data.frames, etc., all behave differently in R

```
1 myNumber <- factor(10)
2 str(myNumber)
```

```
Factor w/ 1 level "10": 1
```

```
1 myNumber^2
```

```
[1] NA
```

```
1 as.numeric(myNumber)^2
```

```
[1] 1
```

```
1 as.character(myNumber)^2
```

```
Error in as.character(myNumber)^2: non-numeric argument to binary operator
```

```
1 as.numeric(as.character(myNumber))^2
```

```
[1] 100
```

General Programming Principles

5. Search the documentation for answers: when something unexpected happens, try to find out why by reading the documentation

```
1 mean(c(3, 4, 5, NA))
```

```
[1] NA
```

```
1 mean(c(3, 4, 5, NA), na.rm = TRUE)
```

```
[1] 4
```

6. It's OK to make mistakes: expert R programmers run into (and *learn* from) errors all the time

Don't panic about those error messages!

Using R for Data Science

The process of reading in data and getting it in a format for analysis is often called data “wrangling”. This step may seem simple to an outside observer, but often takes up a significant proportion of the time spent on a data analysis.

Importing Data into R

The first step when preparing to analyze data is to read in the data into R. There are several ways to do this, but we are going to focus on reading in data stored either as an external Comma-Separated Value (CSV) file or “serialized” R data (RDS) object. Small datasets are often stored as Excel files. Although there are R packages designed to read Excel (xls/xlsx) format, you generally want to avoid this. In addition to the values stored in each individual cell, Excel spreadsheets often contain information in annotations (e.g. bold, italics, colors). Parsing these additional annotations is messy and imperfect. Many of these additional “features” of Excel spreadsheets can cause profound headaches for downstream data analysis. For these reasons, it is almost always preferable to save data as a plain-text or R object file. Frequently, plain text files are saved in either comma delimited (CSV) or tab delimited (TSV) format.

Importing CSV Files

Plain-text formats are often easiest for sharing, as commercial software is not required for viewing or working with the data. CSV files can be read into R with the help of the `read.csv` function. Similarly, data.frames can be written to CSV files using the `write.csv` function.

If your data is not a text file but not in CSV format, there are many other helpful functions that will read your data into R, such as `read.table`, `read.delim`, `download.file`. Check out their help pages to learn more.

Importing RDS Files

Another common format for storing data in R is the **RDS** file format. Unlike plain-text files, **RDS** files are *binary* files and cannot be opened and inspected using standard text editors. Instead, **RDS** files must be read in to R using the **readRDS** function. An object in R can be saved as an **RDS** file using the appropriately named **saveRDS** function. While **RDS** files can only be read using R, they are almost always substantially smaller than the corresponding CSV or TSV file.

Throughout, we will be working with tables stored as **RDS** files.

Paths and the Working Directory

If you are reading in a file stored on your computer, the first step is to find the file containing your data and know its *path*.

When you are working in R it is useful to know your *working directory*. This is the directory or folder in which R will save or look for files by default. You can see your working directory through the console by typing:

```
1 getwd()
```

You can also change your working directory using the function `setwd`. Or you can change it through the RStudio menus by clicking on “Session”.

Paths and the Working Directory

The functions that read and write files (there are several in R) assume you mean to look for files or write files in the working directory. Our recommended approach for beginners will have you reading and writing to the working directory. However, you can also type the [full path](#), which will work independently of the working directory.

As an example, let's read in one of the datasets we'll be analyzing today:

```
1 rawFile <- file.path("../", "data", "rawPharmacoData.rds")
2 if (!file.exists(rawFile)) {
3   source(file.path("../", "downloadData.R"), chdir = TRUE)
4 }
5 pharmacoData <- readRDS(rawFile)
```

Paths and the Working Directory

Once we have read a dataset into an *object* (here we called it `pharmacoData`), we are ready to explore it. What exactly is in `pharmacoData`? To check a summary of its contents, the `str()` function is handy (which stands for ‘structure’).

```
1 str(pharmacoData)
```

```
'data.frame':  43427 obs. of  6 variables:
 $ cellLine      : chr  "22RV1" "22RV1" "22RV1" "22RV1" ...
 $ drug          : chr  "17-AAG" "17-AAG" "17-AAG" "17-AAG" ...
 $ doseID        : chr  "doses1" "doses2" "doses3" "doses4" ...
 $ concentration: num  0.0025 0.008 0.025 0.08 0.25 0.8 2.53 8 0.0025 0.008 ...
 $ viability     : num  94.1 86 99.9 85 62 ...
 $ study         : chr  "CCLE" "CCLE" "CCLE" "CCLE" ...
```

Here we see that this object is a `data.frame`. These are one of the most widely used data types in R. They are particularly useful for storing tables. We can also print out the top of the data frame using the `head()` function

```
1 head(pharmacoData)
```

	cellLine	drug	doseID	concentration	viability	study
1	22RV1	17-AAG	doses1	0.0025	94.100	CCLE
2	22RV1	17-AAG	doses2	0.0080	86.000	CCLE
3	22RV1	17-AAG	doses3	0.0250	99.932	CCLE
4	22RV1	17-AAG	doses4	0.0800	85.000	CCLE
5	22RV1	17-AAG	doses5	0.2500	62.000	CCLE
6	22RV1	17-AAG	doses6	0.8000	29.000	CCLE

Paths and the Working Directory

Another option is to open up a ‘spreadsheet’ tab in another RStudio window, which can be done with the `View` function:

```
1 View(pharmacoData)
```

Class Types

There are many different data types in R, but a list of the more common ones include:

- `data.frame`
- `vector`
- `matrix`
- `list`
- `factor`
- `character`
- `numeric`
- `integer`
- `double`

Class Types

Each has its own properties and reading up on them will give you a better understanding of the underlying R infrastructure. See the respective help files for additional information. To see what type of *class* an object is one can use the `class` function.

```
1 class(pharmacoData)
```

```
[1] "data.frame"
```


Extracting Columns

To extract columns from the data.frame we use the `$` character like this (to avoid printing the entire column to the screen, we'll add the `head` function to just print the top):

```
1 head(pharmacoData$drug)
```

```
[1] "17-AAG" "17-AAG" "17-AAG" "17-AAG" "17-AAG" "17-AAG"
```

This now gives us a vector. We can access elements of the vector using the `[` symbol. Here is the 5000th element of the vector:

```
1 pharmacoData$drug[5000]
```

```
[1] "PD-0332991"
```

Vectors

Vectors are a sequence of data elements of the same type (class). Many of the operations used to analyze data are applied to vectors. In R, vectors can be numeric, characters or logical.

The most basic way to create a vector is with the function `c`.

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

Two very common ways of generating vectors are using `:` or the `seq` function.

```
1 x <- 1:5  
2 x <- seq(1, 5)
```

Vectors can have names.

```
1 names(x) <- letters[1:5]  
2 x
```

```
a b c d e  
1 2 3 4 5
```

Functions

Up to now we have used prebuilt functions. However, many times we have to construct our own. We can do this in R using `function`.

```
1 avg <- function(x) {  
2   return(sum(x) / length(x))  
3 }  
4  
5 avg(1:5)
```

```
[1] 3
```

Resources

Material in this tutorial was adapted from Rafael Irizarry's *Introduction to Data Science* course.

If you want to learn more about R after this event, a great place to start is with the [swirl](#) tutorial, which teaches you R programming interactively, at your own pace and in the R console. Once you have R installed, you can install [swirl](#) and run it the following way:

```
1 install.packages("swirl")
2 library(swirl)
3 swirl()
```

There are also many open and free resources and reference guides for R. Two examples are:

- [Quick-R](#): a quick online reference for data input, basic statistics and plots
- R reference card (PDF)[<https://cran.r-project.org/doc/contrib/Short-refcard.pdf>] by Tom Short

Some Useful Books on S/R

Standard texts

- Chambers (2008). *Software for Data Analysis*, Springer. (your textbook)
- Chambers (1998). *Programming with Data*, Springer.
- Venables & Ripley (2002). *Modern Applied Statistics with S*, Springer.
- Venables & Ripley (2000). *S Programming*, Springer.
- Pinheiro & Bates (2000). *Mixed-Effects Models in S and S-PLUS*, Springer.
- Murrell (2005). *R Graphics*, Chapman & Hall/CRC Press.

Other resources

- Springer has a series of books called *Use R!*.
- A longer list of books is at <http://www.r-project.org/doc/bib/R-books.html>

