R Essentials

BY G. RYAN SPAIN

INTRODUCTION

WHAT IS R?

R is a highly extensible, open-source programming language used mainly for statistical analysis and graphics. It is a GNU project very similar to the S language. R's strengths include its varying data structures, which can be more intuitive than data storage in other languages; its built-in statistical and graphical functions; and its large collection of useful plugins that can enhance the language's abilities in many different ways.

R can be run either as a series of console commands, or as full scripts, depending on the use case. It is heavily object-oriented, and allows you to create your own functions. It also has a common API for interacting with most file structures to access data stored outside of R.

USES

R's biggest use case is performing statistical analysis and developing graphical representations. Its built-in packages allow for advanced statistical functions and simple graphic creation. With additional plugins, these abilities can become even more powerful and customizable. R also allows you to save your scripts and data when analyzing data, so that you can review and repeat analysis processes you've done in the past, whether to recreate results or check previous results.

INSTALLATION AND IDE

R can run on many operating systems like Linux, OS X, and Windows. cran.rstudio.com has download links and instructions for these different systems. You will need a Fortran compiler in order to be able to run R. For more extensive details on installation for your particular system, go to cran.r-project.org/doc/manuals/r-release/R-admin.html.

RSTUDIO

RStudio is a popular open-source integrated development environment (IDE) for R. It includes a console for directly executing R commands, as well as an editor for building longer R scripts. It is also able to keep track of and view variable data and access documentation and R graphics in the same environment. Moreover, RStudio allows you to enable additional R packages through the interface without a command. RStudio is available for Windows, Mac, and several Linux operating systems. You can download RStudio at rstudio.com/products/rstudio/download.

STARTING WITH R

BASIC MATHEMATICAL OPERATIONS

At its simplest, R can function like a calculator. Besides basic operators or functions, R does not need code to execute basic calculations. The line `4 + 5` would return the result `[1] 9`. Since R often deals with lengthy, possibly tabular datasets, its output includes index positions—in this case, the `[1]` that printed with our result.

The following basic mathematic operations can be performed:

OPERATOR	DESCRIPTION
+	Addition symbol.
–	Subtraction symbol.
*	Multiplication symbol.
/	Division symbol.
^	Exponent symbol.
%%	Modulus symbol (remainder of division).
()	Used as normal to force precedence in mathematical expressions (the standard order of operations applies).

DECLARING VARIABLES

Variables in R are defined using the <– operator. You can consider the <– as an arrow pointing **from** the value of the variable on the right **to** the variable name on the left. So the expression `x <– 15` would store the value 15 as the variable x. When you declare a variable in R, it does not automatically print the variable or its value; that is, the interface does not return anything when a variable is declared, and will simply ready itself for the next command. To view the contents of a variable in R, use the variable name with no additional expressions or functions and execute the command. This will display the value of the variable.

```
> x <– 15
> x
[1] 15
```

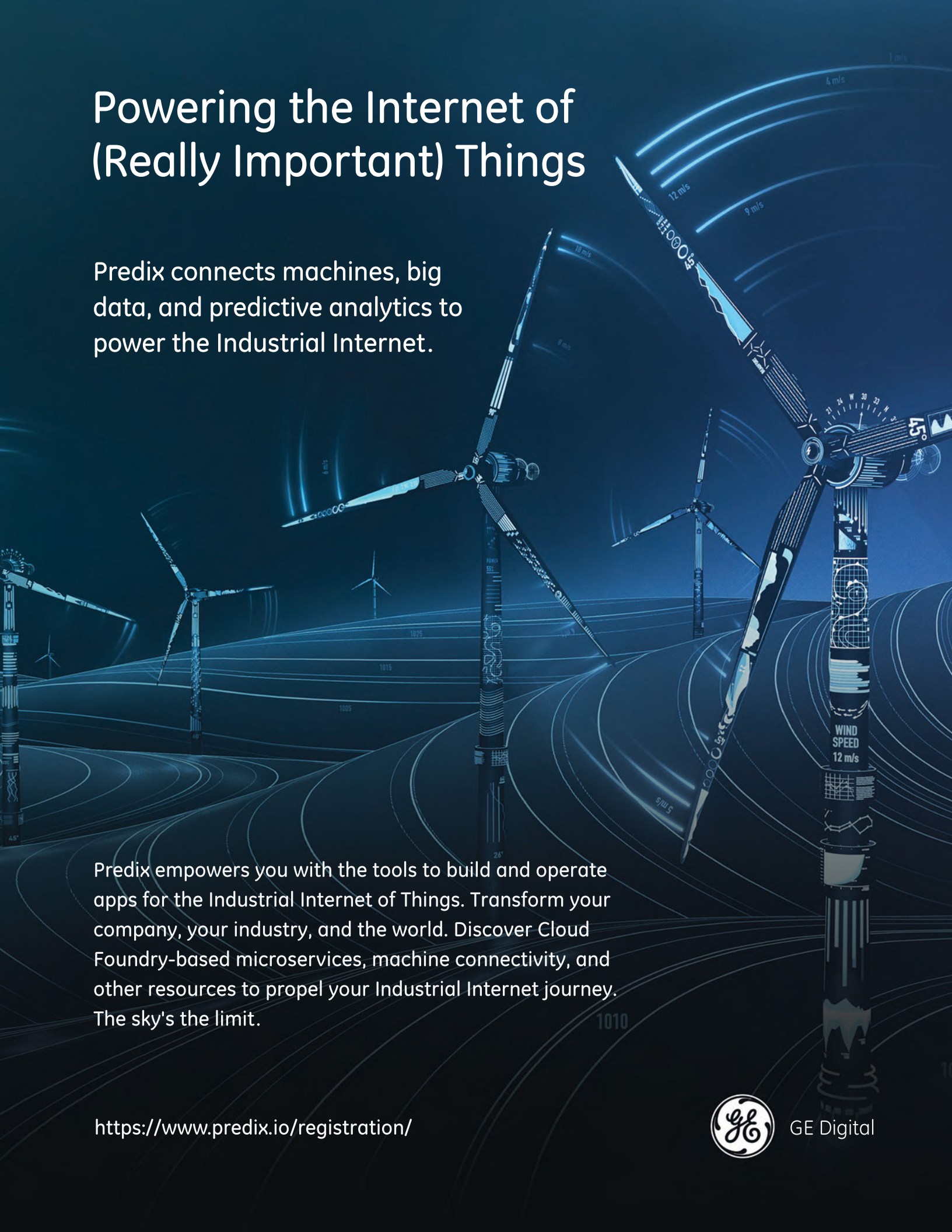
Here, the > denotes the command input, and again the output is printed with its index position within the output.

VECTORS

A vector is the most basic object in R. An "atomic" vector is a linear (flat) collection of values of one basic type. The types of atomic vector are: logical, integer, double, complex, character, and raw.

Powering the Internet of (Really Important) Things

Predix connects machines, big data, and predictive analytics to power the Industrial Internet.



Predix empowers you with the tools to build and operate apps for the Industrial Internet of Things. Transform your company, your industry, and the world. Discover Cloud Foundry-based microservices, machine connectivity, and other resources to propel your Industrial Internet journey. The sky's the limit.

https://www.predix.io/registration/

GE Digital

ATOMIC VECTOR TYPE	DESCRIPTION
Logical	A vector containing values of TRUE or FALSE (i.e. booleans).
Integer	A vector containing integer values.
Double	A vector containing real number values.
Complex	A vector containing real or imaginary number values.
Character	A vector containing character or string values.
Raw	A vector containing bytes.

Even when working with individual values, R considers these values as vectors. For example, in the previous section, when we printed the value of `x`, R displayed the results as a vector of length 1 (which is why even a single value had a position index number assigned to it).

To gather data as a vector, use the `c()` function, which combines its arguments into a vector. Here's a basic example:

```
> x <- c(1, 2, 3, 4)
> x
[1] 1  2  3  4
```

This time, we stored a vector as the variable `x`. When we printed the value of `x`, R returned each value of x in order. Note that R did not return position index values for each element of the vector. R will print a position index each time the results are forced to break a line (generally based on the size of the window in which you're executing commands, or by the default window size of the console).

Note: To learn more about a function in R, you can use the `?` operator or the `help()` function. This will give you more information, including a description, usage, and arguments. To learn more about `c()`, you can enter `?c` or `help(c)`. For more on `help()`, enter `?help` or `help(help)`.

We can get the same result by using the `:` operator, which will create a series from the value of its first argument to the value of its second argument. When using `:` you do not use the `c()` function to combine the data, as this is done automatically.

```
> x <- 1:4
> x
[1] 1  2  3  4
```

Remember that atomic vectors contain values of a single type. If using the `c` function with arguments of different data types, it will force the values into a single type. Two examples of this:

```
> c(1, TRUE, 2)
[1] 1  1  2
> c(1, "two", 3)
[1] "1"    "two"   "3"
```

In the first example, R converted the boolean `TRUE` to 1, since boolean values logically correspond with real numbers, but not the other way around. In the second example, R converted integer number values into strings (note the quotation marks around the returned values), since that conversion is much more logical than attempting to convert a string to an integer.

Operations performed on vectors will affect each element of that vector individually, and return a new vector of those results. For example, `c(1,2,3) * 2` will return a new vector: `2  4  6`. R multiplied each element of the original vector by two, returning a vector of the same length as the original, but with modified values.

Vectors can also be used in operations with other vectors. `c(1,2,3) + c(2,3,4)` will result in an output of `3  5  7`. If the vectors in the operation are not of equal length, R will recycle values of the shorter vector, starting again at the beginning of the shorter vector while still operating in order on the longer vector. Here's an example:

```
> c(1, 2, 3, 4, 5, 6) * c(0, 1)
[1]  0  2  0  4  0  6
```

Notice how the first two operations occurred naturally, but at the third operation, there was no third element of the shorter vector to use. Therefore, R started over at the beginning of the shorter vector and continued the operation. Once the fifth element of the longer vector was reached, R repeated the process.

If the longer vector is not of a length that is a multiple of the shorter vector's length, R will still print the result, but will also return a message warning you that these lengths do not naturally align.

OTHER DATA STRUCTURES

R has many different data structures for different scenarios.

LISTS

Lists are vectors that allow their elements to be any type of object. They are created using the `list()` function.

```
> x <- list(1, "two", c(3, 4))
```

In this example, we've defined `x` as a list consisting of three elements: the number `1`, the string `"two"`, and a vector, `3  4`, of length 2. We can examine the structure of x using the `str()` function.

```
> str(x)
List of 3
 $  : num 1
 $  : chr "two"
 $  : num [1:2] 3 4
```

Remember that each element of the list is a vector; `1` is a numeric vector of length 1, and `two` is a character vector of length 1.

One particularly interesting ability of a list is that it can contain lists within it. Had we defined `x` as `x <- list(1, "two", list(3, 4))`, the `str()` function would have returned:

```
> str(x)
List of 3
 $  : num 1
 $  : chr "two"
 $  :List of 2
   ..$ : num 3
   ..$ : num 4
```

This means that a list is a recursive object (you can test this with the `is.recursive()` function). Lists can be hypothetically nested indefinitely.

FACTORS

A factor is a vector that stores categorical data—data that can be classified by a finite number of categories. These categories are known as the `levels` of a factor.

Say you define `x` as a collection of the strings `"a"`, `"b"`, and `"c"`: `x <- c("b", "c", "b", "a", "c", "c")`.

Using the `factor()` function, you can have R convert the atomic character vector into a factor. R will automatically attempt to determine the levels of the factor; this will produce an error when `factor` is given an argument that is non-atomic. Let's take a look at the factor here:

```
> x <- c("b", "a", "b", "c", "a", "a")
> x <- factor(x)
> \# this can also be written as x <- factor(c("b", "a", "b", "c",
  "a", "a"))
> x
[1] b a b c a a
Levels: a b c
> str(x)
 Factor w/ 3 levels "a","b","c": 2 1 2 3 1 1
> levels(x)
[1] "a" "b" "c"
> table(x)
x
a b c
3 2 1
```

By using the `factor()` function on `x`, R logically categorized the values into "levels." When x was printed, R returned the elements in its original order, but it also printed the levels of the factor. Examining the structure of `x` shows that `x` is a factor with three levels, lists the levels (alphabetically), and then shows which level each element of the factor corresponds to. So here, since `"b"` is alphabetically second, the `2` in `2 1 2 3 1 1` corresponds with `"b"`.

The `levels()` function returns a vector containing only the names of the different levels of the factor. So here, the function `levels(x)` returns the three levels "a", "b", and "c", in order (here from the lowest value of the level to the highest).

The `tables()` function gives a table summarizing the factor. Using the `table()` function on `x` returned the name of the variable, a list of the levels of `x`, and then, underneath, the number of values that occurs in `x` corresponding with the above level. So this table shows us that, in the factor `x`, there are three instances of the level `"a"`, two instances of `"b"`, and one instance of `"c"`.

If the levels of your factor need to be in a particular order, you can use the `factor()` argument `levels` to define the order, and set the argument `ordered` to `TRUE`:

```
> x <- c("b", "a", "b", "c", "a", "a")
> x <- factor(x, levels = c("c", "b", "a"), ordered = TRUE
> x
[1] b a b c a a
Levels: c < b < a
> str(x)
 Ord.factor w/ 3 levels "c"<"b"<"a": 2 3 2 1 3 3
> levels(x)
[1] "c" "b" "a"
> table(x)
x
c b a
1 2 3
```

Now R returned the levels in the order specified by the vector given to the `levels` argument. The `<` (less than) symbol in the output of `x` and `str(x)` indicate that these levels are ordered, and the `str(x)` function reports that the object is an ordered factor.

MATRIXES

A matrix is, in most cases, a two-dimensional atomic data structure (though you can have a one-dimensional matrix, or a non-atomic matrix made from a list). To create a matrix, you can use the `matrix()` function on a vector with the `nrow` and/or `ncol` arguments. `matrix(1:20, nrow = 5)` will produce a matrix with five rows and four columns containing the numbers one through twenty. `matrix(1:20, ncol = 4)` produces the same matrix.

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

The matrix will fill by column unless the argument `byrow` is set to `TRUE`.

Note that the position indexes are assigned to rows **and** columns here. Since a matrix is naturally two-dimensional, R provides column indexes to more easily interact with the matrix. You can use the index vector `[]` to return the value of an individual cell of the matrix. `x[1,2]` will return the value of row one, column 2: `6`. You can also use the index vector to return the values of whole rows or columns. `x[1,]` will return `1 6 11 16`, the elements of the first row of the matrix.

You can also create a matrix by assigning dimensions to a vector using the `dim()` function, as shown here:

```
x <- 1:20
dim(x) <- c(5, 4)
```

This created the same matrix you saw earlier. With the `dim()` function, you can also redefine the dimensions of a matrix. `dim(x) <- c(4,5)` will "redraw" the matrix to have four rows and five columns.

ARRAYS

What happens if the vector you passed to the `dim()` function had more than two elements? If we had written `dim(x) <- c(5, 2, 2)` we would have created another data structure: an array.

Technically, a matrix is specifically a two-dimensional array, but arrays can have unlimited dimensions. When `x` contained 20 elements—`x <- 1:20`—executing `dim(x) <- c(5, 2, 2)` would have given `x` three dimensions. R would represent this as a "series" of matrixes:

```
> x
, , 1
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
, , 2
     [,1] [,2]
[1,]   11   16
[2,]   12   17
[3,]   13   18
[4,]   14   19
[5,]   15   20
```

In the case of an array, the "row" and "column" numbers remain in the same order, and R will show the other dimensions above each matrix. In this case, we received two matrixes (based on the third dimension given) of five rows (based on the first dimension given) and two columns (based on the second dimension given). R displays arrays in order of each dimension given—so if we had an array of **four** dimensions (say `5, 2, 2, 2`), it would print matrixes `, , 1, 1`, then `, , 1, 2`, then `, , 2, 1`, and lastly `, , 2, 2`.

Again, you can use index vectors to find a particular element, or particular elements, of the array. In our three-dimensional array shown earlier, `x[1, 2, 2]` will return `16`. You can see by the way R has printed the array that rows come **before** the first comma, columns come **after** the first comma, and the third dimension of the array comes **after the second comma**.

DATA FRAMES

A data frame is a (generally) two-dimensional structure consisting of

vectors of the same length. Data frames are used often, as they are the closest data structure in R to a spreadsheet or relational data tables. You can use the `data.frame()` function to create a data frame.

```
> x <- data.frame(y = 1:3, z = c("one", "two", "three"),
  stringsAsFactors = FALSE)
> x
  y     z
1 1   one
2 2   two
3 3 three
```

In this example, we have created a data frame with two columns and three rows. Using `y =` and `z =` defines the names of the columns, which will make them easier to access, manipulate and analyze. Here, we've used the argument `stringsAsFactors = FALSE` to make column `z` an atomic character vector instead of a factor. By default, data frames will coerce vectors of strings into factors.

You can use the `names()` function to change the names of your columns. `names(x) <- c("a", "b")` provides a vector of new values to replace the column names, changing the columns to `a` and `b`. To change a certain column or columns, you can use the index vector to specify which column(s) to rename.

```
> names(x)[1] <- "a"
> x
  a     z
1 1   one
2 2   two
3 3 three
```

You can combine data frames with the `cbind()` function or the `rbind()` function. `cbind()` will add the columns of one data frame to another, as long as the frames have the same number of rows.

```
> cbind(x, b = data.frame(c("I", "II", "III"), stringsAsFactors =
  FALSE)))
  a     z   b
1 1   one   I
2 2   two  II
3 3 three III
```

`rbind()` will add the rows of one data frame to the rows of another, so long as the frames have the same number of columns and have the same column names.

```
> rbind(x, data.frame(a = 4, z = "four"))
  a     z
1 1   one
2 2   two
3 3 three
4 4  four
```

`cbind()` and `rbind()` will also coerce vectors and matrixes of the proper lengths into a data frame, so long as one of the arguments of the bind function is a data frame. We could have used `rbind(x, c(4, "four"))` to take the data frame `x` we defined earlier, and coerce the vector `c(4, "four")` to fit into the existing data frame. But coercion can affect the way your data frame stores your data. In this case, the vector `c(4, "four")` would have coerced the integer `4` into the character `"4"`. Then the data frame would have coerced the entire first column into a character vector. This makes it safer to use `rbind()` and `cbind()` to bind data frames with each other.

FUNCTIONS

Once the data you need is structured appropriately, R has many built-in functions for viewing, analyzing, and manipulating that data. Furthermore, R will let you define your own functions. We'll briefly go over many of R's helpful functions here. Remember that you can use the `?` operator or the `help()` function to learn more about these functions.

VIEWING DATA AND METADATA

Different data structures contain different metadata to help define the structure itself and how the data within it should act. These functions allow you to see your data and metadata in different ways. Remember, to view the actual data within a variable, input the variable name and nothing else; R will return the data contained in that variable in the format most fitting for its data structure type.

FUNCTION	DESCRIPTION
summary()	Returns a brief summary of the data based on the data structure and types of data. Will return minimum, first quartile, median, mean, third quartile, and maximum values for a numeric vector. Will return the names of factors along with the count of each level. Will produce tables for data structures and matrixes, summarizing each column as its own vector.
str()	Returns a brief overview of the data's structure, including atomic type and/or data structure type, dimensions, number of factor levels, and examples of the data within the structure (truncated for long data sets). As with `summary()`, the output will differ based on the data structure and data type(s).
dim()	Gets or sets the dimensions of the data structure.
levels()	Gets or sets the levels of an object (usually a factor).
length()	Returns the length of an object.
names()	Gets or sets the names of an object.
class()	Gets or sets the class (type of data structure) of an object.
attributes()	Returns relevant attributes of an object.
object.size()	Returns the size in bytes an object is taking in memory.
order()	Returns a vector of indexes in either ascending or descending order. By default, the first value returned would be the index of the argument with the lowest value.
rank()	Returns a vector that ranks, in order, each element against all other elements. In other words, if the first number of the returned vector is 3, this means that the first element of x is the third smallest of all the other elements of x.
head()	Returns the first elements of an object.
tail()	Returns the last elements of an object.

MANIPULATING DATA

FUNCTION	EXAMPLES	DESCRIPTION
seq()	`seq(x, y, by = z)`	Increments x by z until y is reached/ surpassed. `seq(0, 10, by = 5)` returns `0 5 10`.

FUNCTION	EXAMPLES	DESCRIPTION
seq()	`seq(x, y, length = z)`	Increments x by y-z/(z-1) (i.e. x to y exactly z times).
rep()	`rep(x, times = y)`	Repeats x for y times. x can be a vector or factor.
rep()	`rep(x, each = y)`	Repeats the first value of x for y times, then repeats the next value of x for y times, until all values of x have been repeated.
paste()	`paste(..., sep = " ", collapse = " ")`	Merges string arguments into a single string, separated by `" "` (one space). If character vectors are inserted as arguments, the `paste()` will operate on each element of the vectors, recycling values as needed. The collapse argument will merge character vectors into a single string.
t()	`t(x)`	Transposes rectangular/tabular object x (a matrix or data frame).
rbind()	`rbind(x, y)`	Combines tabular data by rows. Here, objects x and y must have the same number of columns.
cbind()	`cbind(x, y)`	Combines tabular data by columns. Here, objects x and y must have the same number of rows.
strsplit()	`strsplit(x, "regex")`	Splits a character vector x into a list of substring vectors based on a regular expression.
nchar()	`nchar(c(x, y))`	Counts the number of characters in each element of a character vector and returns a vector containing the count of each of those.
substr()	`substr(x, y, z)`	Returns a substring of character vector x, starting with position y and ending with position z.
sort()	`sort(x)`	Sorts the values of vector or factor x either in ascending or descending order based on the argument `decreasing =`. Ascending order is the default.

MATH FUNCTIONS

FUNCTION	DESCRIPTION
abs()	Returns the absolute value of an object.
ceiling()	Returns the smallest integer greater than a numeric value.
floor()	Returns the largest integer less than a numeric value.
trunc()	Truncates a numeric value to an integer, toward 0.
cos(), sin(), tan(), acos(), asin(), atan() atan2(), cospi(), sinpi(), tanpi()	Trigonometric functions, such as sine and cosine.
exp()	Computes the exponential function (i.e. the value of e raised to the power of the argument of the function).
log(), log10(), log2()	Computes the natural, common, or binary logarithm. The argument base = can be used to define the base of the log() function.
max(), min()	Returns the maximum or minimum values of its arguments.
range()	Returns a vector with the minimum and maximum values of its arguments.
cummax(), cummin(), cumprod(), cumsum()	Returns the cumulative maximum or minimum values, or the cumulative product or sum, of its arguments.
mean()	Returns the mean value of its arguments.
median()	Returns the median value of its arguments.
cor()	Returns the level of correlation of a matrix or data frame, or two vectors. Uses the Pearson method by default, but other methods can be set with the `method =` argument.
cov(), var()	Returns the variance/covariance of a matrix or data frame, two vectors, or for `var()` a single vector.
sd()	Returns the standard deviation of the values of its arguments.

```
> x <- (sample(-25:25, 5)) / 3
> x
[1]  5.666667  7.666667  2.666667 -2.333333 -7.666667
> abs(x)
[1] 5.666667 7.666667 2.666667 2.333333 7.666667
> ceiling(x)
[1]  6  8  3 -2 -7
> floor(x)
[1]  5  7  2 -3 -8
> trunc(x)
[1]  5  7  2 -2 -7
> max(x)
[1] 7.666667
> min(x)
[1] -7.666667
> range(x)
[1] -7.666667  7.666667
> cummax(x)
[1] 5.666667 7.666667 7.666667 7.666667 7.666667
> cummin(x)
[1]  5.666667  5.666667  2.666667 -2.333333 -7.666667
> mean(x)
[1] 1.2
> median(x)
[1] 2.666667
> var(x)
[1] 38.75556
> sd(x)
[1] 6.225396
```

STATISTICAL FUNCTIONS

FUNCTION	DESCRIPTION
fitted()	Returns model fitted value from the argument.
predict()	Returns prediction values based on model fitted values.
resid()	Returns extracted residuals from the argument.

FUNCTION	DESCRIPTION
`lm()`	Fits a linear model based on a function given as an argument.
`glm()`	Fits a general linear model based on a function given as an argument.
`deviance()`	Returns the value of deviance from a fitted model object.
`coef()`	Returns coefficients from objects returned by modeling functions.
`confint()`	Returns confidence intervals of a fitted model based on parameters given as articles.
`vcov()`	Returns an estimated covariance matrix from a fitted model object.

D-P-Q-R STATISTICAL DISTRIBUTIONS

The following functions calculate distributions in your data. Each of these functions must be prefixed with either `d`, `p`, `q`, or `r`, which will signify whether the distribution is generated based on density, the distribution function, the quantile function, or random generation. For example, where the table here says `_binom()`, you must append either `d`, `p`, `q`, or `r` to the beginning of the function. The actual function would look like `rbinom()` and **not** `binom()`. Each of these functions have different accepted arguments and outputs. You can use `?dbinom` to find information on all the Binomial Distribution functions.

FUNCTION	DESCRIPTION
`_norm()`	Normal distribution.
`_binom()`	Binomial distribution.
`_pois()`	Poisson distribution.
`_exp()`	Exponential distribution.
`_chisq()`	Chi-Squared distribution.
`_gamma()`	Gamma distribution.
`_unif()`	Unified distribution.

CREATE YOUR OWN FUNCTION

There are many built-in functions in R, and many we could not even list here. If you are unable to find a function you need, though, R allows you to create your own function using the `function()` function. Functions can be created in-console, but often more complex functions are easier to write as .R scripts, which you can run, copy, or alter as you need.

Creating a function in R is much like creating a variable. Give the name you want to use for the function, then use the `<-` operator. After the operator, you will use `function(...)`, where the `...` represents the argument or arguments that will be provided to the function. You will then define the function using `{}` curly braces. See the example below:

```
> quad <- function(x) {
+   x * 4
+ }
> quad(1:3)
[1]  4  8 12
```

Here I have defined a function `quad()`. I have required only one argument for that function, and I have called the argument `x`. I used a curly bracer to begin defining what the function does. (Note the `+` plus signs on the left of the console; after hitting return after my first line, R noticed I was not finished writing my function, and used the plus sign to indicate it was waiting for more input.) On the next line, I define what the function does: in this case, I have the function `quad()` multiply the argument by four, or quadruple it. On the next line, I finish defining the function by closing the curly bracers `}`. This could have all been done on a single line; I've broken it up for readability.

You can also set default values for the arguments passed to your function. To do so, name the function, then type `=` and then the default value. Giving an argument a default value makes that argument optional.

```
> addminus1 <- function(x, y = 1) {
+   x + y - 1
+ }
> addminus1(3)
[1] 3
> addminus1(3,4)
[1] 6
```

If needed, functions can also be used as arguments in other functions. These functions can be predefined or inserted directly (an anonymous function).

OTHER USEFUL OPERATORS AND FUNCTIONS

LOGICAL OPERATORS

OPERATOR	DESCRIPTION
`<`	Less than operator.
`<=`	Less than or equal to operator.
`>`	Greater than operator.
`>=`	Greater than or equal to operator.
`==`	Exactly equals operator.
`!=`	Not equal to operator.
`|`	OR operator.
`||`	OR operator that evaluates the leftmost element of a vector.
`&`	AND operator (evaluated before OR operators).
`&&`	AND operator that evaluates the leftmost element of a vector.
`!`	NOT operator.

LOGICAL FUNCTIONS

FUNCTION	EXAMPLES	DESCRIPTION
isTRUE()	isTRUE(x)	Returns TRUE if argument x is TRUE.
xor()	xor(x, y)	Returns TRUE if either argument x OR argument y is TRUE, but NOT if both are TRUE or both are FALSE (exclusive OR logical).
which()	which(x)	Returns the indexes of logical vector x that are TRUE.
any()	any(x)	Returns TRUE if any element of logical vector x is TRUE.
all()	all(x)	Returns TRUE if all elements of logical vector x are TRUE.

INDEX VECTORS

There are several ways to isolate certain pieces of data from larger data sets. Index vectors are one way you can do this. There are four types of index vector, and each is accessed by placing square brackets `[]` directly next to the name of the data structure you want to access.

TYPE	EXAMPLES	DESCRIPTION
Logical	x[x > 0]	Checks each element of x and returns only elements that return TRUE for the logical indexed expression.
Positive Integer	x[y]	Returns a subset of x including only elements at positions included in y.
Negative Integer	x[-y]	Returns a subset of x including all elements of x except for elements at positions included in y.
Named	x["y"]	Returns a subset of x including only elements named y.

```
> x <- 1:5
> names(x) <- c("one", "two", "three", "four", "five")
> x[x > 3]
four five
   4    5
> x[2]
two
  2
> x[-c(1,4)]
  two three  five
    2     3     5
> x["five"]
five
   5
```

FILES AND DIRECTORIES

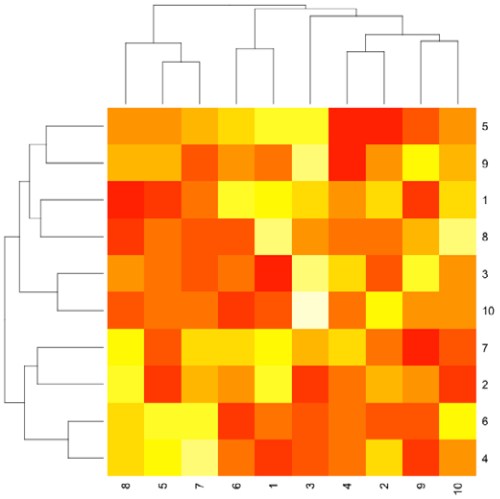
FUNCTION	DESCRIPTION
read.csv()	Reads the contents of the specified .csv file.
write.csv()	Writes to the contents of the specified .csv file.
getwd()	Gives the current working directory.
ls()	Returns the objects in your local workspace.
list.files() or dir()	Returns the files in your working directory.
dir.create()	Creates a directory within the working directory.
setwd()	Set a new working directory.
file.create()	Create a new file in the current working directory.
file.exists()	Checks whether a file exists in the working directory.
file.info()	Returns information about a file. Includes size, whether file is a directory, permissions in octal notation, modified time, created time, accessed time, user ID, group ID, username & group name.
file.rename()	Renames a file.
file.remove()	Deletes a file.
file.copy()	Makes a copy of a file in the working directory.
unlink()	Deletes a directory. Use the argument `recursive = TRUE` to delete a directory with contents (this will delete the contents).

GRAPHICAL FUNCTIONS

R is known for its extensive, easy-to-use graphical functions. Here are a few to get you started. Packages gplot and ggplot2 can help you create even more customized graphics. These are just a few basic graphical functions you can use in R. While for the sake of length, we can't go over all the graphs you can create here, or all the arguments you can use to customize them, you should get a sense of what kind of graphs you can create in R.

FUNCTION	DESCRIPTION
plot()	Creates a scatter plot; varies based on the data structure(s) provided.
hist()	Creates a histogram from a numeric vector.
dotchart()	Creates a dot chart based on a numeric vector and its labels.
barplot()	Creates a bar graph from a vector or matrix.
boxplot()	Creates a box plot from a formula and a data frame.
heatmap()	Creates a heat map from a matrix.
lines()	Can add lines to an existing graph or chart.

Here's a very basic example of a heatmap from function `heatmap( matrix(sample(1:100, 100, replace = TRUE), ncol = 10))`:



PACKAGE	DESCRIPTION
`swirl`	A package for R that gives walkthrough tutorials in the R console. It's a great hands-on way to get to know R.
`ggplot2`	Brings to R more graphical capabilities, allowing for the creation of more complex and more configurable graphs.
`RColorBrewer`	Contains built-in color palettes for better looking and easier to read graphics.
`data.table`	Enhances the abilities of data frames and allows for faster processing on large data sets.
`plyr`	Simplifies the process of performing split-apply-combine operations.

USEFUL RESOURCES

There's a lot more you can learn about and do with R than we can cover in this Refcard. But there are a lot of resources out there to help you learn more, and there are also a lot of R packages that can make R even more powerful. Try looking into these resources and packages to step up your R game. Use the `install.packages()` function to download a package (just put the package name in quotation marks as the function's argument). You'll need to load packages on new R sessions using the `library()` function, or by using your IDE (in RStudio, you can select checkboxes in the packages tab to load other installed packages).

RESOURCE	DESCRIPTION
r-bloggers.com	A site dedicated to rounding up blog posts from expert data scientists and R programmers.
datacamp.com	This site has some in-browser tutorials that can help you dig deeper into R.
inside-r.org	Has news, how-tos, daily features, and more handy R info.
Big Data Machine Learning	This DZone Refcard shows patterns for machine learning using R examples: bit.ly/machinelearningR.

ABOUT THE AUTHOR

G. RYAN SPAIN is a Publications Editor at DZone. He lives in Raleigh, North Carolina. He received his MFA in Poetry in 2014 and loves mixing his passions of poetry, science, and technology. When he's not producing DZone Refcardz, he enjoys writing poetry, programming with Java, querying with SQL, and analyzing data in R.

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513
888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795

VERSION 1.0