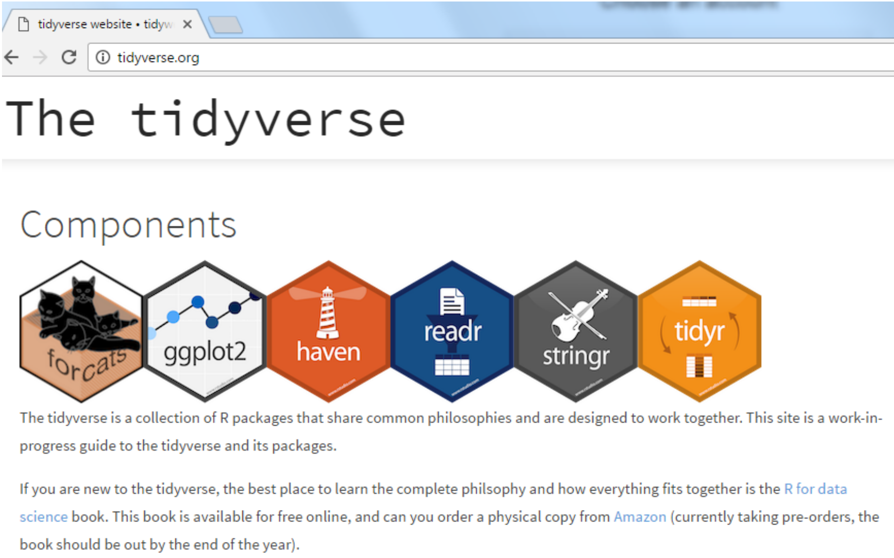
Kevin O'Brien

- Python Ireland and DublinR organizer
- University of Limerick, Ireland
  - Teach `R` in various undergraduate courses
- Previously: Statistical Progammer at ESRI
  - **Growing Up In Ireland** - Cohort Study
  - Statistical Disclosure Control

tidyverse.org



tidyverse website • tidyv

tidyverse.org

The tidyverse

Components

The tidyverse is a collection of R packages that share common philosophies and are designed to work together. This site is a work-in-progress guide to the tidyverse and its packages.

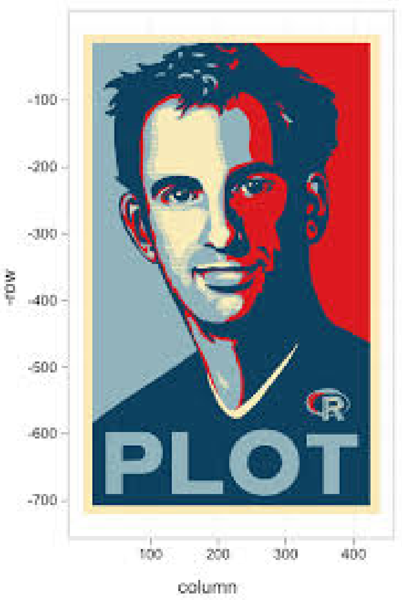
If you are new to the tidyverse, the best place to learn the complete philsophy and how everything fits together is the R for data science book. This book is available for free online, and can you order a physical copy from Amazon (currently taking pre-orders, the book should be out by the end of the year).

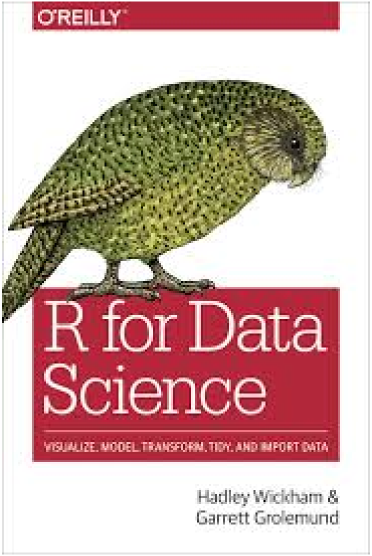
Tidyverse

- The tidyverse is a collection of R packages that share common philosophies and are designed to work together.
- This site is a work-in-progress guide to the tidyverse and its packages.
- If you are new to the tidyverse, the best place to learn the complete philsophy and how everything fits together is the R for data science book.
- This book is available for free online, and can you order a physical copy from Amazon (currently taking pre-orders, the book should be out by the end of the year).

(from Tidyverse Website)

Hadley Wickham




O'REILLY
R for Data Science
VISUALIZE, MODEL, TRANSFORM, TIDY, AND IMPORT DATA
Hadley Wickham &
Garrett Grolemund

Vignettes

NeedsCompilation:	yes
Materials:	README NEWS
CRAN checks:	tidyr results

Downloads:

Reference manual:	tidyr.pdf
Vignettes:	Tidy data
Package source:	tidyr_0.6.1.tar.gz
Windows binaries:	r-devel: tidyr_0.6.1.zip, r-release: tidyr_0.6.1.zip, r-oldrel:
OS X Mavericks binaries:	r-release: tidyr_0.6.1.tgz, r-oldrel: tidyr_0.6.1.tgz
Old sources:	tidyr archive

Tidverse Principles

There are four basic principles for Tidyverse programming:

- Reuse existing data structures.
- Compose simple functions with the pipe. (Important!)
- Embrace functional programming.
- **Design for humans**. (i.e. Undergrads!)

Two Key Parts of Tidyverse

- **dplyr** - data manipulation
- **magrittr** - pipe operator

THE %>% OPERATOR

%>%
magrittr

Ceci n'est pas un pipe.

The $\% > \%$ operator

- From **magrittr** package.
- Used extensively in **dplyr**.
- $\% > \%$ is a piping operator, and can be verbalised as "*then*".
- It takes the output of the left side, and uses it as the first argument of the function on the right side.

magrittr : the % > % operator

```
subset(mtcars, cyl == 6, c(mpg, wt))

mtcars %>% subset(cyl == 6, c(mpg, wt))
```

R Console

```
>   mtcars %>% subset(cyl == 6, c(mpg, wt))
                mpg    wt
Mazda RX4      21.0 2.620
Mazda RX4 Wag  21.0 2.875
Hornet 4 Drive 21.4 3.215
Valiant        18.1 3.460
Merc 280       19.2 3.440
Merc 280C      17.8 3.440
Ferrari Dino   19.7 2.770
> 
```

magrittr : the %>% operator

```
summary(subset(mtcars, cyl == 6,
c(mpg, wt)), digits=2)

mtcars %>%
subset(cyl == 6, c(mpg, wt)) %>%
summary(digits=2)
```

magrittr : the $\% > \%$ operator

```
mtcars %>%
subset(cyl == 6, c(mpg, wt)) %>%
summary(digits=2)
```

- Get the mtcars data set
- **Then** subset it like this
- **Then** get the summary, with this setting

```
> mtcars %>%
+     subset(cyl == 6, c(mpg, wt)) %>%
+         summary(digits=2)
      mpg              wt
 Min.   :18    Min.   :2.6
 1st Qu.:19    1st Qu.:2.8
 Median :20    Median :3.2
 Mean   :20    Mean   :3.1
 3rd Qu.:21    3rd Qu.:3.4
 Max.   :21    Max.   :3.5
>
```

dplyr

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.

(from Hadley Wickham's Vignette)

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.

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.

3 The bottleneck in most data analyses is the time it takes for you to figure out what to do with your data

dplyr makes this easier by having individual functions that correspond to the most common operations.

Functions include `group_by()`, `glimpse()` and the single table verbs. We will have a look at these verbs that we shall see shortly.

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.

Single table verbs

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

(Also `group_by()` and `glimpse()`)

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.

Tidy Data

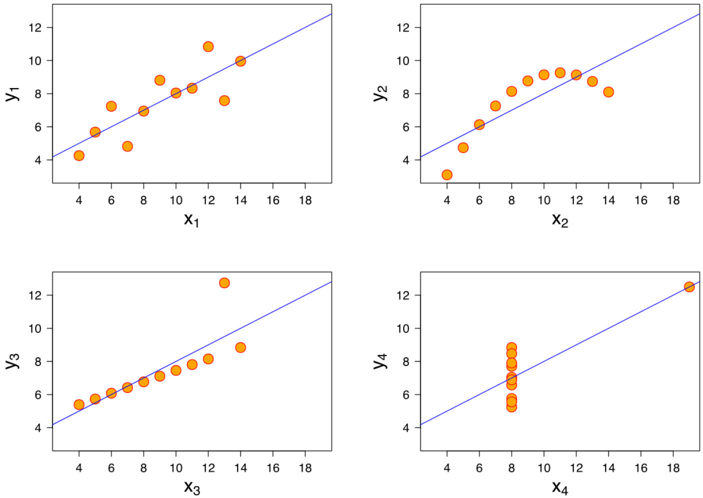
Three Principles from Hadley Wickham's paper

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`

Anscombe Quartet



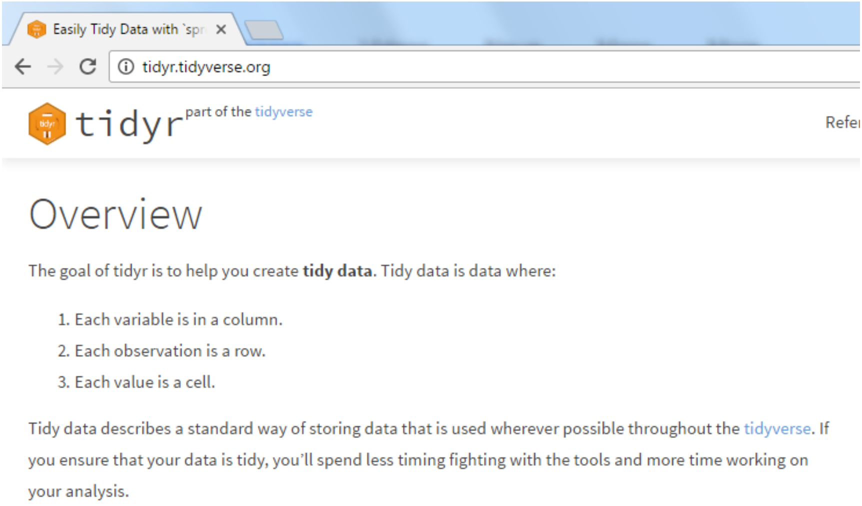
Key data structures

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.


forcats
ggplot2
haven
readr
stringr
tidyr

tidyr.tidyverse.org



Easily Tidy Data with `spr

tidyr.tidyverse.org

tidyr part of the tidyverse

Refe

Overview

The goal of tidyr is to help you create **tidy data**. Tidy data is data where:

1. Each variable is in a column.
2. Each observation is a row.
3. Each value is a cell.

Tidy data describes a standard way of storing data that is used wherever possible throughout the tidyverse. If you ensure that your data is tidy, you'll spend less timing fighting with the tools and more time working on your analysis.

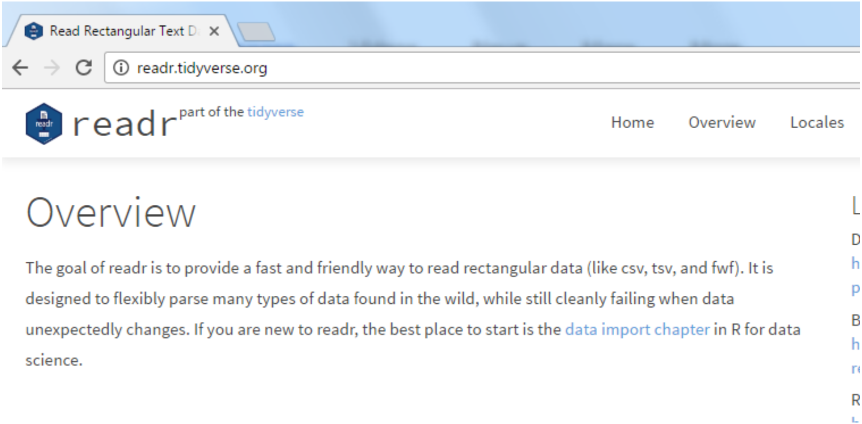
Wide to Long

ID	T	P.1	P.2	P.3
1	24.3	10.2	5.5	2.1
2	23.4	10.4	5.7	2.8
3	22.1	10.5	5.9	3.1
4	19.9	10.2	5.2	2.4

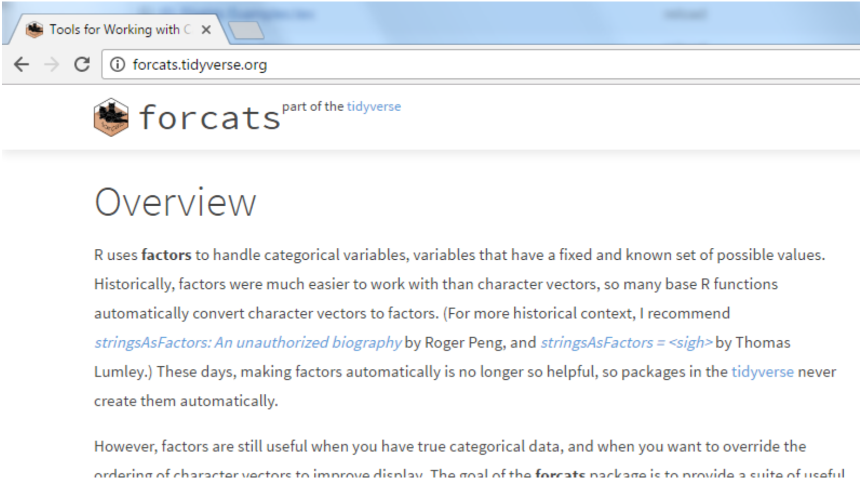
ID	Channel	T	P
1	1	24.3	10.2
2	1	23.4	10.4
3	1	22.1	10.5
4	1	19.9	10.2
1	2	24.3	5.5
2	2	23.4	5.7
3	2	22.1	5.9
4	2	19.9	5.2
1	3	24.3	2.1
2	3	23.4	2.8
3	3	22.1	3.1
4	3	19.9	2.4

www.StackOverflow.com

readr.tidyverse.org



forcats.tidyverse.org



Tools for Working with C

forcats.tidyverse.org

forcats part of the tidyverse

Overview

R uses **factors** to handle categorical variables, variables that have a fixed and known set of possible values. Historically, factors were much easier to work with than character vectors, so many base R functions automatically convert character vectors to factors. (For more historical context, I recommend *stringsAsFactors: An unauthorized biography* by Roger Peng, and *stringsAsFactors = <sigh>* by Thomas Lumley.) These days, making factors automatically is no longer so helpful, so packages in the tidyverse never create them automatically.

However, factors are still useful when you have true categorical data, and when you want to override the ordering of character vectors to improve display. The goal of the **forcats** package is to provide a suite of useful

haven.tidyverse.org

Import and Export 'SPSS'

haven.tidyverse.org

haven part of the tidyverse

Overview

Haven allows is designed to write and read data formats used by other statistical packages by wrapping the fantastic ReadStat C library written by Evan Miller. Currently it supports:

- **SAS**: `read_sas()` reads `.sas7bdat` + `.sas7bcat` files and `read_xpt()` SAS transport files (version 5 and version 8). `write_sas()` writes `.sas7bdat` files.
- **SPSS**: `read_sav()` reads `.sav` files and `read_por()` reads the older `.por` files. `write_sav()` writes `.sav` files.
- **Stata**: `read_dta()` reads `.dta` files (up to version 14). `write_dta()` writes `.dta` files (versions 8-14).

The output objects:

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)
```

Grouped operations

- 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; theyll automatically work "by group" when the input is a grouped structure.

`group_by`:

Group a *tbl* by one or more variables.

Description

Most data operations are useful done on groups defined by variables in the the dataset.
The **group_by** function takes an existing tbl and converts it into a grouped tbl where operations are performed "by group".

The `glimpse()` Function

- **dplyr** also provides a function `glimpse()` that makes it easy to look at the data in a transposed view.
- similar to the `str()` (structure) function, but has a few advantages.

The glimpse() Function

```
> glimpse(iris)
Variables:
$ Sepal.Length (dbl) 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4
$ Sepal.Width  (dbl) 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3
$ Petal.Length (dbl) 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1
$ Petal.Width  (dbl) 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0
$ Species      (fctr) setosa, setosa, setosa, setosa
> |
```

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()`

Summary Statistics

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

Supports functions in base R like `min()`, `max()`, `mean()`, `sum()`, `sd()`, `median()`, and `IQR()`.

- `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,
  meanSL = mean(Sepal.Length),
  sdSL = sd(Petal.Length))
```

(Remark: This breaks a tidyverse principle, use pipe instead)

```
> 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 3.2

iris.vir2 <- filter(iris,
Species=="virginica" | Petal.Length >3.2)

iris.vir3 <- filter(iris,
Species=="virginica" & Petal.Length >3.9)
```

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 target 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 contain
- `matches(x, ignore.case = TRUE)`: selects all variables whose name matches expression x
- `num_range("x", 1:5, width = 2)`: selects all variables (numerically) from x0
- `one_of("x", "y", "z")`: selects variables provided in a character vector.
- `everything()`: selects all variables.

Selection Options with `select()`

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

The Idaho Data Set

```
> glimpse(idaho)
Variables:
$ RT       (fctr) H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H,
$ SERIALNO (int) 186, 306, 395, 506, 835, 989, 1861, 2120, 2278, 242
$ DIVISION (int) 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
$ PUMA     (int) 700, 700, 100, 700, 800, 700, 700, 200, 400, 500, 4
$ REGION   (int) 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
$ ST       (int) 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16,
$ ADJUST   (int) 1015675, 1015675, 1015675, 1015675, 1015675, 101567
$ WGTP     (int) 89, 310, 106, 240, 118, 115, 0, 35, 47, 51, 114, 51
$ NP       (int) 4, 1, 2, 4, 4, 4, 1, 1, 2, 2, 2, 2, 3, 1, 1, 2, 3,
$ TYPE     (int) 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1,
$ ACR      (int) 1, NA, 1, 1, 2, 1, NA, 1, 1, 1, 1, 1, 1, NA, 1, 1,
$ AGS      (int) NA, NA, NA, NA, 1, NA, NA, NA, NA, NA, NA, NA, NA,
$ BDS      (int) 4, 1, 3, 4, 5, 3, NA, 2, 3, 2, 3, 3, 2, NA, 3, 4, 5
$ BLD      (int) 2, 7, 2, 2, 2, 2, NA, 1, 2, 1, 2, 2, 2, NA, 1, 2, 2
$ BUS      (int) 2, NA, 2, 2, 2, 2, NA, 2, 2, 2, 2, 2, 2, NA, 2, 2,
$ CONP     (int) NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA,
$ ELEP     (int) 180, 60, 70, 40, 250, 130, NA, 40, 2, 20, 50, 100,
$ FS       (int) 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
$ FULP     (int) 2, 2, 2, 2, 2, 2, NA, 480, 2, 2, 2, 2, 600, NA, 2,
```

Idaho Data Set: Multiple Selections with `select()`

Remark : Regular Expressions.

```
idaho2 = select(idaho,
contains("AX"),
starts_with("FK"),
starts_with("SM"),
ends_with("SP")
)
```

Dropping Variables with `select()`

```
# Drop variables
select(iris, -starts_with("Petal"))
select(iris, -ends_with("Width"))
select(iris, -contains("etal"))
select(iris, -matches(".t."))
select(iris, -Petal.Length, -Petal.Width)
```

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
```

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
1          5.1         3.5          1.4         0.2  setosa -1.6094379 1.1
2          4.9         3.0          1.4         0.2  setosa -1.6094379 1.1
3          4.7         3.2          1.3         0.2  setosa -1.6094379 1.1
4          4.6         3.1          1.5         0.2  setosa -1.6094379 1.2
5          5.0         3.6          1.4         0.2  setosa -1.6094379 1.1
6          5.4         3.9          1.7         0.4  setosa -0.9162907 1.3
> 
```

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()

```
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

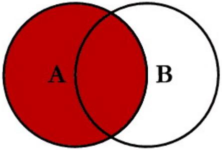
As well as verbs that work on a single `tbl`, there are also a set of useful verbs that work with two `tbls` at a time: joins and set operations.

Joins

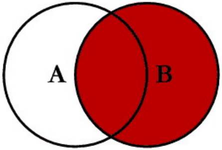
dplyr implements the four most useful joins from SQL:

- `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

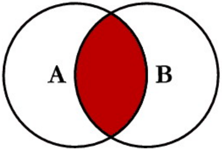
SQL JOINS



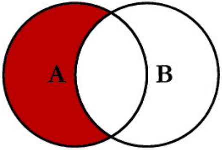

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



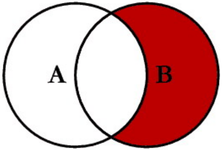

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



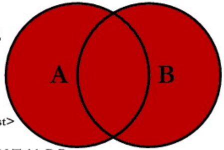

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



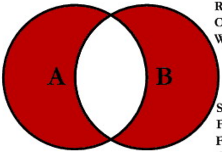

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

Joins

Pretend data set listing country of origin for each species. The variables “Species” is common to both data frames.

```
> irisnewdata
     Species   origin
1  virginica  Ireland
2     setosa Scotland
3 versicolor    Wales
> 
```

Figure: Second Data Frame

Joins

```
>
>
> irisnewdata = data.frame(Species=c("virginica","setosa","versicolor"),origin=c(
>
>
> left_join(iris,irisnewdata)
Joining by: "Species"
        Species Sepal.Length Sepal.Width Petal.Length Petal.Width   origin
1        setosa          5.1         3.5          1.4         0.2 Scotland
2        setosa          4.9         3.0          1.4         0.2 Scotland
3        setosa          4.7         3.2          1.3         0.2 Scotland
4        setosa          4.6         3.1          1.5         0.2 Scotland
5        setosa          5.0         3.6          1.4         0.2 Scotland
6        setosa          5.4         3.9          1.7         0.4 Scotland
7        setosa          4.6         3.4          1.4         0.3 Scotland
8        setosa          5.0         3.4          1.5         0.2 Scotland
```

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