



R — Subsetting

In the functional language R, one can subset data in various ways. Subsetting is one of the fundamental operations and this article outlines the basics.



Christina B.

8 min read · Feb 25, 2022



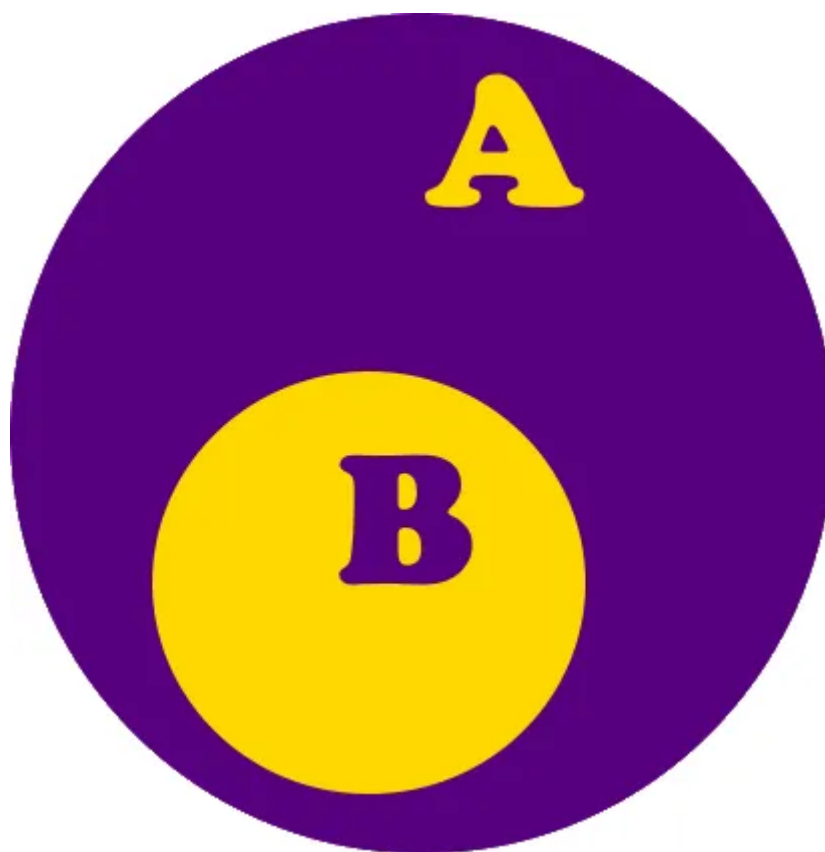
Listen



Share



More



B is a subset of A

A **subset** is a set of each of whose elements is an element of an inclusive set. This is defined in set theory.

For example, if $A=\{1,2,3,4\}$ then $B=\{2,4\}$, so B is a proper subset of A.

Operators

One can extract subsets of R objects with the following operators:

- `[]` Always returns the same object. Can select more than one.
- `[[]]` For list and data frame extractions. Only one item. The returned object might have a different class.
- `$` Extract by 'name' from list or data frame.

Simple or preserved

As one can choose the operators to various data types the data returned is either

- **simplified** (to the most basic return type) or
- **preserved** so input type = output type.

Below you see a simple table, which shows how to use the subsetting operators on different data types and what the output of the subset will be.

		Simplifying	Preserving
	Vector	<code>x[[1]]</code>	<code>x[1]</code>
	Atomic Vector	<code>x[[1]]</code> is the same as <code>x[1]</code>	
	List	<code>x[[1]]</code> <code>#[]</code> gets list contents	<code>x[1]</code> <code>#[]</code> returns a list
Drops any unused levels but it remains a factor class	Factor	<code>x[1:4, drop = T]</code>	<code>x[1:4]</code>
dimension.length = 1 -> dimension is dropped	Array	<code>x[1,]</code> or <code>x[, 1]</code>	<code>x[1, , drop = F]</code> or <code>x[, 1, drop = F]</code>
output = single column -> returns vector	Data frame	<code>x[, 1]</code> or <code>x[[1]]</code>	<code>x[, 1, drop = F]</code> or <code>x[1]</code>

This can seem a bit overwhelming, so I have gathered some examples to explain the most common use cases with each type.

ATOMIC VECTOR

Atomic vectors are the fundament of the data structures in R. Atomic vectors have no dimensions (no *dim* attribute). An atomic vector has elements that all are of the same type.

You construct an atomic vector with either the function `c()` or `vector()`. But one can also use `seq()` and `as.vector()`. Atomic vectors have three properties: type, length, attributes.

<https://renenyffenegger.ch/notes/development/languages/R/data-structures/vector/>

```
x <- rep_len(1:10, 1e7)
```

VECTOR

Vectors can be any data type. For example this numeric vector:

```
vector <- c(1,2,3,4,5,6,7,8,9,10) # a numeric vector
```

To subset elements from this vector one can apply many different tactics. Depending on what end result suits best. See the below table for reference.

	Format	Result
Return ALL	vector[]	[1] 1 2 3 4 5 6 7 8 9 10
Preserved type	vector[3]	[1] 3
Simplified type	vector[[3]]	[1] 3
Range	vector[1:3]	[1] 1 2 3
Specific elements	vector[c(2, 5)]	[1] 2 5
Everything except	vector[-4]	[1] 1 2 3 5 6 7 8 9 10
Select logically	vector[c(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE)]	[1] 1 3 4 6 7 9 10
All which pass this	vector[vector > 5]	[1] 6 7 8 9 10
Subset function	subset(vector, vector > 5)	[1] 6 7 8 9 10
Replace all values with x	vector[] <- 'x'	[1] "x" "x" "x" "x" "x" "x" "x" "x" "x" "x"
Override vector as number	vector <- 1	[1] 1

<https://r-coder.com/subset-r/>

With a vector, one can do numerous operations as the table above demonstrates. However, the preserved and simplified types don't really show in this example. If you add a name to the vector it is easier.

```
vectorNames <- setNames(vector, letters[1:10]) # assign letters as
names to every lement in the vector.
```

```
vectorNames["i"]          # preserved: Element which has the name "i"
i
9
```

```
vectorNames[["i"]]        # simplified: Element "i"
[1] 9
```

```
vectorNames[c("a", "c")]  # Preserved: Elements "a" and "c"
a c
1 3
```


Or you can extract with the double bracket and enter a name as a string.

```
x[['two']]  
[1] 7.0 # the contents of list item  
'two'
```

However, if you use the single bracket, you get back a list.

```
x['two']  
$two # a list with one  
element  
[1] 7.0
```

Extracting more than one

In this case, you have to use the **single bracket operator**. So, for example, to get the element at index 1 and 2 I can simply specify this as a vector:

```
x[c(1, 2)]  
$one # A list is returned with two  
elements: 'one' and 'two'  
[1] 1 2 3 4  
$two  
[1] 7.0
```

Subsetting nested elements

The double bracket operator recurses into the list, that is why it is used to subset nested elements. It can take an integer sequence.

Below, a list is created with another list and a vector. Using the double bracket operator with a vector (1,3) tells R that we want the first element of the list 'a', and within 'a' we want the third element, which is the number 8.

```
x <- list(a = list(6,7,8), b = c(1,2,3))  
# method 1: with a vector  
x[[c(1,3)]]  
[1] 8
```

```
# method 2: two times double brackets
x[[1]][[3]]
[1] 8
```

MATRIX

Subsetting matrices with INDEX

In using single square brackets, one can specify the subset to be extracted with indexes. The first index selects rows, the second index selects columns.

```
d <- matrix(1:6, 2, 3)
x[1, 2]

[1] 3          # Returns one element: vector. first row, second
column

x[1, ]

[1] 1 3 5      # the first row, all
columns
```

When the matrix returns a single value, it is a vector and not a matrix. You can, however, get a matrix back, if you disable the default setting. 'Drop' drops the dimensions if it is TRUE.

```
x[1, 2, drop=FALSE]

[, 1]
[1, ] 3          # a one by one matrix with the element 3
```

Subsetting with partial matching

You can use it on names with the `[[]]` and the `$`. Partial matching means, you can type part of the name, and if there is a match it is returning it. If you have a very long name, which you don't want to type (or type a lot), you can use the `$` sign:

```
g <- list(hurteknadil= 1:5)
g$h
[1] 1 2 3 4 5
```

The `[]` operator does not do this on default. It expects the name you pass it to be exact. But you can turn this off with the 'exact' attribute.

```
g[['a', exact = FALSE]]  
[1] 1 2 3 4 5
```

Remove missing values

Subsetting by removing missing values from a single object

Missing values are in R called 'NA' (not available). In most datasets, there are many missing values, so this is an operation that will be done frequently.

For a vector, matrix or data frame you can use the `is.na` function. You can create a logical vector that maps where NA is found and then remove them by subsetting like the example below:

```
t <- c(6, 10, NA, 5, 7, NA)  
empty <- is.na(t)  
  
empty # logical vector, maps which items are NA by true/false  
[1] FALSE FALSE  TRUE FALSE FALSE  TRUE  
  
t[!empty] # gives all items in 't', which are mapped as FALSE  
[1] 6, 10, 5, 7
```

Subsetting by removing missing values from multiple objects

If you have more than one object, and you want to combine them, have a subset and also remove the NA values. For this, you will want to use the `complete.cases()` function. See a great tutorial and examples [here](#). There are different approaches, for dealing with vectors, matrix and data frames.

vector

```
x <- c(1,2,NA,4,NA, 5)  
x <- x[complete.cases(x)]  
x  
  
[1] 1 2 4 5
```

matrix

```

m <- matrix( sample( c(NA,1:5) , 15 , TRUE ) , 5 )
sum(complete.cases(m))           # all complete
cases
sum(complete.cases(m[,1:2]))     # first two
columns

[1] 3
[1] 4

```

data frame.

```

df <- data.frame(x=c(7, 84, NA, 6, NA, 79),
                 y=c(NA, 3, 42, 8, NA, 42),
                 z=c(NA, 7, 3, 55, 4, 14))

```

df

x <dbl>	y <dbl>	z <dbl>
7	NA	NA
84	3	7
NA	42	3
6	8	55
NA	NA	4
79	42	14

Now we have 6 NA values. We can remove the rows which have the NA values.

```

#method 1
df <- df[complete.cases(df), ] # remove all rows with NA

#method 2
accepted <- complete.cases(df) # which rows are complete?
df[accepted, ]

df <- df[complete.cases(df[, c('y', 'z')]), ] # remove only from
column y and z

```


x <dbl>	y <dbl>	z <dbl>
84	3	7
6	8	55
79	42	14

x <dbl>	y <dbl>	z <dbl>
84	3	7
NA	42	3
6	8	55
79	42	14

Computed index

In the case that we have a computed name for an index in a list, we have to use the **double bracket operator** `[[]]`. This is because you have to search the list with a literal symbol (the dollar sign won't work).

```
name <- 'two'           # computed name
variable
x[[name]]
[1] 7.0
```

Vectorized operations

This is **used instead of looping**. Things can happen in parallel with vectorized operations. So, with for example two vectors, if you wanted to add them together; say add the first item of vector 1 to the first item of vector 2 and so on. This can be done in parallel, and with just a few lines of code very intuitively. Code is easier to write, and this is the beauty of functional programming.

All the normal mathematical operation can be used this way (+ , - , * , /) and also logical comparisons (> , < , == , <= , >=).

See some more nice examples [here](#).

Vectors

```
c <- 1:4
h <- 6:9
c + h
[1] 7 9 11 13                # 1 + 6 = 7, 2 + 7 = 9 etc.

y == 2                      # test all elements in y
FALSE TRUE FALSE FALSE
```

Matrices

There are two types of vectorized operations. Element by element or 'true' matrix multiplication. In the element operation, the first matrix element from row1, column 1 is calculated with the same element from the other matrix. In the true matrix operation, each element is calculated with each element of a row.

Element by element

```
r <- matrix(1:4, 2, 2)
e <- matrix(rep(10, 4), 2, 2)
r
e
      [,1] [,2]
[1,]    1    3
[2,]    2    4

      [,1] [,2]
[1,]   10   10
[2,]   10   10

multiplication> r * e
      [,1] [,2]
[1,]    10   30
[2,]    20   40                # 1 * 10 = 10, and 3 * 3 = 30
                                # 2 * 10 = 20, and 4 * 10 = 40
```

True matrix multiplication

```
multiplication> r %*% e
      [,1] [,2]
[1,]    40   40                # (1*10) + (3*10) = 40
[2,]    60   60
```

Resources

See more examples on subsetting :link: [here](#).

R

Subsets

Some rights reserved ⓘ



Edit profile

Written by Christina B.

7 Followers

Master of the dark art of CSS | Computer Science Student | INFJ | Science fiction lover | Green fingers and red hair hippie witch

More from Christina B.

icken and the Fox.

line-height

oon a time ...

letter-spacing: 0.12em

16em

 Christina B.

Text Spacing

WCAG 2.1 Success Criterion 1.4.12 (Level AA)


8 min read · Jan 28, 2022



See all from Christina B.

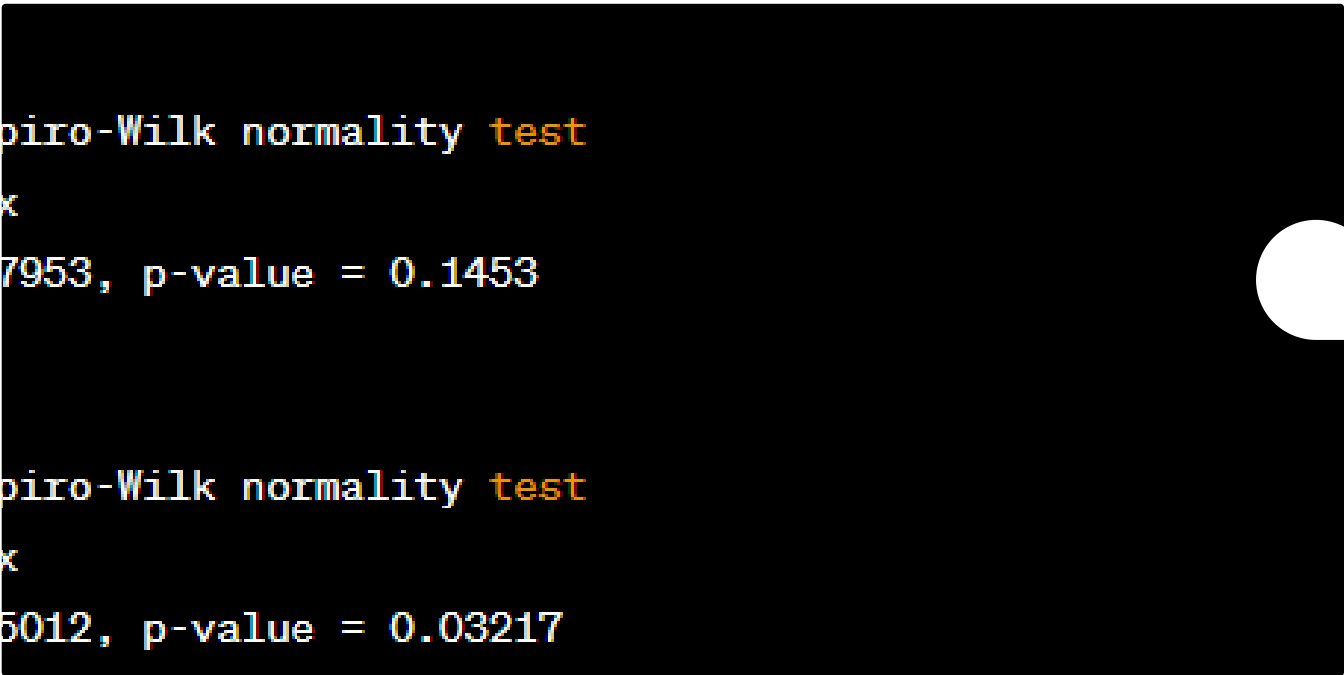
Recommended from Medium



 Romina Mendez

Transform your R Dataframes: Styles, Colors, and Emojis

A few weeks ago I wrote an article about pandas dataframes and how to assign styles, but I received messages about how to do it in R (my...



 James Jones

Comparing Date Ranges among Groups with R

Hi all! Recently, while deeply immersed in a work project, I found myself exploring statistical significance by different groups based on...

3 min read · Jan 5, 2024

Lists



Staff Picks

634 stories · 948 saves



Stories to Help You Level-Up at Work

19 stories · 600 saves



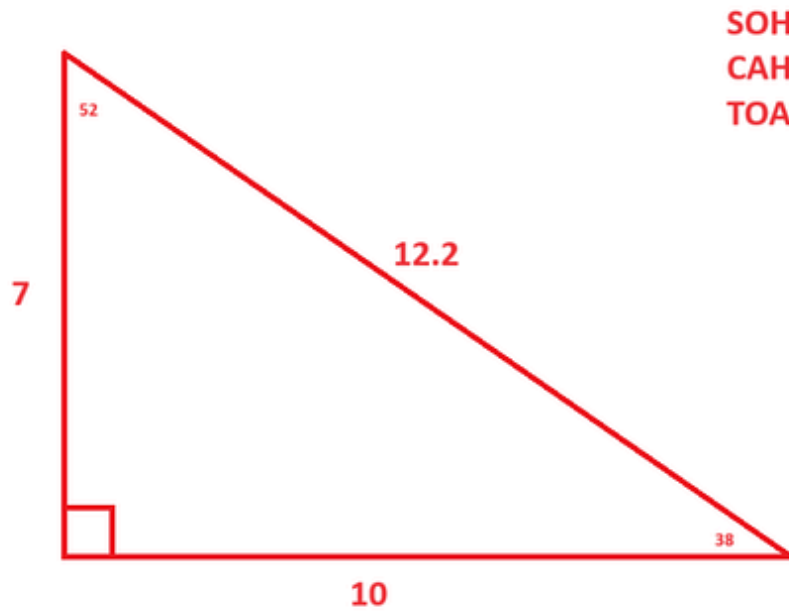
Self-Improvement 101


20 stories · 1732 saves



Productivity 101

20 stories · 1610 saves




 Michael Orozco-Fletcher

R Lesson 33: Inverse Trigonometric Ratios in R

3 min read · Apr 10, 2024



 Nilimesh Halder, PhD in Level Up Coding

Mastering the Essentials of Structured Data: A Comprehensive Guide with Python and R Examples

Mastering the Essentials of Structured Data: A Comprehensive Guide with Python and R Examples

★ · 26 min read · Apr 24, 2024



Chaitanya Pradeep in Uplearner.AI Clinical Blog

Mastering the Mysteries of R Lists: Unraveling the Intricacies of R's Versatile Data Structure

Lists in R are a fundamental data structure that allow you to store a collection of elements, potentially of different types. Here are some...

4 min read · Jan 27, 2024



description: df [96 × 3]

	Actual <dbl>	Predicted <dbl>	rmse <dbl>
Mazda RX4	16.46	18.61311	2.15311084
Mazda RX4 Wag	17.02	18.58948	1.56948188
Datsun 710	18.61	18.85208	0.24208410
Hornet 4 Drive	19.44	18.48737	0.95262898
Hornet Sportabout	17.02	17.33783	0.31782942
Volvo 460	20.22	18.54595	1.67404660
Cadillac Fleetwood	15.84	16.05505	0.21505052
Ford Pantera	20.00	19.36048	0.63952000
Ford Pantera L	22.90	18.58907	4.31000000
Ford Mustang Mach 1	18.30	18.28547	0.01452000

10 of 96 rows

Previous 1 2 3



R Train Data

Leave-One-Out Cross-Validation (LOOCV) for Linear Regression in R using mtcars

Cross-validation is an great technique for model evaluation that allows us to understand both bias and variance components in the models we...

5 min read · Apr 10, 2024



See more recommendations