R Basics

Contents

Introduction

This document contains brief descriptions of core R functionality. Examples are provided as code blocks. Code block results are printed below the code block, prefixed by `##`.

```
"This is a code block. Code blocks use different fonts and have light gray backgrounds.
They contain examples of valid R code."
```

```
## The results of running a code block are shown underneath each block.
```

As you read through this document, follow along by typing the code block examples into an R console and running them yourself. Check that you get the same output as displayed in this document.

You will also find a series of exercises at the end of the document.

Basic R syntax

Math operations

Basic arithmetic

```
(17 * 0.35) ^ 1/3
```

```
## [1] 1.983333
```

Logarithms and exponentiation use the `log` and `exp` functions. The default base is `e`, with a value of approximately 2.71828.

```
log(2.71828)
```

```
## [1] 0.9999993
```

```
exp(1)
```

```
## [1] 2.718282
```

Specify the base of your logarithm with the `base` argument. Common logarithm bases have their own functions.

```
log(10, base=10)
```

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

```
log10(10)
```

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

```
log2(10)
```

```
## [1] 3.321928
```

Logical operators

Logical operators are used for value and/or variable comparisons.

- Equivalent: `==`
- Not equivalent: `!=`
- Less than: `<`
- Greater than: `>`
- Or: `|`
- And: `&`

```
g = 12
h = 13
g == h
```

```
## [1] FALSE
```

Variables

There are a few important properties of R variables:

1. Variables are given values by the process of variable assignment
2. Variables have different types
3. Variables can be reused
4. Variables are mutable; their values can be modified or changed

Variable assignment

Assignment by convention uses the `<-` operator, with the variable name on the left and its assigned value on the right.

```
a <- 100
print(a)
```

```
## [1] 100
```

Variable types

Numeric variables are integers and decimal values (also known as type "double", for double-precision floating point value). *For most of your work, this distinction is irrelevant,* but sometimes it does impact calculations so it's good to be aware this distinction exists.

The `typeof` function can be used to report the type of a variable. Recall variable `a` assigned above.

```
typeof(a)
```

```
## [1] "double"
```

Even though `a` was assigned the value of "100" without a decimal, it is by default treated as a variable that can have decimals. If you need an integer data type, you can ask for one explicitly:

```
b <- as.integer(100)
typeof(b)
```

```
## [1] "integer"
```

Character strings are text variables. They must be enclosed in either single or double quotes.

```
d <- 'Hello world!'
typeof(d)
```

```
## [1] "character"
```

Logical variables are `TRUE` or `FALSE`.

```
e <- TRUE
typeof(e)
```

```
## [1] "logical"
```

```
f <- FALSE
typeof(f)
```

```
## [1] "logical"
```

Missing values

Variables of each data type (numeric, character and logical) can also take the value of `NA`: "not available". - `NA` is not the same as 0 - `NA` is not the same as " " - `NA` is not the same as FALSE

Any operations (calculations or comparisons) that involve `NA` may or may not produce `NA`.

Variable reassignment and modification

You can also assign variables the value of other variables. The code block below assigns variable `f` the value of `a` converted to an integer. This assignment does not change the value of `a`.

```
g <- as.integer(a)
print(typeof(g))
```

```
## [1] "integer"
```

```
print(typeof(a))
```

```
## [1] "double"
```

If we change the value of `a`, what happens to the value of `f`?

```
a <- 'Now a is a character string'
print(a)
```

```
## [1] "Now a is a character string"
```

```
print(g)
```

```
## [1] 100
```

The value of `f` remains unchanged because in R, variable assignment copies values rather than referencing them. For more details, see examples here.

Scalar vs vector variables

So far we have considered only **scalar** variables – variables with a single value. Variables can also hold multiple variables in vectors.

We often use functions to create vector variables. The `c()` function combines values into vectors:

```
my_vector <- c(1, 1, 2, 3, 5, 8)
print(my_vector)
```

```
## [1] 1 1 2 3 5 8
```

The `seq()` function makes a sequence. Its arguments are start, stop, and step.

```
my_sequence <- seq(30, 20, -1)
print(my_sequence)
```

```
##  [1] 30 29 28 27 26 25 24 23 22 21 20
```

Getting help with functions

You've seen a few examples of functions so far: `print()`, `typeof()`, `c()` and `seq()`. There are many, many functions in R and its optional packages. For most, you can get access standardized documentations for functions with the `help()` function or `?` operator.

```
help("c")
```

```
?c
```

The help documentation has several sections of which these are typically the most informative.

1. "Description" is a basic description of the function.
2. "Usage" shows brief examples of the function call.
3. "Arguments" lists the arguments (the variables or data you pass to the function, upon which the function does work). This section will also explain the expected types for argument values and the argument default values (if any).
4. "Details" expands on the description a bit.
5. "Value" describes the value the function returns.
6. "Examples" shows usage examples in more detail.

Flow control

Flow control helps you move data through your script with looping and conditional statements.

Loops

For-loops

For-loops allow you to perform actions (like evaluate functions) on iterable variables (like vectors) one element at a time. For-loops terminate when all elements in the iterable have been acted on:

```
for(i in 1:10){
  print(i*i)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
## [1] 49
## [1] 64
## [1] 81
## [1] 100
```

While-loops

While-loops also work on iterables, but have terminate when a pre-specified condition is met.

```
i <- 1
while(i < 6){
  print(i)
  i <- i + 1
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

If you don't select a suitable termination condition, your while-loop could continue indefinitely. For example, this code would loop infinitely:

```
j <- 1
while(j < 6){
  print(j)
  j <- j - 1
}
```

If/Else conditionals

If/Else statements are used to evaluate mutually exclusive conditions:

```
bigger_than_breadbox <- FALSE
if(bigger_than_breadbox == TRUE){
  print('The item bigger than breadbox')
}else{
  print('The item is not bigger than a breadbox')
}
```

```
## [1] "The item is not bigger than a breadbox"
```

If/Else statements can be extended to more than two mutually exclusive conditions with an `else if` clause:

```
object <- 'plastic'

if(object == 'vegetable'){
  print('Object is a vegetable')
}else if(object == 'animal'){
  print('Object is an animal')
}else if(object == 'mineral'){
  print('Object is a mineral!')
}else{
  print('Object must be something else.')
}
```

```
## [1] "Object must be something else."
```

Basic data structure

Vectors

We briefly discussed vectors above, in contrast to scalar variables. Vectors are ordered collections of data of the same type. Elements in vectors are usually accessed by index. In R, indexing starts with "1".

Recall the `my_vector` variable defined above. Use square brackets to reference elements in the list by their index number:

```
print(my_vector)
```

```
## [1] 1 1 2 3 5 8
```

```
print(my_vector[1])
```

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

```
print(my_vector[4])
```

```
## [1] 3
```

We can perform mathematical operations on vectors. We can multiply each value in the vector by a scalar value:

```
my_vector*3
```

```
## [1]  3  3  6  9 15 24
```

We can also perform aggregations on the vector:

```
sum(my_vector)
```

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

Matrices

Matrices have a `[row, column]` data structure; like vectors, all elements in a matrix must have the same data type. Matrices can be used for linear algebra computations.

```
A = matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow=3)
A
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Extract single elements from a matrix with their `[row, column]` positional indexes.

```
A[2, 2]
```

```
## [1] 5
```

Slice out rows and columns by specifying only the row or column number:

```
print(A[2, ])
```

```
## [1] 2 5 8
```

```
print(A[ ,2])
```

```
## [1] 4 5 6
```

Perform matrix multiplication with the `%*%` operator:

```
x <- c(1, 2, 3)
A %*% x
```

```
##      [,1]
## [1,]   30
## [2,]   36
## [3,]   42
```

Lists

Lists are a special type of vector containing ordered key-value pairs and can contain arbitrary data types. Elements in lists can be accessed by key or by positional index.

```
phonebook <- list(name="Jenny", number="867-5309")
print(phonebook$name)
```

```
## [1] "Jenny"
```

```
print(phonebook[1])
```

```
## $name
## [1] "Jenny"
```

Dataframes

Data frames are `[row, column]` organized data objects. Rows contain data items (e.g. public health records) and columns contain values of different attributes (e.g. age, address). Values within a column should all have the same type.

R has a built-in `data.frame` type, and and the `tibble` package implements a version of the dataframe that has become very popular for data analysis. This document will use the `tbl_df` dataframe; for more information about the built-in `data.frame` see here. The `tibble` documentation outlines the differences between `tbl_df` and `data.frame`.

We will load the `tidyverse` package, which contains the `tibble` package and several other helpful data packages. This will make `tibble` operations available:

```r
library('tidyverse')
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
```

```
## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.6     v dplyr   1.0.7
## v tidyr   1.1.4     v stringr 1.4.0
## v readr   2.1.1     v forcats 0.5.1
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

We will use one of the built-in example datasets to briefly demonstrate interaction with dataframes. We will load the Motor Trend Car Road Tests, "mtcars" with the `data()` function, convert the row names to a column with the `rownames_to_column()` function, and then convert the dataset to a `tbl_df` with the `tibble()` function:

```r
data("mtcars")
mtcars <- rownames_to_column(mtcars, var="car_name")
mtcars <- tibble(mtcars)
```

The `head()` and `tail()` functions allow you to inspect the first several and last several rows of a dataframe, respectively:

```r
head(mtcars)
```

```
## # A tibble: 6 x 12
##   car_name       mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <chr>        <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Mazda RX4     21       6   160   110  3.9   2.62  16.5     0     1     4     4
## 2 Mazda RX4 W~  21       6   160   110  3.9   2.88  17.0     0     1     4     4
## 3 Datsun 710    22.8     4   108    93  3.85  2.32  18.6     1     1     4     1
## 4 Hornet 4 Dr~  21.4     6   258   110  3.08  3.22  19.4     1     0     3     1
## 5 Hornet Spor~  18.7     8   360   175  3.15  3.44  17.0     0     0     3     2
## 6 Valiant       18.1     6   225   105  2.76  3.46  20.2     1     0     3     1
```

```r
tail(mtcars)
```

```
## # A tibble: 6 x 12
##   car_name       mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <chr>        <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Porsche 914~  26       4 120.     91  4.43  2.14  16.7     0     1     5     2
## 2 Lotus Europa  30.4     4  95.1   113  3.77  1.51  16.9     1     1     5     2
## 3 Ford Panter~  15.8     8 351     264  4.22  3.17  14.5     0     1     5     4
## 4 Ferrari Dino  19.7     6 145     175  3.62  2.77  15.5     0     1     5     6
## 5 Maserati Bo~  15       8 301     335  3.54  3.57  14.6     0     1     5     8
## 6 Volvo 142E    21.4     4 121     109  4.11  2.78  18.6     1     1     4     2
```

The printed display from the `head()` and `tail()` functions indicates the variable type in each column. We see that all the columns contain type `<dbl>`, which is a double precision floating point number.

There is a header row with column names, which we can access using the `names()` function:

```
names(mtcars)
```

```
##  [1] "car_name" "mpg"      "cyl"      "disp"     "hp"       "drat"
##  [7] "wt"       "qsec"     "vs"       "am"       "gear"     "carb"
```

We can count the number of columns with the `dim()` function, which returns the dimensions in (row, column) order:

```
dim(mtcars)
```

```
## [1] 32 12
```

Because this is a built-in dataset, we can also inspect help documentation that tells us a bit more about the data:

```
?mtcars
```

The "Format" section tells us the column interpretations:

A data frame with 32 observations on 11 (numeric) variables.

[, 1] mpg Miles/(US) gallon [, 2] cyl Number of cylinders [, 3] disp Displacement (cu.in.) [, 4] hp Gross horsepower [, 5] drat Rear axle ratio [, 6] wt Weight (1000 lbs) [, 7] qsec 1/4 mile time [, 8] vs Engine (0 = V-shaped, 1 = straight) [, 9] am Transmission (0 = automatic, 1 = manual) [,10] gear Number of forward gears [,11] carb Number of carburetors

Selecting data

You can select data by column with the `$` operator:

```
mtcars$mpg
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
## [16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
## [31] 15.0 21.4
```

The %>% operator The `%>%` operator serves as a pipe, allowing you to build sequences of operations. You can read more about pipe operators here.

We can use the %>% operator and the `select()` function from the `dplyr` package (which was loaded as part of the `tidyverse` collection of packages) to select multiple columns at the same time:

```
mtcars %>% dplyr::select(mpg, disp)
```

```
## # A tibble: 32 x 2
##      mpg  disp
##    <dbl> <dbl>
##  1  21    160
##  2  21    160
##  3  22.8  108
##  4  21.4  258
##  5  18.7  360
##  6  18.1  225
##  7  14.3  360
##  8  24.4  147.
##  9  22.8  141.
## 10  19.2  168.
## # ... with 22 more rows
```

Data summaries

Summary statistics We can calculate the mean and standard deviation of individual columns in the `mtcars` dataset:

```
print(mean(mtcars$mpg))
```

```
## [1] 20.09062
```

```
print(sd(mtcars$mpg))
```

```
## [1] 6.026948
```

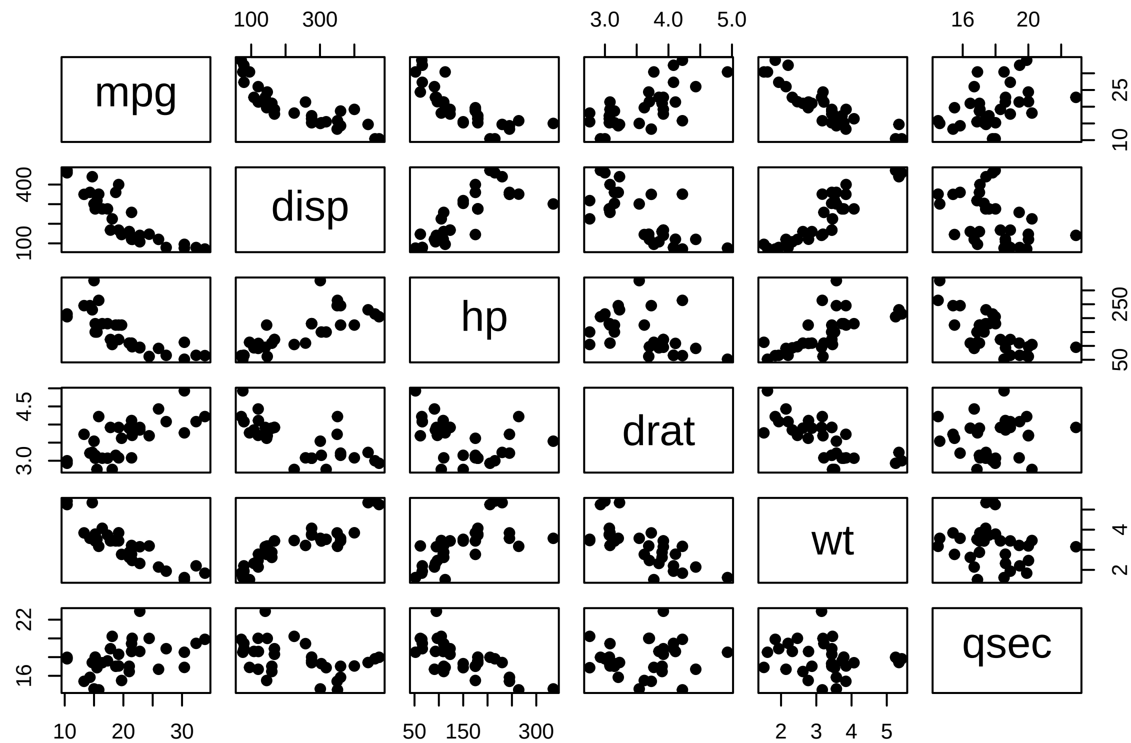
We can also calculate a summary for all the numeric (non-categorical) values in the dataset. The `summary()` function returns the minimum, 25th percentile (1st quartile), median, mean, 75th percentile (3rd quartile), and maximum values.

```
numeric_mtcars <- mtcars %>% dplyr::select(car_name, mpg, disp, hp, drat, wt, qsec)
summary(numeric_mtcars)
```

```
##    car_name              mpg             disp             hp       
##  Length:32          Min.   :10.40   Min.   : 71.1   Min.   : 52.0  
##  Class :character   1st Qu.:15.43   1st Qu.:120.8   1st Qu.: 96.5  
##  Mode  :character   Median :19.20   Median :196.3   Median :123.0  
##                     Mean   :20.09   Mean   :230.7   Mean   :146.7  
##                     3rd Qu.:22.80   3rd Qu.:326.0   3rd Qu.:180.0  
##                     Max.   :33.90   Max.   :472.0   Max.   :335.0  
##       drat             wt             qsec      
##  Min.   :2.760   Min.   :1.513   Min.   :14.50  
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89  
##  Median :3.695   Median :3.325   Median :17.71  
##  Mean   :3.597   Mean   :3.217   Mean   :17.85  
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90  
##  Max.   :4.930   Max.   :5.424   Max.   :22.90  
```

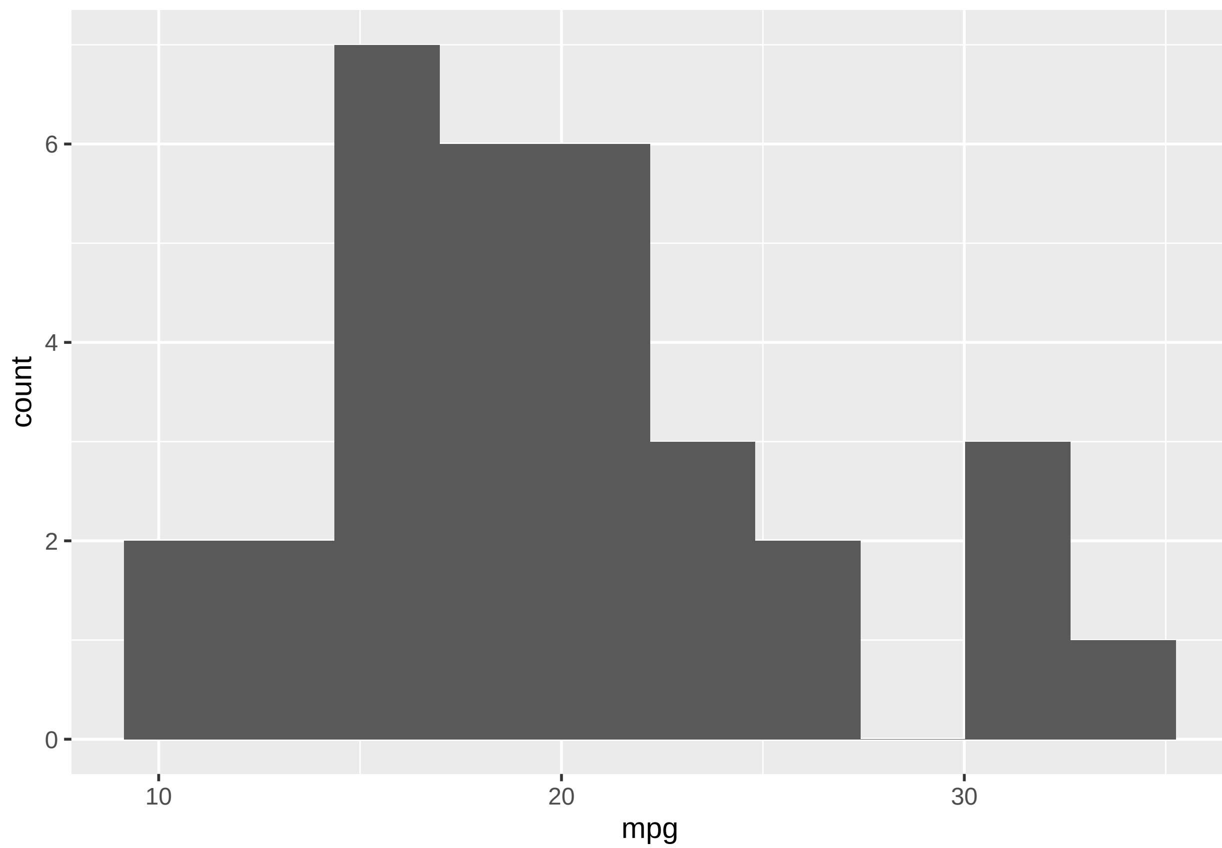
Summary plots The `pairs()` function shows generates a scatterplot matrix for all pairwise combinations of columnar variables in a dataset. The diagonal panel indicates the column name and the (x, y) axis coordinate labels. We use the `%>%` operator to drop out the `car_name` column and pass only the truly numeric data to the `pairs()` function.

```
numeric_mtcars %>% dplyr::select(-car_name) %>% pairs(pch=19)
```



Histograms provide summaries of the distribution of data. We can use the `ggplot2` library, also part of the `tidyverse`, to generate histograms for the numeric data. Because there are only 32 observations in the `mtcars` dataset, we use only 10 bins in our histogram.

```
ggplot(data=mtcars, aes(x=mpg)) + geom_histogram(bins=10)
```



Reshaping data

Datasets can be broadly categorized as having either "long" or "wide" form. The `mtcars` dataset is "wide" form data: each row is an observation, and each column represents a different measured variable. In contrast, "long" form data includes a single column for "variable type" and many more rows. Wide form data are useful when you need to select single variable types and work with them independently, and for certain types of aggregations. Long form data are useful when you want to do certain types of data joins or merges, and are sometimes required for plotting functions.

The `pivot_longer()` and `pivot_wider()` functions allow you to reshape data as needed.

```
numeric_mtcars_long <- numeric_mtcars %>%
  pivot_longer(!car_name, names_to="variable", values_to="value")
head(numeric_mtcars_long)
```

```
## # A tibble: 6 x 3
##   car_name  variable  value
##   <chr>     <chr>     <dbl>
## 1 Mazda RX4 mpg       21
## 2 Mazda RX4 disp     160
## 3 Mazda RX4 hp       110
## 4 Mazda RX4 drat       3.9
## 5 Mazda RX4 wt         2.62
## 6 Mazda RX4 qsec      16.5
```

With our long format data, we can easily create a series of histograms with `ggplot`. We use the `scales="free"` argument to allow each panel to have its own x-axis range.

```
ggplot(data=numeric_mtcars_long, aes(value)) +
  geom_histogram(bins=10) +
  facet_wrap(~variable, scales="free")
```