mattm / **r-cheat-sheet**

Branch: master ▾ r-cheat-sheet / Data Frames.md Find file Copy path

mattm Update notes aaca237 on 2 May 2017

1 contributor

322 lines (243 sloc) 5.92 KB

Data Frames

> A typical data set contains data of different modes. In an employee data set, for example, we might have character string data, such as employee names, and numeric data, such as salaries. So, although a data set of (say) 50 employees with 4 variables per worker has the look and feel of a 50-by-4 matrix, it does not qualify as such in R, because it mixes types.

> Instead of a matrix, we use a data frame. A data frame in R is a list, with
> each component of the list being a vector corresponding to a column in our
> “matrix” of data. Indeed, you can create data frames in just this way:

> *The Art of R Programming*

Creating a data frame from lists

```
> d <- data.frame(list(kids = c("Jack", "Jill"), ages = c(12, 10)))

> d
  kids ages
1 Jack   12
2 Jill   10
```

Checking the dimensions of a data frame

That's **rows** then **columns**:

```
> d <- data.frame(list(kids = c("Jack", "Jill", "Johnny"), ages = c(12, 10, 4)))
> d
    kids ages
1   Jack   12
2   Jill   10
3 Johnny    4
> dim(d)
[1] 3 2
```

Returning the column names

```
> colnames(d)
[1] "kids" "ages"
```

Viewing a summary of the data

```
> summary(d)
     kids        ages
 Jack  :1   Min.   : 4.000
 Jill  :1   1st Qu.: 7.000
 Johnny:1   Median :10.000
            Mean   : 8.667
            3rd Qu.:11.000
            Max.   :12.000
```

Viewing the structure of the data

```
> str(d)
'data.frame':   3 obs. of  2 variables:
 $ kids: Factor w/ 3 levels "Jack","Jill",..: 1 2 3
 $ ages: num  12 10 4
```

Returning the values of a data frame component

Just like you would a list:

```
> d$kids
[1] Jack Jill
Levels: Jack Jill
```

Returning a component of the data frame

Use single brackets [ ] to return a list:

```
> d['kids']
  kids
1 Jack
2 Jill
```

Using the standard `[]` method

```
> d <- data.frame(list(kids = c("Jack", "Jill"), ages = c(12, 10)))
> d[d$kids == "Jack",]
  kids ages
1 Jack   12
```

Subsetting using `subset`

Single column, exact value

```
> housing <- read.csv("data/landdata-states.csv")
```

With `subset`:

```
> fl = subset(housing, State == "FL")
```

With `dplyr`'s `filter` function:

```
> fl = filter(housing, State == "FL")
```

Single column, any of multiple values

With `subset`:

```
both = subset(housing, State %in% c("FL", "GA"))
```

With dplyr's `filter`:

```
> both = filter(housing, State == "FL" | State == "GA")
```

Multiple columns

With `subset`:

```
> subset(housing,  State == "AK" & Home.Value == 224952)
```

With dplyr's `filter`:

```
> filter(housing, State == "AK" & Home.Value == 224952)
```

Re-ordering rows

With `order`:

```
> # Ascending
> housing[order(housing$Home.Value), ]

> # Descending
> housing[order(-housing$Home.Value), ]
> housing[order(housing$Home.Value, decreasing = TRUE), ]
```

With dplyr's `arrange`:

```
> # Ascending
> arrange(housing, Home.Value)

> # Descending
> arrange(housing, desc(Home.Value))
```

Selecting columns

With subsetting:

```
> housing[, c("State", "Home.Value")]
```

With dplyr's `select`:

```
> select(housing, State, Home.Value)
```

Removing a column

With `select` :

```
housing <- select(housing, -State)
```

Renaming a column

With dplyr's `rename` :

```
> rename(housing, State.Name = State)
```

Note that `State.Name` is the *new* name.

Extract distinct (unique) rows

With `unique` :

```
> unique(housing[, c("State", "region")])
```

With dplyr's `distinct` :

```
> distinct(housing, State, region)
```

Removing NA values

Use `complete.cases` :

```
> d <- data.frame(list(kids = c("Jack", "Jill"), ages = c(12, NA)))
> d
  kids ages
1 Jack   12
2 Jill   NA
> d[complete.cases(d), ]
  kids ages
1 Jack   12
```

Taking a sample

With `sample` :

```
> housing[sample(nrow(housing), 5), ]
```

(Source)

With dplyr's `sample_n` :

```
> sample_n(housing, 5)
```

There's also a `sample_frac` that grabs a random percentage.

Adding a new column

```
> d <- data.frame(list(name = c("Jack", "Jill"), age = c(12, 10)))
> d
  name age
1 Jack   12
2 Jill   10
```

Natively:

```
> d$next_age = d$ages + 1
> d
  name age  next_age
1 Jack   12       13
2 Jill   10       11
```

With dplyr's `mutate` :

```
> d <- mutate(d, next_age = ages + 1)
> d
  name  age next_age
1 Jack   12       13
2 Jill   10       11
```

Note that with `mutate` , you can reference columns you're currently adding:

```
> d <- mutate(d, next_age = age + 1, next_next_age = next_age + 1)
> d
  name age next_age next_next_age
1 Jack  12       13            14
2 Jill  10       11            12
```

Grouping Operations

Applying `summarize` to groups of observations

With one summary statistic:

```
> by_state = group_by(housing, State)
> summarize(by_state, Avg.Home.Value = mean(Home.Value))
# A tibble: 51 × 2
    State     Avg.Home.Value
   <fctr>              <dbl>
1      AK          147385.14
2      AL           92545.22
3      AR           82076.84
4      AZ          140755.59
```

With multiple:

```
> by_state = group_by(housing, State)
> summarize(by_state, count = n(), Avg.Home.Value = mean(Home.Value))
# A tibble: 51 × 3
    State count Avg.Home.Value
   <fctr> <int>          <dbl>
1      AK   153      147385.14
2      AL   153       92545.22
3      AR   153       82076.84
```

```
4      AZ   153      140755.59
5      CA   153      282808.08
```

Note that `n()` is an *aggregate function* provided by dplyr that returns the number of observations in each group, which in this example are all 153.

There are other aggregate functions as well: `n_distinct(x)` (the number of unique values of x), `first(x)` , `last(x)` , and `nth(x)` .

We can chain dplyr functions together using the `%>%` operator:

```
> group_by(housing, State) %>% summarize(Avg.Home.Value = mean(Home.Value))
# A tibble: 51 × 2
    State Avg.Home.Value
   <fctr>          <dbl>
1      AK      147385.14
2      AL       92545.22
3      AR       82076.84
4      AZ      140755.59
5      CA      282808.08
```