R Programming Basics

A Reading Companion

Jaewon ("Jay-one") Yoo
National Tsing Hua University

Programming for Business Analytics | Fall 2026

How to use this companion

This document adds the explanations that would normally accompany the PBA R Basics reference slides. It follows their topics and uses their examples, so that you can read a concept, run its code, and check what the result means. Page references point to the *physical PDF pages* of the 54-page reference deck, including its section pages.

Before the Week 2 visualization class, read ModernDive Chapter 1, Sections 1.2–1.4, and try its examples. This is the foundational review; the Week 2 schedule also assigns ModernDive Sections 2.1–2.7 for visualization. Chapter 1 introduces objects, functions and arguments, packages, and data frames. Use this companion to work through anything you find unclear. A useful route through it is Sections 1, 2.1, 2.2, 2.4, 3.1, 4.1, and 4.5. The aim is to recognize what a short piece of code is doing and to bring specific questions to class.

The remaining sections are references for the semester. Logical filtering becomes especially useful when we choose observations in data wrangling. Matrices, arrays, and lists help explain how different R objects are organized. Writing functions, nesting, scope, and custom operators are available here when we need them; mastering those later sections is not a prerequisite for the first visualization class.

In class, we will use these foundations to answer questions with data: choosing variables, comparing groups, making graphs, and explaining what the results tell us. We will return to the underlying R concepts as we use them. Reading a definition is a first encounter; applying it repeatedly is how it becomes familiar. Questions about the basics are welcome during class and the scheduled practice period.

Reading code and output. Blue blocks contain code you can run. The output below each block appears in the same order as the commands. Do not type the output into R. The accompanying `PBA_R_Basics_Companion.R` script contains the examples. Most use base R; one uses the `tibble` package. Examples reuse names such as `x`, `y`, and `f`, so run a section from its beginning when you revisit it.

Contents

1 Reading R: objects, operators, and function calls

R Basics reference slides, PDF pages 5–12; function calls revisited on 44–45.

1.1 An object name lets us use a result again

An expression such as `2 + 3` produces a value. Assignment gives a name to a value, so that we can refer to it in later commands. In `x <- 2`, the arrow points from the value on the right to the name on the left. R evaluates the right-hand side first and associates the result with `x`.

```
x <- 2
y = 5
x <- y
x
3 -> z
a <- b <- 7
c(x, y, z, a, b)
```

Output, in command order:

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

The line `x <- y` makes `x` hold the current value of `y`, which is 5. It does not create an instruction to keep the two names synchronized whenever `y` changes later. The final output shows the values of `x`, `y`, `z`, `a`, and `b` in that order. Assignment itself usually does not print a value; typing the name `x` on a new line asks R to display it.

The console label `[1]` is not part of your data. It means that the first value on that line is element 1 of the printed vector. An R vector with a single value is still a vector. We will use longer vectors shortly.

You will also see `=` used for assignment, and the slides show `3 -> z`. We use `<-` consistently in our analysis scripts. The arrow makes saving a result easy to distinguish from supplying a named function argument, such as `mean(x, na.rm = TRUE)`. The operators `<<-` and `->>` have a different role involving environments; see Section 6.6 when needed.

1.2 Names and punctuation

R is case-sensitive: `data`, `Data`, and `DATA` are different names. Ordinary names can contain letters, digits, periods, and underscores. They must begin with a letter or a period that is not followed by a digit. A name cannot begin with a digit or underscore. Reserved words, such as `if` and `function`, have special meanings. Descriptive names such as `purchase.freq`, `price`, and `quantity` make a script easier to follow.

Quotation marks distinguish text from a name. The expression `data` asks for an existing object; `"data"` is the four-letter character string. If you receive `object 'x' not found`, first check the spelling and capitalization, then check whether you ran the command that created `x` in this R session.

Symbol	How to read it in a script
`#`	A comment. R ignores the rest of that line.
`( )`	Enclose function arguments, or control evaluation order.
`,`	Separate arguments; inside two-dimensional brackets, separate rows from columns.
`{ }`	Group several expressions into a function body or another block.
`[ ]`	Select part of an object. The result depends on the object's structure.
`$`	Extract a named component, often a column of a data frame.
`;`	Separate expressions on one line. A new line is usually clearer.

1.3 Arithmetic and precedence

The arithmetic operators `+`, `-`, `*`, and `/` add, subtract, multiply, and divide. Exponentiation uses `^`, so `2^3` is 8. The operator `%%` gives a remainder, while `%/%` gives the integer quotient. For positive integers, 7 divided by 3 gives quotient 2 and remainder 1.

```
1 + 2 * 3
(1 + 2) * 3
7 %% 3
7 %/% 3
```

Output, in command order:

```
[1] 7
[1] 9
[1] 1
[1] 2
```

Multiplication is evaluated before addition, so the first expression is $1 + (2 \times 3) = 7$. Parentheses change which calculation happens first. Use them when a formula's intended order would otherwise be hard to read. The sequence operator `:` also has its own precedence; Section 3.1 shows why this matters.

1.4 Comparisons ask questions and return logical values

The operators `==`, `!=`, `>`, `>=`, `<`, and `<=` ask whether two values are equal, unequal, greater, greater or equal, smaller, or smaller or equal. Their results are logical values: `TRUE` or `FALSE` for the known values below. Comparisons involving missing values return `NA` (see Section 2.5). A single `=` and a double `==` have different meanings: assignment or argument naming versus a comparison.

```
2 == 3
2 != 3
2 < 3
2 >= 3
"data" == "stats"
```

Output, in command order:

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

A comparison can involve a whole vector. Its result can then identify which observations to keep. Logical operators combine these questions, as in "spending exceeds 1,000 *and* visits exceed 10." We develop that example in Section 2.4.

1.5 A function call names an action and supplies its arguments

In `round(c(1.2, 3.9, 0.4))`, `round` is the function name. Parentheses contain the input. Here that input is itself produced by a function call: `c(...)` combines three numbers into a vector, and `round(...)` rounds those numbers. Evaluate the inner call first to see the calculation clearly.

Arguments may be matched by position or named explicitly. For example, `rep(1:3, 2)` repeats a sequence twice; `rep(x = 1:3, times = 2)` states the same request using the argument names. Defaults are values a function uses when you omit an optional argument. The help page, opened with `?rep`, describes the arguments, defaults, and examples. Section 5.2 later shows how to set defaults in a function you write.

Packages supply additional functions and sometimes datasets. Install a package on your computer once using `install.packages("ggplot2")`; attach it in each new session with `library(ggplot2)` when you want to use its exported names directly. `require()` can also attach a package, but differs in how it reports failure, so the two functions are not identical. Our scripts will show the form needed for the example.

The double colon in `gapminder::gapminder` means "the exported object named `gapminder` in the package named `gapminder`." The package must be installed, but it need not be attached first. The `gapminder` data package and `ggplot2` plotting package are separate packages. A function such as `tibble::tibble()` uses the same notation to select a function from a particular package.

2 Data types: what kind of value are we working with?

R Basics reference slides, PDF pages 14–21.

An object's *type* describes the kind of values it stores. Its *structure* describes how those values are organized. For example, numeric values can form a vector, a matrix, or one column of a data frame. The distinction matters because a calculation suitable for numbers may not make sense for text, and a selection appropriate to a vector differs from a selection of rows and columns.

2.1 Numeric values and integers

Ordinary numeric literals such as `5` are stored as double-precision numbers in R, even when the value has no decimal part. Appending `L`, as in `5L`, requests integer storage. This is a distinction between storage types, not a claim that an integer-valued observation must always be stored as an integer.

```
i <- 5L
is.integer(i)
class(i)
typeof(5)
class(4L * 2.8)
class(5L / 2L)
```

Output, in command order:

```
[1] TRUE
[1] "integer"
[1] "double"
[1] "numeric"
[1] "numeric"
```

Mixing an integer with a decimal number can produce a double. Division also produces a numeric result in this example. You can generally calculate with both types without manually converting them.

`class()` describes how R treats an object, while `typeof()` describes its internal storage type. These often agree for simple objects but are not interchangeable. A factor, for instance, has class `"factor"` and underlying integer codes; a data frame has class `"data.frame"` and is organized as a list of columns.

2.2 Character strings

A character string is text inside matching single or double quotation marks. Numbers inside quotation marks are also text: `"5"` and `5` are different kinds of values. That distinction often explains why imported data cannot yet be used in a calculation.

```
x <- "data"
class(x)
nchar(x)
paste("Hello", x)
paste0("Hello", x)
substr(x, 1, 2)
```

Output, in command order:

```
[1] "character"
[1] 4
[1] "Hello data"
[1] "Hellodata"
[1] "da"
```

`nchar(x)` counts characters in the string. `paste()` joins text with a space by default; `paste0()` joins it without that space. `substr(x, 1, 2)` takes the first two characters. The positions begin at 1, as do ordinary vector indices in R.

When a graph requires a label such as `"Life expectancy"`, the quotation marks provide literal text. When an argument refers to a column, the required syntax depends on the function. For example, `aes(x = gdpPercap)` in ggplot2 interprets `gdpPercap` as a column in the supplied

data. Replacing it with `"gdpPercap"` would supply a constant string, changing the meaning of the plot.

2.3 Factors represent categories

A factor stores categories using integer codes together with labels called *levels*. This lets an analysis treat category membership as category membership, rather than as an arbitrary numerical amount. Factors can also specify an intended display order in a graph.

```
gender <- factor(c("Male", "Female", "Male"))
levels(gender)
as.integer(gender)
ordered_gender <- factor(gender, ordered = TRUE)
is.ordered(ordered_gender)
```

Output, in command order:

```
[1] "Female" "Male"
[1] 2 1 2
[1] TRUE
```

In this example, the default level order is `Female`, then `Male`, and the stored codes are consequently 2, 1, and 2. The code 2 does not mean that "Male" is twice "Female." The numeric codes describe the encoding, not a quantity to average.

The slide also constructs `ordered_gender`. This is valid R syntax, but it imposes an ordering on the categories. There is no natural low-to-high order in these labels, so this is a syntax demonstration, not a reason to use an ordered factor for this variable. For a category that really has an order, supply meaningful levels explicitly, such as `c("Low", "Medium", "High")`.

Do not use `as.numeric()` on a factor when you intend to recover numeric values printed in its labels. It returns the internal codes. If the labels are genuinely numbers, convert the labels first, for example `as.numeric(as.character(x))`, and check the result.

2.4 Logical values and combined conditions

`TRUE` and `FALSE` are logical values. In arithmetic they behave as 1 and 0; in logical operations, a nonzero number behaves as true and zero as false. Use the full names `TRUE` and `FALSE`: the shorter `T` and `F` seen in some slides can be overwritten as ordinary names.

```
TRUE * 5
FALSE * 5
3 & 0
3 | 0
!0
3 & 1
3 | 2
!1
```

Output, in command order:

```
[1] 5
[1] 0
[1] FALSE
[1] TRUE
[1] TRUE
[1] TRUE
[1] TRUE
[1] FALSE
```

The operator `&` means element-by-element AND: both conditions must be true. The operator `|` means element-by-element OR: at least one must be true. The operator `!` reverses a logical value. Use parentheses around each condition to make a compound rule easy to read.

The longer forms `&&` and `||` are intended for a single logical decision, such as an `if` condition. They short-circuit: they evaluate the second condition only when it is needed to determine the answer. Use `&` and `|` to evaluate a rule for every row of a table. Current R expects length-one inputs to `&&` and `||`; do not use them to discard all but the first element of a longer vector.

Following the customer example one row at a time

Each row below is a customer. We define a VIP as someone who has spent more than 1,000 *and* visited more than 10 times. An at-risk customer has spent less than 400 *or* visited fewer than 5 times. These are illustrative business rules from the slides, not estimated definitions of loyalty or churn.

```
customers <- data.frame(
  id = 1:6,
  spend = c(500, 1200, 300, 750, 2000, 150),
  visits = c(12, 4, 30, 7, 20, 1)
)
customers$VIP <- (customers$spend > 1000) &
  (customers$visits > 10)
customers$AtRisk <- (customers$spend < 400) |
  (customers$visits < 5)
customers
```

Output, in command order:

```
  id spend visits   VIP AtRisk
1  1   500     12 FALSE  FALSE
2  2  1200      4 FALSE   TRUE
3  3   300     30 FALSE   TRUE
4  4   750      7 FALSE  FALSE
5  5  2000     20  TRUE  FALSE
6  6   150      1 FALSE   TRUE
```

The dollar sign extracts a column. Thus `customers$spend > 1000` produces six logical values, one per customer. The AND operation compares these with the six visit conditions. Only customer 5 satisfies both VIP conditions. Customer 2 spends a great deal but has only four visits, so the chosen at-risk rule flags that customer. The two labels answer different questions and are not automatically opposites.

Assignment to `customers$VIP` adds a column to the data frame. To check this calculation, inspect the spend and visits values for a row and work out whether its two conditions are true. This is the same logic we will use when selecting observations in data wrangling.

2.5 Missing values and data checks

`NA` means that a value is missing. It is not the number zero and it is not the string `"NA"`. An unknown value is not equal to another unknown value in the usual sense, so use `is.na()` to test for missingness.

```
NA == NA
is.na(c(50000, NA, 30000))
mean(c(50000, NA, 30000))
mean(c(50000, NA, 30000), na.rm = TRUE)
```

Output, in command order:

```
[1] NA
[1] FALSE  TRUE FALSE
[1] NA
[1] 40000
```

The default mean is missing because one input is unknown. The argument `na.rm = TRUE` explicitly asks for a mean using the observed values. That is a decision about which observations enter the calculation. It does not recover the missing income or explain why it is missing.

The slides combine missingness with checks for impossible or implausible ages. This example uses a *tibble*, a modern form of data frame. Install the `tibble` package once if needed. Writing `tibble::tibble()` makes the package supplying the function explicit; the original slides attach `tidyverse` first.

```
df <- tibble::tibble(
  age = c(25, -3, 45, 200, 30),
  income = c(50000, 20000, NA, 100000, 30000)
)
invalid <- (df$age < 0) | (df$age > 120) | is.na(df$income)
invalid
df[invalid, ]
```

Output, in command order:

```
[1] FALSE  TRUE  TRUE  TRUE FALSE
# A tibble: 3 × 2
    age income
  <dbl>  <dbl>
1    -3  20000
2    45     NA
3   200 100000
```

The vector `invalid` marks rows 2, 3, and 4. The expression `df[invalid, ]` keeps those rows and all columns. The blank after the comma means "all columns." Identifying a row for review does not automatically tell us whether it should be corrected, excluded, or retained.

2.6 Dates and times

Dates and date-times have special classes so that R can calculate with them. `Date` represents dates using a count of days relative to January 1, 1970. `POSIXct` represents date-times using seconds relative to the epoch, while `POSIXlt` stores date-time components such as the hour, day, and month.

```
date1 <- as.Date("2012-06-28")
class(date1)
as.numeric(date1)
date2 <- as.POSIXlt("2012-06-28 17:42", tz = "Asia/Taipei")
class(date2)
date2$hour
as.Date("2012-06-30") - date1
```

Output, in command order:

```
[1] "Date"
[1] 15519
[1] "POSIXlt" "POSIXt"
[1] 17
Time difference of 2 days
```

The integer 15,519 is a date's internal day count, not a year or a timestamp in seconds. The example makes the time zone explicit so that it has a clear interpretation. The original slides use your system's time zone and print `Sys.Date() - date1`; that result changes with the date on which you run it. Here a fixed second date makes the two-day difference reproducible. `Sys.time()` returns the current date-time. Later, `lubridate` will provide convenient functions for date and time work.

3 Vectors: create, calculate, select, and convert

R Basics reference slides, PDF pages 25–34.

3.1 Creating a vector

A vector is an ordered collection of values. A column such as customer spending is naturally represented by a vector: its first entry belongs to the first customer, its second entry to the second customer, and so on. In this section, "vector" means an *atomic vector*, whose elements share one storage type.

```
x <- c(1, 2, 4)
x
5:8
seq(5, 8)
rep(1:3, 2)
i <- 2
1:(i - 1)
1:i - 1
```

Output, in command order:

```
[1] 1 2 4
[1] 5 6 7 8
[1] 5 6 7 8
[1] 1 2 3 1 2 3
[1] 1
[1] 0 1
```

`c()` combines supplied values. A colon produces a sequence with steps of one, and `seq()` offers more control over a sequence, such as the step size or desired length. `rep(1:3, 2)` repeats the *whole sequence* twice. Its result is 1, 2, 3, 1, 2, 3; repeating each entry twice would be a different request, `rep(1:3, each = 2)`.

With `i <- 2`, the expression `1:(i - 1)` means `1:1`, giving just 1. By contrast, `1:i - 1` is

evaluated as `(1:i) - 1`, giving 0 and 1. Parentheses express the intended boundary of the sequence. When programming with potentially empty objects, `seq_along(x)` is safer than `1:length(x)`, because it correctly returns an empty sequence when `x` has length zero.

3.2 Arithmetic is usually element by element

The two vectors in the slides have three entries each. In ordinary vector arithmetic, R pairs the first entry of `x` with the first entry of `y`, then the second with the second, and so forth.

```
x <- c(1, 2, 4)
y <- c(5, 0, -1)
x + y
x - y
x * y
x / y
x^y
x %% y
x %/% y
x %*% y
```

Output, in command order:

```
[1] 6 2 3
[1] -4  2  5
[1]  5  0 -4
[1]  0.2  Inf -4.0
[1] 1.00 1.00 0.25
[1]   1 NaN   0
[1]   0 Inf  -4
     [,1]
[1,]    1
```

The second entry of `x / y` is `Inf` because it divides 2 by zero. The second remainder is `NaN`, meaning "not a number," because a remainder on division by zero is not defined. These are results to investigate, not ordinary finite observations.

The distinction between `*` and `%*%` matters. The first multiplies corresponding entries, returning three values. The second performs matrix multiplication. For these two vectors it returns the inner product $1 \times 5 + 2 \times 0 + 4 \times (-1) = 1$, stored as a one-by-one matrix. We will use the relevant operation when an analysis calls for it.

3.3 Comparisons produce a logical vector

```
x <- c(1, 2, 4)
y <- c(5, 0, -1)
x > y
x <= y
x == y
x != y
!y
x >= 2 | y < 3
x < 2 & y >= 3
x <- 1:100
sum(x > 50)
```

Output, in command order:

```
[1] FALSE  TRUE  TRUE
[1]  TRUE FALSE FALSE
[1] FALSE FALSE FALSE
[1] TRUE TRUE TRUE
[1] FALSE  TRUE FALSE
[1] FALSE  TRUE  TRUE
[1]  TRUE FALSE FALSE
[1] 50
```

The result of `x > y` tells us which paired comparisons are true. The expression `!y` first interprets nonzero numbers as true, then reverses them. In the last example, `x > 50` creates 100 logical values. There are 50 true entries, and `sum()` counts them because `TRUE` contributes 1 and `FALSE` contributes 0. Missing values would require a separate decision, as in Section 2.5.

3.4 Vectorized functions and recycling

Many R functions work directly on every element of a vector. For example, `sqrt()` returns a square root for each input and `round()` returns a rounded value for each input. This is called *vectorization*.

```
sqrt(1:3)
round(c(1.2, 3.9, 0.4))
y <- c(12, 5, 13)
y + 4
`+`(y, 4)
```

Output, in command order:

```
[1] 1.000000 1.414214 1.732051
[1] 1 4 0
[1] 16  9 17
[1] 16  9 17
```

The function-call form `` `+`(y, 4) `` gives the same result as `y + 4`: the arithmetic operator is a function too. The slides contrast this with a loop that works through one position at a time:

```
y <- c(12, 5, 13)
for (i in 1:length(y)) {
  print(y[i] + 4)
}
```

Output, in command order:

```
[1] 16
[1] 9
[1] 17
```

Here `i` takes the values 1, 2, and 3. On each pass, the braces enclose the command to execute. The vectorized expression is shorter because it asks for the entire calculation directly. Not every function is elementwise: `sum(y)`, for example, summarizes the whole vector into one value. Consult a function's help when its behavior is unfamiliar.

In `y + 4`, the one-element vector 4 is repeated to match the length of `y`. Base R also recycles longer short vectors. If the longer length is not a multiple of the shorter length, arithmetic issues a warning.

```
c(1, 2, 4) + c(6, 0, 9)
c(1, 2, 4) + c(6, 0, 9, 20, 22)
```

Output, in command order:

```
[1]  7  2 13
Warning: longer object length is not a multiple of shorter object length
[1]  7  2 13 21 24
```

In the second expression, `c(1, 2, 4)` is reused as 1, 2, 4, 1, 2 to match five entries. R returns an answer *and* warns. A warning does not mean that execution stopped; it means the result may differ from what you intended. Even exact recycling can be accidental, so check lengths and alignment rather than treating an absence of warnings as proof of correctness.

3.5 Selecting entries by their positions

```
y <- c(1.2, 3.9, 0.4, 0.12)
y[1]
y[2:3]
v <- 3:4
y[v]
z <- c(5, 12, 13)
z[-1]
z
```

Output, in command order:

```
[1] 1.2
[1] 3.9 0.4
[1] 0.40 0.12
[1] 12 13
[1]  5 12 13
```

Indices begin at 1. A positive index includes an entry; a vector of positive indices includes several entries in the requested order. A negative index omits an entry from the returned result. The expression `z[-1]` does not modify `z`; the final output confirms that the original still contains 5, 12, and 13. Saving the result with `z <- z[-1]` would replace it.

Do not combine positive and negative indices in one selection, such as `z[c(1, -2)]`. The request would mix "include these" and "omit these" conventions. Use one clear selection rule. Also distinguish `y[2:3]` from `y[2, 3]`: the comma requests two dimensions and is not appropriate

for an ordinary one-dimensional vector.

3.6 Selecting and replacing entries with a condition

```
z <- c(5, 2, -3, 8)
z[c(TRUE, FALSE, TRUE, TRUE)]
which(z * z > 8)
z[which(z * z > 8)]
w <- z[z * z > 8]
w
x <- c(1, 3, 8, 2)
x[x > 3] <- 0
x
```

Output, in command order:

```
[1]  5 -3  8
[1] 1 3 4
[1]  5 -3  8
[1]  5 -3  8
[1] 1 3 0 2
```

The condition `z * z > 8` is true for 5, -3, and 8. Logical indexing returns those values directly. `which()` instead returns their *positions*, 1, 3, and 4; indexing with those positions retrieves the same values here. Both examples select entries according to the condition, using different intermediate results.

Assignment to a subset changes just the selected entries. In `x[x > 3] <- 0`, only 8 satisfies the condition, so only that value becomes zero. This pattern will reappear when cleaning or transforming data.

There is a useful difference when conditions contain `NA`. Direct logical indexing can retain an unknown entry in the result; `which()` returns only positions that are definitely true. Decide how missing values should be treated instead of assuming the two expressions are interchangeable for every dataset.

3.7 Mixed values trigger type conversion

```
num.char.vec <- c(1, 3, "five", 7)
num.char.vec
num.vec <- c(1, 3, 5, 7)
comb.vec <- c(num.vec, num.char.vec)
comb.vec
typeof(comb.vec)
```

Output, in command order:

```
[1] "1"    "3"    "five" "7"
[1] "1"    "3"    "5"    "7"    "1"    "3"    "five" "7"
[1] "character"
```

An atomic vector cannot retain some numeric entries and some character entries as different storage types. Combining a character value with these numbers converts the numbers to character strings. The printed quotation marks make the result visible. Combining two vectors concatenates their entries; it does not pair them into rows or columns.

Explicit conversion functions state the intended type. They can also expose a problem in the original values.

```r
numbers.vec <- c(-3, -2, -1, 0, 1, 2, 3)
num2char <- as.character(numbers.vec)
num2char
as.logical(numbers.vec)
char.vec <- c("1", "3", "five", "7")
char2num <- as.numeric(char.vec)
char2num
```

Output, in command order:

```
[1] "-3" "-2" "-1" "0"  "1"  "2"  "3"
[1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
Warning: NAs introduced by coercion
[1]  1  3 NA  7
```

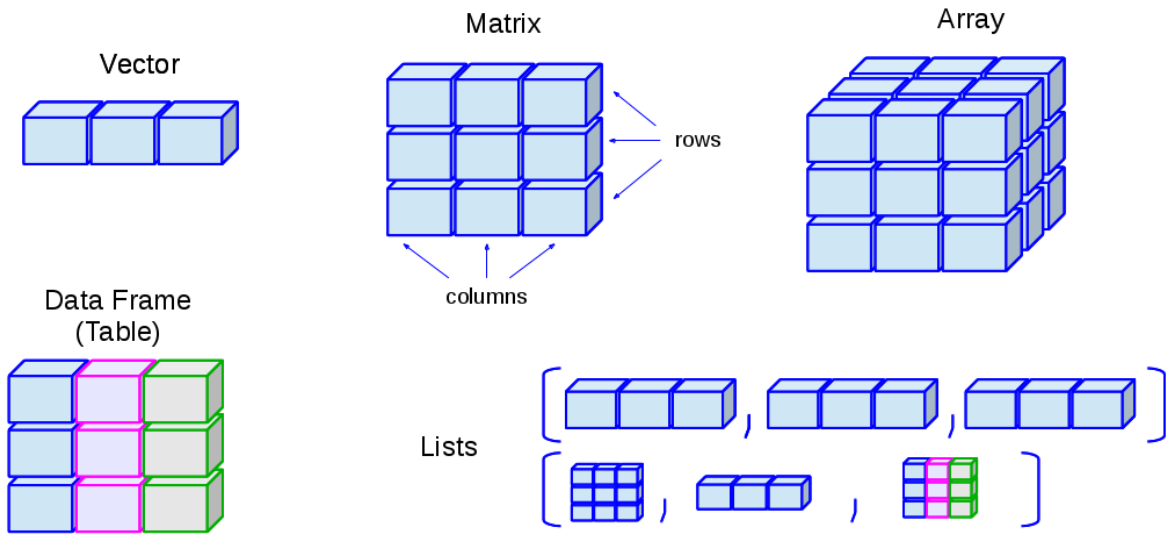
Only zero becomes `FALSE`; the other numbers become `TRUE`. In the final conversion, `"1"`, `"3"`, and `"7"` are recognizable numeric strings. The word `"five"` is not a numeric representation R can parse here, so it becomes `NA` and triggers a warning. Check converted values with `is.na()`, `class()`, or `str()` before using them in analysis.

4 From vectors to tables and other structures

R Basics reference slides, PDF pages 24; 35–41.

4.1 A map of the structures

The original diagram brings the structures together. A vector is one-dimensional. A matrix has rows and columns. An array adds dimensions, such as several matrices stored together. A list can contain components of different kinds and sizes. A data frame organizes columns into a rectangular table.



Original data-structure diagram from the PBA R Basics reference slides, PDF page 24.

Structure	Organization	Typical selection
Atomic vector	One sequence, one storage type	`x[2]`
Matrix	Two dimensions, usually one atomic type	`x[2, 3]`
Array	One or more dimensions, usually one atomic type	`x[2, 3, 1]`
List	Components may differ in type and size	`x[[2]]` or `x$name`
Data frame	Columns with the same number of rows; types may differ across columns	`x[2, ]` or `x$name`

The single-type description applies to the usual *atomic* matrices and arrays in this introduction. R can also make arrays whose underlying vector is a list. The practical distinction for our examples is that a numeric matrix holds numbers throughout, while a data frame can retain names in one column and numeric ages in another.

In analysis tables, each row represents an observation and each column a variable. What counts as an observation depends on the data: a customer in the example above, or a country in a particular year in Gapminder. Identifying the row's meaning is part of understanding the data, not something R can decide for us.

4.2 Matrices: rows, columns, and filling order

R Basics reference slides, PDF pages 36.

The function `matrix()` arranges a vector into rows and columns. By default it fills down a column before moving to the next column. Supplying `byrow = TRUE` fills across a row instead.

```
y <- matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)
y
z <- matrix(c(1, 2, 3, 4), nrow = 2, byrow = TRUE)
z
z[1, 2]
z[1, ]
z[, 2]
z[, 2, drop = FALSE]
```

Output, in command order:

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

In `z[1, 2]`, the first index chooses row 1 and the second chooses column 2. Leaving an index blank selects all entries on that dimension. Selecting a single column ordinarily simplifies the result to a vector; `drop = FALSE` keeps its matrix shape. This matters when a later function expects a matrix rather than a vector.

```
matrix(c(1, 2, 3, 4), nrow = 3)
```

Output, in command order:

```
Warning: data length [4] is not a sub-multiple or multiple of the number of rows [3]
     [,1] [,2]
[1,]    1    4
[2,]    2    1
[3,]    3    2
```

Four input values cannot fill a rectangular matrix of three rows without reuse or unused space. R infers two columns and recycles values to fill six positions, issuing a warning. Specify dimensions consistent with the data and inspect `dim(x)` when the shape is not what you expected.

4.3 Arrays: the same indexing idea with another dimension

The reference diagram includes arrays. This extension of its small-number matrix example shows how the third dimension works. An array with dimensions `c(2, 2, 2)` contains two rows, two columns, and two slices, for eight entries in total.

```
a <- array(1:8, dim = c(2, 2, 2))
a
a[1, 2, 2]
a[, , 2]
```

Output, in command order:

```
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8

[1] 7
     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

The first dimension varies fastest as R fills the array, so the first slice is filled before the second. The expression `a[1, 2, 2]` selects row 1, column 2, slice 2, giving 7. The expression `a[, , 2]` retrieves the entire second slice. The commas keep the dimension positions clear. As with matrices, `drop = FALSE` preserves dimensions when simplification would otherwise remove them.

4.4 Lists contain components, not necessarily equal-length columns

R Basics reference slides, PDF pages 37–38.

The employee example stores a name, salary, and union-membership indicator. A list retains these different types without converting them all to strings.

```
j <- list(name = "Joe", salary = 55000, union = TRUE)
j
jalt <- list("Joe", 55000, TRUE)
names(jalt)
z <- vector(mode = "list")
length(z)
```

Output, in command order:

```
$name
[1] "Joe"

$salary
[1] 55000

$union
[1] TRUE

NULL
[1] 0
```

The first list names its three components. The second has the same values without component names, so `names(jalt)` is `NULL`, meaning there is no names attribute. `vector(mode = "list")` creates an empty list. A list component can itself be a vector, matrix, data frame, or another list, as illustrated by the diagram.

```
j$salary
j[["salary"]]
j[[2]]
class(j[[2]])
j["salary"]
class(j[2])
```

Output, in command order:

```
[1] 55000
[1] 55000
[1] 55000
[1] "numeric"
$salary
[1] 55000

[1] "list"
```

Double brackets extract *one component's contents.* The dollar sign selects a component by its name. Here all three forms return the numeric salary 55,000. Single brackets retain the list container: `j["salary"]` is a one-component list whose component holds 55,000. Its class is consequently `"list"`.

This is why the indexing operators cannot be described only as alternative ways to print a value. The result's structure can change. For a list, choose `[ ]` when you want a sub-list and `[[ ]]` when you want one component's contents. Ordinary atomic-vector indexing with `[ ]` returns an atomic vector, not a list.

4.5 Data frames: columns together, observations across rows

R Basics reference slides, PDF pages 35; 39.

A data frame is a rectangular collection of columns. Different columns may hold different types, but all columns describe the same set of rows. In the slides' example, the first row concerns Jack and the second concerns Jill.

```
kids <- c("Jack", "Jill")
ages <- c(2, 10)
data <- data.frame(kids, ages, stringsAsFactors = FALSE)
data
data[[1]]
data$kids
data[, 1]
```

Output, in command order:

```
  kids ages
1 Jack    2
2 Jill   10
[1] "Jack" "Jill"
[1] "Jack" "Jill"
[1] "Jack" "Jill"
```

The character vector `kids` and numeric vector `ages` become two columns. `stringsAsFactors = FALSE` makes the intended treatment of text explicit. It is already the default in current R, though older R versions behaved differently.

A data frame supports both list-like column extraction and matrix-like row-and-column indexing. `data[[1]]`, `data$kids`, and `data[, 1]` all return the character column here. Single-bracket column selection without a comma, `data["kids"]`, retains a one-column data frame.

```
data[2, ]
data[, "kids", drop = FALSE]
data["kids"]
data[data$ages > 3, ]
str(data)
```

Output, in command order:

```
  kids ages
2 Jill   10
  kids
1 Jack
2 Jill
  kids
1 Jack
2 Jill
  kids ages
2 Jill   10
'data.frame':   2 obs. of  2 variables:
 $ kids: chr  "Jack" "Jill"
 $ ages: num  2 10
```

The expression `data[2, ]` selects row 2 and every column. The condition in `data[data$ages > 3, ]` keeps Jill. The result from `str()` reports two observations and two variables, then shows each column's type and initial values. This is often more informative than looking only at the printed table.

Tibbles follow the same general table idea but have some stricter subsetting and printing behavior. For example, selecting one column with `[ , ]` ordinarily keeps a tibble rather than simplifying to a vector. Use `$` or `[[ ]]` when you specifically want a column's contents. Before interpreting a printed number as a dataset's size, check whether the output is displaying only a preview of a larger table.

4.6 Importing a table and checking what arrived

R Basics reference slides, PDF pages 40–41.

Most analysis begins with a file rather than manually typed values. The reference slides import a comma-separated file with columns `ID`, `Exam1`, `Exam2`, and `Quiz`. The original commands are:

```
exam1 <- read.csv(url("https://bit.ly/3ZqTQLY"))
exam1 <- read.table(url("https://bit.ly/3ZqTQLY"),
                    sep = ",", header = TRUE)
```

These are two ways to read the *same* CSV. The argument `sep = ","` identifies the separator and `header = TRUE` says that the first line contains column names. Assignment stores the imported table as `exam1`. A URL points to an external file, so successful execution also depends on that address remaining accessible. The script leaves these lines commented for use with the actual course file.

For a local file, the original slides show a full path on the instructor's computer. That path will not exist on yours. Use the location of your own downloaded file. Within an RStudio project containing `import_data.csv`, the corresponding command is:

```
exam1 <- read.csv("import_data.csv")
head(exam1)
str(exam1)
names(exam1)
dim(exam1)
```

`head()` displays the first rows, `str()` shows column types and structure, `names()` returns column names, and `dim()` returns the numbers of rows and columns. Reading a file successfully does not prove that its separators, missing values, dates, and numeric columns were interpreted as intended. These checks help catch problems before plotting or modeling.

`getwd()` reports R's working directory, from which a relative file path is resolved. `setwd()` changes it, as shown in the reference slides. An RStudio project provides a convenient consistent starting directory. The RStudio import menu is also useful; keep the generated import code in your script so that you can repeat the import in a later session.

5 Writing functions you can use again

R Basics reference slides, PDF pages 42–45.

A function collects an operation under a name. It can accept different inputs each time it is called, while keeping the calculation in one place. You have already used functions supplied by R and packages. Writing one yourself uses the same idea.

5.1 Inputs, body, and returned value

The temperature-conversion example starts with the formula $C = (F - 32)/1.8$. In `function(x)`, `x` names the input. The expression following it is the body. Assigning the function to `f` lets us call it with `f(100)`.

```
f <- function(x) (x - 32) / 1.8
f(100)
f <- function(x) { (x - 32) / 1.8 }
f(100)
f <- function(x) return((x - 32) / 1.8)
f(100)
f <- function(x) x * 1.8 + 32
f(37.7778)
```

Output, in command order:

```
[1] 37.77778
[1] 37.77778
[1] 37.77778
[1] 100
```

The first three definitions are equivalent ways to return the Celsius conversion. Braces let a body contain several expressions. Without an explicit `return()`, a function returns the value of its last evaluated expression. An explicit `return()` states what to return and can end the function before the end of the body.

The last definition replaces `f` with the inverse conversion, from Celsius to Fahrenheit. The old definition is no longer what `f` names. When keeping both functions in an analysis, distinct function names would make the direction unambiguous. For example, use `fahrenheit_to_celsius` for the first conversion and `celsius_to_fahrenheit` for its inverse. Here the reused name follows the slides and illustrates that functions are objects too.

Since the arithmetic in the conversion is vectorized, this function can take a numeric vector and return a converted vector. That behavior comes from its body; defining something as a function does not automatically make every possible body work on vectors.

5.2 Defaults and argument matching

The next function displays its arguments. Argument `a` has no default, while `b`, `c`, and `d` do. The added newline `"\n"` simply keeps each displayed result on its own line.

```
f <- function(a, b = 1, c = 2, d = NULL) {
  cat(a, b, c, d, "\n")
}
f(1)
f(1, 2, 3, 4)
f(1, 2, 3)
f(3, c = 5)
f(d = 7)
```

Output, in command order:

```
1 1 2
1 2 3 4
1 2 3
3 1 5
Error: argument "a" is missing, with no default
```

Calling `f(1)` supplies `a` and uses the remaining defaults. The value `NULL` contributes no text to `cat()`. Calling `f(3, c = 5)` supplies `a` by position and `c` by name, leaving `b` and `d` at their defaults. Naming an argument lets you change it without filling every preceding optional position.

The final call deliberately fails: it supplies `d` but omits required `a`. The error identifies which argument is missing. The companion script wraps that demonstration in `try()` so that you can

continue running subsequent examples.

`cat()` is for displaying text; it does not return the displayed numbers as a numeric vector. If you want to save a function's result for later calculations, return the desired object, then assign the function call to a name. Printing a result and returning a useful object are separate actions.

6 Later reference: nesting, scope, and custom operators

R Basics reference slides, PDF pages 46–54.

This section explains the remaining reference slides. Use it when you begin reading or writing more involved functions. The first visualization class does not assume that you have mastered these features.

6.1 A function can define and call another function

```
f <- function(x, y) {
  print(x)
  g <- function(y) { print(y) }
  g(y)
}
f(1, 2)
args(f)
```

Output, in command order:

```
[1] 1
[1] 2
function (x, y)
NULL
```

Calling `f(1, 2)` first prints `x`, then creates a local function `g`, then calls `g(y)` to print the second value. `g` is created inside this call to `f`. This organization can be useful when a helper operation belongs specifically to a larger function. `args(f)` shows its argument list, and `body(f)` shows the code in its body.

6.2 Passing arguments through with three dots

```
f <- function(x, y) {
  print(x)
  print(y)
}
g <- function(z, ...) {
  print(z)
  f(...)
}
g(1, 2, 3)
```

Output, in command order:

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

The first value in `g(1, 2, 3)` fills `z`, so `g` prints 1. The remaining values are collected by `...` and passed to `f(...)`. There they fill `x` and `y`, so `f` prints 2 and 3. Three dots allow a wrapper function to pass arguments onward without naming all of them in its own definition. They do

not make arbitrary arguments valid: the receiving function must still know what to do with them.

6.3 Scope: where R finds a name

Each function call has a local environment containing its arguments and locally created objects. When a name is not found there, R searches enclosing environments. For a function defined at the console, this commonly leads to the global environment. More generally, R uses the environment in which the function was *defined*, which need not be the environment from which it was called. This is lexical scoping.

An object such as `global_var <- "I am global"`, created at the console, is ordinarily available to functions defined there. An object created inside a function is local to that call unless you explicitly return it or arrange for it to be stored elsewhere. The total example shows why this matters:

```
total <- 0
add_to_total <- function(value) {
  total <- total + value
  print(paste("Total is now:", total))
}
add_to_total(10)
add_to_total(5)
print(total)
```

Output, in command order:

```
[1] "Total is now: 10"
[1] "Total is now: 5"
[1] 0
```

On the right-hand side of `total <- total + value`, R has not yet created a local `total`, so it finds the outer value 0. Assignment then creates a *local* `total`. When the function is called again, it gets a fresh local environment and once again starts from the outer zero. That is why the displayed totals are 10 and 5 rather than 10 and 15, while the outer `total` remains zero.

For ordinary analysis, an explicit input and output are often easier to follow: define `add <- function(total, value) total + value`, then write `total <- add(total, 10)`. The assignment at the calling level makes the state change visible.

6.4 Outer values, local values, and arguments

```
n <- 1
f <- function() print(n)
f()
n <- 2
f()
n <- 100
f <- function() { n <- 1; print(n) }
f()
n
```

Output, in command order:

```
[1] 1
[1] 2
[1] 1
[1] 100
```

The first definition finds `n` outside the function, so changing that outer value changes what the next call prints. The second definition creates a local `n = 1`; it takes precedence while that function is executing. Printing the outer `n` afterward shows that it is still 100. This is often called *shadowing*.

```
f <- function() x_local <- 1
f()
exists("x_local")
n <- 100
f <- function(n) print(n)
f(1)
n
```

Output, in command order:

```
[1] FALSE
[1] 1
[1] 100
```

The reference slides demonstrate local visibility by clearing the entire workspace, defining `x` inside a function, and then trying to access `x` outside it. This companion uses the distinct name `x_local` and `exists()` instead, so the demonstration does not remove your other objects. The function's assignment creates no outer `x_local`. The `FALSE` result checks that fact.

Arguments are local bindings as well. In `f(1)`, the argument `n` has value 1 within the function even though the outer `n` is 100. This is another instance of the same lookup rule, rather than an unrelated exception.

6.5 Nested lookup follows the enclosing environments

```
f <- function(x) {
  a <- 2
  g <- function(y) { print(y + a) }
  g(x)
}
a <- 100
f(1)
```

Output, in command order:

```
[1] 3
```

Within `g`, `y` is the argument supplied by `g(x)`. The name `a` is absent from `g`'s local environment, so R finds `a = 2` in the enclosing call to `f`. It therefore prints $1 + 2 = 3$, rather than using the outer `a = 100`. If the assignment `a <- 2` were removed, lookup would continue outward and the same call would print 101.

6.6 Super-assignment: assigning in an enclosing environment

The operator `<<-` assigns a value to a named variable in an enclosing environment. To find where to assign it, R searches outward through the enclosing environments and updates the first existing binding for that name. It skips the current function-call environment. If no binding is found, it assigns in the global environment. The rightward form `->>` performs the same assignment with the value written on the left and the name on the right.

```
b <- 0
f <- function() {
  a <- 1
  g <- function(a, b) {
    a <<- 2
    b <<- 2
    print(a)
    print(b)
  }
  g(3, 3)
  print(a)
  print(b)
}
f()
```

Output, in command order:

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

The arguments of `g(3, 3)` remain 3 and 3. The assignment `a <<- 2` changes `a` in the surrounding call to `f`. The assignment `b <<- 2` finds `b` further out and changes it there. Consequently, `g` prints 3 and 3, while the subsequent commands in `f` print 2 and 2.

If both operators were replaced with ordinary `<-`, the assignments would change the local `a` and `b` inside `g`. The four printed values would instead be 2, 2, 1, and 0. Super-assignment is useful for specialized programming, but its changes are less visible at the calling site. Returning a value and explicitly assigning it is usually clearer for the analysis code we write in PBA.

6.7 Writing a binary operator

```
"%a2b%" <- function(a, b) return(a + 2 * b)
3 %a2b% 5
"%sdf%" <- function(a, b) {
  sdfxy <- setdiff(a, b)
  sdfyx <- setdiff(b, a)
  return(union(sdfxy, sdfyx))
}
x <- c(1, 2, 5)
y <- c(5, 1, 8, 9)
x %sdf% y
```

Output, in command order:

```
[1] 13
[1] 2 8 9
```

The custom operator `%a2b%` takes an input on either side and returns the first plus twice the second: $3 + 2 \times 5 = 13$. Percent signs mark this as a user-defined infix operator. The function still receives two ordinary arguments.

The second operator calculates the symmetric difference of two sets. `setdiff(a, b)` returns values in `a` but not `b`; reversing the arguments finds values unique to `b`. `union()` combines the two results without duplicates. The shared values 1 and 5 disappear, leaving 2, 8, and 9. These are set operations, so repeated copies of the same value would not be counted separately.

Finding the right reference when a question comes up

The undergraduate PBA course website and graduate PBA course website provide the current weekly readings and lecture materials. Use ModernDive, Chapter 1 for the introductory preparation, and this companion for explanations tied to the original R Basics slides. Later reading requirements are listed by week on the course website.

For a function's exact behavior in your installed R version, open its help page, such as `?matrix`, `?factor`, or `?read.csv`. The freely available R Core manual, *An Introduction to R*, provides further reference: Chapter 2 for vectors and indexing, Chapter 4 for factors, Chapters 5–6 for structures, Chapter 7 for importing data, and Chapter 10 for writing functions. These are places to look up a specific question, not an additional requirement to read the whole manual before Week 2.

When asking for help, bring the expression you ran, the exact output or error, and what you expected instead. If it uses an object created earlier, include the lines that created that object. This lets us work through the point where the code and your intended calculation diverge.