

dplyr : Grammar of data manipulation

What is dplyr?

- ▶ dplyr is mainly authored by Hadley Wickham and Romain Francois. It is designed to be intuitive and easy to learn, thereby making “doing things” in R more user-friendly.
- ▶ dplyr is a new package which provides a set of tools for efficiently manipulating datasets in R.
- ▶ dplyr is the next iteration of plyr, focussing on only data frames.
- ▶ dplyr is faster, has a more consistent API and should be easier to use.

dplyr : abstract by Hadley Wickham

(Hadley Wickham's Abstract) There are three key ideas that underlie dplyr:

- 1 Your time is important, so Romain Francois has written the key pieces in Rcpp to provide blazing fast performance. Performance will only get better over time, especially once we figure out the best way to make the most of multiple processors.

dplyr : abstract by Hadley Wickham

- 2 Tabular data is tabular data regardless of where it lives, so you should use the same functions to work with it.
With dplyr, anything you can do to a local data frame you can also do to a remote database table.
PostgreSQL, MySQL, SQLite and Google bigquery support is built-in; adding a new backend is a matter of implementing a handful of S3 methods.

dplyr : abstract by Hadley Wickham

- 3 The bottleneck in most data analyses is the time it takes for you to figure out what to do with your data, and dplyr makes this easier by having individual functions that correspond to the most common operations (`group_by`, `summarise`, `mutate`, `filter`, `select` and `arrange`). Each function does one only thing, but does it well.

Working with dplyr

dplyr focussed on tools for working with data frames (hence the **d** in the name). **dplyr** has three main goals:

- ▶ Identify the most important data manipulation tools needed for data analysis and make them easy to use from R.
- ▶ Provide very fast performance for in-memory data by writing key pieces in C++.
- ▶ Use the same interface to work with data no matter where it's stored, whether in a data frame, a data table or database.

Tidy Data

- ▶ To make the most of dplyr, Hadley Wickham recommends that you familiarise yourself with the **principles of tidy data**.
- ▶ This will help you get your data into a form that works well with **dplyr**, **ggplot2** and R's many modelling functions.

Three Principles from Hadley Wickham's paper

1. Each variable forms a column,
2. Each observation forms a row,
3. Each table/file stores data about one kind of observation.

Remark:

The paper “**Tidy data**” by Hadley Wickham (RStudio) can be downloaded from

<http://vita.had.co.nz/papers/tidy-data.pdf>

Key data structures

The key object in **dplyr** is a `tbl`, a representation of a tabular data structure. Currently dplyr supports:

- ▶ data frames - the most commonly encountered R data structure.
- ▶ data tables - a data structure that is designed for intensive data analysis.

For advanced users, **dplyr** also supports the following databases:
SQLite, PostgreSQL, Redshift, MySQL/MariaDB, Bigquery, MonetDB and data cubes with arrays (partial implementation).
We will not cover those topics in this workshop.

Introduction to dplyr

2015-01-13

When working with data you must:

- Figure out what you want to do.
- Precisely describe what you want in the form of a computer program.
- Execute the code.

The dplyr package makes each of these steps as fast and easy as possible by:

- Elucidating the most common data manipulation operations, so that your options are helpfully constrained when thinking about how to tackle a problem.
- Providing simple functions that correspond to the most common data manipulation verbs, so that you can easily translate your thoughts into code.
- Using efficient data storage backends, so that you spend as little time waiting for the computer as possible.

Installing dplyr

Straightforward R package installation.

```
install.packages("dplyr")  
library(dplyr)
```

```
# Data Set Examples
```

```
# 1. iris
```

```
# 2. mtcars
```

iris data set

```
> names(iris)
[1] "Sepal.Length"
[2] "Sepal.Width"
[3] "Petal.Length"
[4] "Petal.Width"
[5] "Species"
```

mtcars data set

```
> names(mtcars)
[1] "mpg"  "cyl"  "disp" "hp"
[5] "drat" "wt"   "qsec" "vs"
[9] "am"   "gear" "carb"
```

Example Data Sets

```
dim(iris)  
class(iris)  
mode(iris)
```

```
dim(mtcars)  
class(mtcars)  
mode(mtcars)
```

dplyr; Single Table Verbs

dplyr aims to provide a function for each basic verb of data manipulating:

- ▶ `filter()` (and `slice()`)
- ▶ `arrange()`
- ▶ `select()` (and `rename()`)
- ▶ `distinct()`
- ▶ `mutate()` (and `transmute()`)
- ▶ `summarise()`
- ▶ `sample_n()` and `sample_frac()`

Grouped operations

- ▶ The verbs are useful, but they become really powerful when you combine them with the idea of group by, repeating the operation individually on groups of observations within the dataset.
- ▶ In dplyr, you use the `group_by()` function to describe how to break a dataset down into groups of rows.
- ▶ You can then use the resulting object in exactly the same functions as above; they'll automatically work by group when the input is a grouped.

Summary Statistics

You use `summarise()` with aggregate functions, which take a vector of values, and return a single number.

There are many useful functions in base R like `min()`, `max()`, `mean()`, `sum()`, `sd()`, `median()`, and `IQR()`.

`dplyr` provides a handful of others:

- ▶ `n()`: number of observations in the current group
- ▶ `n_distinct(x)`: count the number of unique values in `x`.

Grouping with the group_by command

```
iris.sp <- group_by(iris,Species)
class(iris.sp)

summarise(iris.sp,mean(Sepal.Length), sd(Petal.Length))
```

```
> iris.sp <- group_by(iris, Species)
> class(iris.sp)
[1] "grouped_df" "tbl_df"      "tbl"        "data.frame"
>
> summarise(iris.sp, mean(Sepal.Length), sd(Petal.Length))
Source: local data frame [3 x 3]
```

	Species	mean(Sepal.Length)	sd(Petal.Length)
1	setosa	5.006	0.1736640
2	versicolor	5.936	0.4699110
3	virginica	6.588	0.5518947

```
>
> |
```

```
> summarise(mtcars2, mean(mpg), sd(mpg))  
Source: local data frame [6 x 4]  
Groups: cyl
```

	cyl	am	mean(mpg)	sd(mpg)
1	4	0	22.90000	1.4525839
2	4	1	28.07500	4.4838599
3	6	0	19.12500	1.6317169
4	6	1	20.56667	0.7505553
5	8	0	15.05000	2.7743959
6	8	1	15.40000	0.5656854

```
> |
```

Filter rows with `filter()`

- ▶ `filter()` allows you to select a subset of the rows of a data frame.
- ▶ The first argument is the name of the data frame, and the second and subsequent are filtering expressions evaluated in the context of that data frame.

```
iris.vir1 <- filter(iris,Species=="virginica")  
  
# Species is Virginica OR Petal.length is greater than 3.  
  
iris.vir2 <- filter(iris,Species=="virginica" | Petal.Length > 3)  
  
iris.vir3 <- filter(iris,Species=="virginica" & Petal.Length > 3)
```

Ordering Data Sets with `arrange()`

- ▶ `arrange()` works similarly to `filter()` except that instead of filtering or selecting rows, it reorders them.
- ▶ It takes a data frame, and a set of column names (or more complicated expressions) to order by.
- ▶ If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns.
- ▶ Use `desc()` (or `rev()`) to order a column in descending order:

```

>
> arrange(iris, Petal.Length, Petal.Width)

```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	4.6	3.6	1.0	0.2	setosa
2	4.3	3.0	1.1	0.1	setosa
3	5.8	4.0	1.2	0.2	setosa
4	5.0	3.2	1.2	0.2	setosa
5	4.7	3.2	1.3	0.2	setosa
6	5.5	3.5	1.3	0.2	setosa
7	4.4	3.0	1.3	0.2	setosa
8	4.4	3.2	1.3	0.2	setosa
9	5.0	3.5	1.3	0.3	setosa
10	4.5	2.3	1.3	0.3	setosa
11	5.4	3.9	1.3	0.4	setosa
12	4.8	3.0	1.4	0.1	setosa
13	4.9	3.6	1.4	0.1	setosa
14	5.1	3.5	1.4	0.2	setosa
15	4.9	3.0	1.4	0.2	setosa
16	5.0	3.6	1.4	0.2	setosa
17	4.4	2.9	1.4	0.2	setosa
18	5.2	3.4	1.4	0.2	setosa
19	5.5	4.2	1.4	0.2	setosa


```
> arrange(iris, Petal.Length, rev(Petal.Width))
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	4.6	3.6	1.0	0.2	setosa
2	4.3	3.0	1.1	0.1	setosa
3	5.8	4.0	1.2	0.2	setosa
4	5.0	3.2	1.2	0.2	setosa
5	5.4	3.9	1.3	0.4	setosa
6	4.5	2.3	1.3	0.3	setosa
7	4.4	3.2	1.3	0.2	setosa
8	4.4	3.0	1.3	0.2	setosa
9	4.7	3.2	1.3	0.2	setosa
10	5.5	3.5	1.3	0.2	setosa
11	5.0	3.5	1.3	0.3	setosa
12	5.1	3.5	1.4	0.2	setosa
13	4.8	3.0	1.4	0.1	setosa

Select columns with `select()`

- ▶ Often you work with large datasets with many columns where only a few are actually of interest to you.
- ▶ `select()` allows you to rapidly zoom in on a useful subset using operations that usually only work on numeric variable positions.

Selection Options with `select()`

- `starts_with(x, ignore.case = TRUE)`: names starts with `x`
- `ends_with(x, ignore.case = TRUE)`: names ends in `x`
- `contains(x, ignore.case = TRUE)`: selects all variables whose name contains `x`
- `matches(x, ignore.case = TRUE)`: selects all variables whose name matches the regular expression `x`
- `num_range("x", 1:5, width = 2)`: selects all variables (numerically) from `x01` to `x05`.
- `one_of("x", "y", "z")`: selects variables provided in a character vector.
- `everything()`: selects all variables.

Selection Options with select()

```
> select(iris, ends_with("width"))
```

	Sepal.Width	Petal.Width
1	3.5	0.2
2	3.0	0.2
3	3.2	0.2
4	3.1	0.2
5	3.6	0.2
6	3.9	0.4
7	3.4	0.3
8	3.4	0.2
9	2.9	0.2
10	3.1	0.1
11	3.7	0.2
12	3.4	0.2
13	3.0	0.1

Selection Options with select()

```
> select(iris, contains("etal"))
```

	Petal.Length	Petal.Width
1	1.4	0.2
2	1.4	0.2
3	1.3	0.2
4	1.5	0.2
5	1.4	0.2
6	1.7	0.4
7	1.4	0.3
8	1.5	0.2
9	1.4	0.2

Sampling rows with `sample_n()` and `sample_frac()`

- ▶ You can use `sample_n()` and `sample_frac()` to take a random sample of rows, either a fixed number for `sample_n()` or a fixed fraction for `sample_frac()`.

Sampling rows with `sample_n()` and `sample_frac()`

```
> sample_n(iris, 10)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
12	4.8	3.4	1.6	0.2	setosa
86	6.0	3.4	4.5	1.6	versicolor
137	6.3	3.4	5.6	2.4	virginica
94	5.0	2.3	3.3	1.0	versicolor
56	5.7	2.8	4.5	1.3	versicolor
36	5.0	3.2	1.2	0.2	setosa
134	6.3	2.8	5.1	1.5	virginica
7	4.6	3.4	1.4	0.3	setosa
35	4.9	3.1	1.5	0.2	setosa
15	5.8	4.0	1.2	0.2	setosa

```
> |
```

Sampling rows with `sample_n()` and `sample_frac()`

```
> sample_frac(iris,0.08)
      Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
73          6.3         2.5         4.9         1.5 versicolor
51          7.0         3.2         4.7         1.4 versicolor
24          5.1         3.3         1.7         0.5   setosa
139         6.0         3.0         4.8         1.8 virginica
15          5.8         4.0         1.2         0.2   setosa
33          5.2         4.1         1.5         0.1   setosa
62          5.9         3.0         4.2         1.5 versicolor
94          5.0         2.3         3.3         1.0 versicolor
30          4.7         3.2         1.6         0.2   setosa
22          5.1         3.7         1.5         0.4   setosa
39          4.4         3.0         1.3         0.2   setosa
58          4.9         2.4         3.3         1.0 versicolor
> |
```


Sampling rows with `sample_n()` and `sample_frac()`

```
> sample_frac(iris.sp,0.08)
```

```
Source: local data frame [12 x 5]
```

```
Groups: Species
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	4.9	3.6	1.4	0.1	setosa
2	5.1	3.8	1.5	0.3	setosa
3	5.0	3.0	1.6	0.2	setosa
4	5.2	4.1	1.5	0.1	setosa
5	5.5	2.5	4.0	1.3	versicolor
6	6.4	3.2	4.5	1.5	versicolor
7	6.0	3.4	4.5	1.6	versicolor
8	5.5	2.4	3.7	1.0	versicolor
9	7.7	3.0	6.1	2.3	virginica
10	6.1	2.6	5.6	1.4	virginica
11	4.9	2.5	4.5	1.7	virginica
12	6.8	3.0	5.5	2.1	virginica

```
> |
```

Add new columns with mutate()

As well as selecting from the set of existing columns, its often useful to add new columns that are functions of existing columns. This is the job of `mutate()`:

```
iris2 = mutate(iris,  
  PW2 = log(Petal.Width),  
  PL2=sqrt(Petal.Length) )  
  
head(iris2)
```

Add new columns with mutate()

```
> iris2 = mutate(iris, PW2 = log(Petal.Width), PL2=sqrt(Petal.Length) )
> head(iris2)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species	PW2	PL2
1	5.1	3.5	1.4	0.2	setosa	-1.6094379	1.183216
2	4.9	3.0	1.4	0.2	setosa	-1.6094379	1.183216
3	4.7	3.2	1.3	0.2	setosa	-1.6094379	1.140175
4	4.6	3.1	1.5	0.2	setosa	-1.6094379	1.224745
5	5.0	3.6	1.4	0.2	setosa	-1.6094379	1.183216
6	5.4	3.9	1.7	0.4	setosa	-0.9162907	1.303840

```
> |
```

Add new columns with mutate()

mutate allows you to refer to columns that you just created:

```
iris3 = mutate(iris,  
  PW2 = log(Petal.Width),  
  PL2=sqrt(Petal.Length),  
  Ratio=PL2/PW2 )
```

```
head(iris3)
```

Add new columns with mutate()

```
1      1.0      3.1      1.3      0.2
5      5.0      3.6      1.4      0.2
6      5.4      3.9      1.7      0.4
```

	Species	PW2	PL2	Ratio
1	setosa	-1.6094379	1.183216	-0.7351734
2	setosa	-1.6094379	1.183216	-0.7351734
3	setosa	-1.6094379	1.140175	-0.7084308
4	setosa	-1.6094379	1.224745	-0.7609768
5	setosa	-1.6094379	1.183216	-0.7351734
6	setosa	-0.9162907	1.303840	-1.4229550

```
> |
```

<

Multiple table verbs

As well as verbs that work on a single `tbl`, there are also a set of useful verbs that work with two `tbls` at a time: joins and set operations.

Joins

dplyr implements the four most useful joins from SQL:

- ▶ `inner_join(x, y)`: matching $x + y$
- ▶ `left_join(x, y)`: all $x +$ matching y
- ▶ `semi_join(x, y)`: all x with match in y
- ▶ `anti_join(x, y)`: all x without match in y

Set Theory Operations

dplyr implements the methods for set theory operations

- ▶ `intersect(x, y)`: all rows in both x and y
- ▶ `union(x, y)`: rows in either x or y
- ▶ `setdiff(x, y)`: rows in x, but not y