

vapply and tapply

In the last lesson, you learned about the two most fundamental members of R's `*apply` family of functions: `lapply()` and `sapply()`. Both take a list as input, apply a function to each element of the list, then combine and return the result. `lapply()` always returns a list, whereas `sapply()` attempts to simplify the result.

In this lesson, you'll learn how to use `vapply()` and `tapply()`, each of which serves a very specific purpose within the Split-Apply-Combine methodology. For consistency, we'll use the same dataset we used in the 'lapply and sapply' lesson.

The Flags dataset from the UCI Machine Learning Repository contains details of various nations and their flags. More information may be found here: <http://archive.ics.uci.edu/ml/datasets/Flags>

I've stored the data in a variable called `flags`. If it's been a while since you completed the 'lapply and sapply' lesson, you may want to reacquaint yourself with the data by using functions like `dim()`, `head()`, `str()`, and `summary()` when you return to the prompt (`>`). You can also type `viewinfo()` at the prompt to bring up some documentation for the dataset. Let's get started!

As you saw in the last lesson, the `unique()` function returns a vector of the unique values contained in the object passed to it. Therefore, `sapply(flags, unique)` returns a list containing one vector of unique values for each column of the flags dataset. Try it again now.

```
sapply(flags, unique)
```

```
## $name
##      [1] Afghanistan      Albania
##      [3] Algeria             American-Samoa
##      [5] Andorra             Angola
##      [7] Anguilla            Antigua-Barbuda
##      [9] Argentina           Argentine
##     [11] Australia           Austria
##     [13] Bahamas             Bahrain
##     [15] Bangladesh           Barbados
##     [17] Belgium             Belize
##     [19] Benin               Bermuda
##     [21] Bhutan              Bolivia
##     [23] Botswana            Brazil
##     [25] British-Virgin-Isles Brunei
##     [27] Bulgaria            Burkina
##     [29] Burma              Burundi
##     [31] Cameroon            Canada
##     [33] Cape-Verde-Islands  Cayman-Islands
##     [35] Central-African-Republic Chad
##     [37] Chile              China
##     [39] Colombia            Comorro-Islands
##     [41] Congo              Cook-Islands
```

## [43] Costa-Rica	Cuba
## [45] Cyprus	Czechoslovakia
## [47] Denmark	Djibouti
## [49] Dominica	Dominican-Republic
## [51] Ecuador	Egypt
## [53] El-Salvador	Equatorial-Guinea
## [55] Ethiopia	Faeroes
## [57] Falklands-Malvinas	Fiji
## [59] Finland	France
## [61] French-Guiana	French-Polynesia
## [63] Gabon	Gambia
## [65] Germany-DDR	Germany-FRG
## [67] Ghana	Gibraltar
## [69] Greece	Greenland
## [71] Grenada	Guam
## [73] Guatemala	Guinea
## [75] Guinea-Bissau	Guyana
## [77] Haiti	Honduras
## [79] Hong-Kong	Hungary
## [81] Iceland	India
## [83] Indonesia	Iran
## [85] Iraq	Ireland
## [87] Israel	Italy
## [89] Ivory-Coast	Jamaica
## [91] Japan	Jordan
## [93] Kampuchea	Kenya
## [95] Kiribati	Kuwait
## [97] Laos	Lebanon
## [99] Lesotho	Liberia
## [101] Libya	Liechtenstein
## [103] Luxembourg	Malagasy
## [105] Malawi	Malaysia
## [107] Maldive-Islands	Mali
## [109] Malta	Marianas
## [111] Mauritania	Mauritius
## [113] Mexico	Micronesia
## [115] Monaco	Mongolia
## [117] Montserrat	Morocco

## [119] Mozambique	Nauru
## [121] Nepal	Netherlands
## [123] Netherlands-Antilles	New-Zealand
## [125] Nicaragua	Niger
## [127] Nigeria	Niue
## [129] North-Korea	North-Yemen
## [131] Norway	Oman
## [133] Pakistan	Panama
## [135] Papua-New-Guinea	Parguay
## [137] Peru	Philippines
## [139] Poland	Portugal
## [141] Puerto-Rico	Qatar
## [143] Romania	Rwanda
## [145] San-Marino	Sao-Tome
## [147] Saudi-Arabia	Senegal
## [149] Seychelles	Sierra-Leone
## [151] Singapore	Soloman-Islands
## [153] Somalia	South-Africa
## [155] South-Korea	South-Yemen
## [157] Spain	Sri-Lanka
## [159] St-Helena	St-Kitts-Nevis
## [161] St-Lucia	St-Vincent
## [163] Sudan	Surinam
## [165] Swaziland	Sweden
## [167] Switzerland	Syria
## [169] Taiwan	Tanzania
## [171] Thailand	Togo
## [173] Tonga	Trinidad-Tobago
## [175] Tunisia	Turkey
## [177] Turks-Cocos-Islands	Tuvalu
## [179] UAE	Uganda
## [181] UK	Uruguay
## [183] US-Virgin-Isles	USA
## [185] USSR	Vanuatu
## [187] Vatican-City	Venezuela
## [189] Vietnam	Western-Samoa
## [191] Yugoslavia	Zaire
## [193] Zambia	Zimbabwe

```
## 194 Levels: Afghanistan Albania Algeria American-Samoa Andorra ... Zimbabwe
##
## $landmass
## [1] 5 3 4 6 1 2
##
## $zone
## [1] 1 3 2 4
##
## $area
## [1] 648 29 2388 0 1247 2777 7690 84 19 1 143
## [12] 31 23 113 47 1099 600 8512 6 111 274 678
## [23] 28 474 9976 4 623 1284 757 9561 1139 2 342
## [34] 51 115 9 128 43 22 49 284 1001 21 1222
## [45] 12 18 337 547 91 268 10 108 249 239 132
## [56] 2176 109 246 36 215 112 93 103 3268 1904 1648
## [67] 435 70 301 323 11 372 98 181 583 236 30
## [78] 1760 3 587 118 333 1240 1031 1973 1566 447 783
## [89] 140 41 1267 925 121 195 324 212 804 76 463
## [100] 407 1285 300 313 92 237 26 2150 196 72 637
## [111] 1221 99 288 505 66 2506 63 17 450 185 945
## [122] 514 57 5 164 781 245 178 9363 22402 15 912
## [133] 256 905 753 391
##
## $population
## [1] 16 3 20 0 7 28 15 8 90 10 1 6 119 9
## [15] 35 4 24 2 11 1008 5 47 31 54 17 61 14 684
## [29] 157 39 57 118 13 77 12 56 18 84 48 36 22 29
## [43] 38 49 45 231 274 60
##
## $language
## [1] 10 6 8 1 2 4 3 5 7 9
##
## $religion
## [1] 2 6 1 0 5 3 4 7
##
## $bars
## [1] 0 2 3 1 5
##
```

```
## $stripes
## [1] 3 0 2 1 5 9 11 14 4 6 13 7
##
## $colours
## [1] 5 3 2 8 6 4 7 1
##
## $red
## [1] 1 0
##
## $green
## [1] 1 0
##
## $blue
## [1] 0 1
##
## $gold
## [1] 1 0
##
## $white
## [1] 1 0
##
## $black
## [1] 1 0
##
## $orange
## [1] 0 1
##
## $mainhue
## [1] green red blue gold white orange black brown
## Levels: black blue brown gold green orange red white
##
## $circles
## [1] 0 1 4 2
##
## $crosses
## [1] 0 1 2
##
## $saltires
```

```
## [1] 0 1
##
## $quarters
## [1] 0 1 4
##
## $sunstars
## [1] 1 0 6 22 14 3 4 5 15 10 7 2 9 50
##
## $crescent
## [1] 0 1
##
## $triangle
## [1] 0 1
##
## $icon
## [1] 1 0
##
## $animate
## [1] 0 1
##
## $text
## [1] 0 1
##
## $topleft
## [1] black red green blue white orange gold
## Levels: black blue gold green orange red white
##
## $botright
## [1] green red white black blue gold orange brown
## Levels: black blue brown gold green orange red white
```

What if you had forgotten how `unique()` works and mistakenly thought it returns the *number* of unique values contained in the object passed to it? Then you might have incorrectly expected `sapply(flags, unique)` to return a numeric vector, since each element of the list returned would contain a single number and `sapply()` could then simplify the result to a vector.

When working interactively (at the prompt), this is not much of a problem, since you see the result immediately and will quickly recognize your mistake. However, when working non-interactively (e.g. writing your own functions), a misunderstanding may go undetected and cause incorrect results later on. Therefore, you may wish to be more careful and that's where `vapply()` is useful.

Whereas `apply()` tries to 'guess' the correct format of the result, `vapply()` allows you to specify it explicitly. If the result doesn't match the format you specify, `vapply()` will throw an error, causing the operation to stop. This can prevent significant problems in your code that might be caused by getting unexpected return values from `apply()`.

Try `vapply(flags, unique, numeric(1))`, which says that you expect each element of the result to be a numeric vector of length 1. Since this is NOT actually the case, YOU WILL GET AN ERROR. Once you get the error, type `ok()` to continue to the next question.

```
ok()
```

Recall from the previous lesson that `apply(flags, class)` will return a character vector containing the class of each column in the dataset. Try that again now to see the result.

```
apply(flags, class)
##      name  landmass      zone      area population  language
## "factor" "integer" "integer" "integer" "integer" "integer" "integer"
## religion      bars    stripes    colours      red      green
## "integer" "integer" "integer" "integer" "integer" "integer" "integer"
##      blue      gold      white      black      orange    mainhue
## "integer" "integer" "integer" "integer" "integer" "integer" "factor"
## circles crosses saltires quarters sunstars crescent
## "integer" "integer" "integer" "integer" "integer" "integer" "integer"
## triangle      icon    animate      text    topleft    botright
## "integer" "integer" "integer" "integer" "integer" "factor" "factor"
```

If we wish to be explicit about the format of the result we expect, we can use `vapply(flags, class, character(1))`. The 'character(1)' argument tells R that we expect the class function to return a character vector of length 1 when applied to EACH column of the flags dataset. Try it now.

```
vapply(flags, class, character(1))
##      name  landmass      zone      area population  language
## "factor" "integer" "integer" "integer" "integer" "integer" "integer"
## religion      bars    stripes    colours      red      green
## "integer" "integer" "integer" "integer" "integer" "integer" "integer"
##      blue      gold      white      black      orange    mainhue
## "integer" "integer" "integer" "integer" "integer" "integer" "factor"
## circles crosses saltires quarters sunstars crescent
## "integer" "integer" "integer" "integer" "integer" "integer" "integer"
## triangle      icon    animate      text    topleft    botright
## "integer" "integer" "integer" "integer" "integer" "factor" "factor"
```

Note that since our expectation was correct (i.e. `character(1)`), the `vapply()` result is identical to the `apply()` result – a character vector of column classes.

You might think of `vapply()` as being 'safer' than `apply()`, since it requires you to specify the format of the output in advance, instead of just allowing R to 'guess' what you wanted. In addition, `vapply()` may perform faster than

supply() for large datasets. However, when doing data analysis interactively (at the prompt), supply() saves you some typing and will often be good enough.

As a data analyst, you'll often wish to split your data up into groups based on the value of some variable, then apply a function to the members of each group. The next function we'll look at, tapply(), does exactly that.

Use ?tapply to pull up the documentation.

```
?tapply
```

The 'landmass' variable in our dataset takes on integer values between 1 and 6, each of which represents a different part of the world. Use table(flags\$landmass) to see how many flags/countries fall into each group.

```
table(flags$landmass)
##
##  1  2  3  4  5  6
## 31 17 35 52 39 20
```

The 'animate' variable in our dataset takes the value 1 if a country's flag contains an animate image (e.g. an eagle, a tree, a human hand) and 0 otherwise. Use table(flags\$animate) to see how many flags contain an animate image.

```
table(flags$animate)
##
##    0    1
## 155   39
```

This tells us that 39 flags contain an animate object (animate = 1) and 155 do not (animate = 0).

If you take the arithmetic mean of a bunch of 0s and 1s, you get the proportion of 1s. Use tapply(flags\$animate, flags\$landmass, mean) to apply the mean function to the 'animate' variable separately for each of the six landmass groups, thus giving us the proportion of flags containing an animate image WITHIN each landmass group.

```
tapply(flags$animate, flags$landmass, mean)
##           1           2           3           4           5           6
## 0.4193548 0.1764706 0.1142857 0.1346154 0.1538462 0.3000000
```

The first landmass group (landmass = 1) corresponds to North America and contains the highest proportion of flags with an animate image (0.4194).

Similarly, we can look at a summary of population values (in round millions) for countries with and without the color red on their flag with tapply(flags\$population, flags\$red, summary).

```
tapply(flags$population, flags$red, summary)
## $`0`
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00   0.00    3.00   27.63    9.00   684.00
##
## $`1`
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
```


##	0.0	0.0	4.0	22.1	15.0	1008.0
----	-----	-----	-----	------	------	--------

What is the median population (in millions) for countries *without* the color red on their flag?

1. 9.0
2. 4.0
3. 27.6
4. 3.0
5. 22.1
6. 0.0

3

Lastly, use the same approach to look at a summary of population values for each of the six landmasses.

```
tapply(flags$population, flags$landmass, summary)

## $`1`
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##      0.00   0.00   0.00   12.29   4.50  231.00
##
## $`2`
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##      0.00   1.00   6.00   15.71  15.00  119.00
##
## $`3`
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##      0.00   0.00   8.00   13.86  16.00   61.00
##
## $`4`
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##      0.000   1.000   5.000   8.788   9.750  56.000
##
## $`5`
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##      0.00   2.00  10.00   69.18  39.00 1008.00
##
## $`6`
##      Min. 1st Qu.  Median      Mean 3rd Qu.      Max.
##      0.00   0.00   0.00   11.30   1.25  157.00
```

What is the maximum population (in millions) for the fourth landmass group (Africa)?

1. 56.00
2. 1010.0
3. 119.0
4. 5.00

5. 157.00

56

In this lesson, you learned how to use `vapply()` as a safer alternative to `sapply()`, which is most helpful when writing your own functions. You also learned how to use `tapply()` to split your data into groups based on the value of some variable, then apply a function to each group. These functions will come in handy on your quest to become a better data analyst.