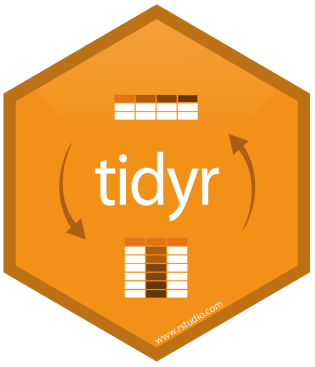
dplyr

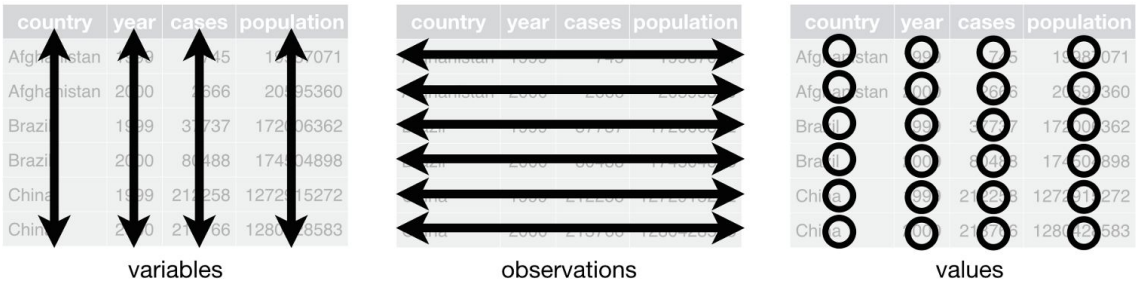
a quick overview

Rabindra Khadka
Andrea Lorenzon

Introduction: tidyverse basics[1]



Tidyverse is “an opinionated collection of R packages designed for data science”, uniformed by philosophy, grammar and data structures. One of the main shared concepts is the one of **tidy data**, seen as a standard way of organising data inside a dataset.
Tidy data is designed to facilitate exploration and analysis, and simplify the development of analysis tools that just work well together, without translations and custom interfaces, thus reducing time wasted on data logistics, in favor of focusing on interesting domain problems.



Tidy data structure.

In tidy data:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

as per Codd’s third normal form[2]. Any other data format is thus known as **messy data**. In messy data, for example:

1. Column headers are values, not variable names.
2. Multiple variables are stored in one column.
3. Variables are stored in both rows and columns.
4. Multiple types of observational units are stored in the same table.
5. A single observational unit is stored in multiple tables.

---

[1] As convention, in this essay, all functions coming from `dplyr, magrittr and tibble` packaged are just called by name `function()`. Every function from other packages is usually called by `package::function()`

[2] An approximation of Codd's definition of 3NF, paralleling the traditional pledge to give true evidence in a court of law, was given by Bill Kent: "*[Every] non-key [attribute] must provide a fact about the key, the whole key, and nothing but the key.*" A common variation supplements this definition with the oath: "*so help me Codd*".

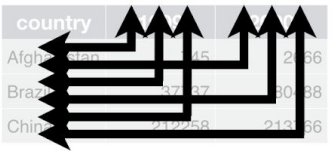
country	1999	2000
Afghanistan	745	2666
Brazil	37737	80488
China	212258	213766

table4

country	1999	2000
Afghanistan	19987071	20595360
Brazil	172006362	174504898
China	1272915272	1280428583

table5

variables



observations

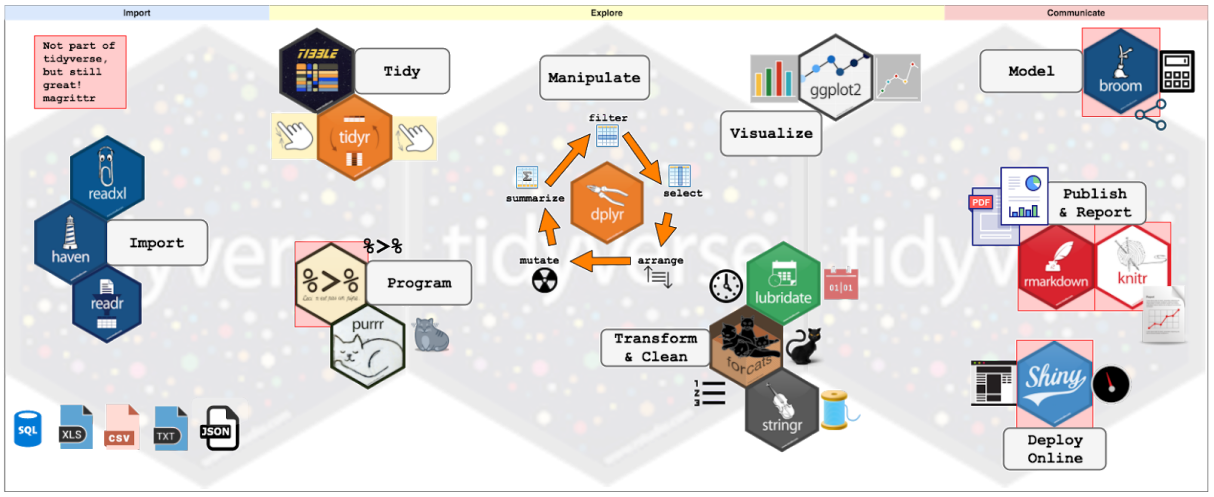
Messy data.

To achieve this result, tidyverse provides an upgraded version of data frames format called **tibble**, using the `tbl_df` class, and the library `tidyr`. This format keeps the features of R data frames that works, and drops the problematic ones. The main differences are:

1. tibbles never changes input types (i.e., no more `stringsAsFactors = FALSE!`)
2. they never adjust the names of variables
3. they evaluate their arguments lazily and sequentially. e.g. `tibble(x = 1:5, y = x ^ 2)` returns a column of numbers and a column of their squares.
4. neat printing, with column types
5. coherent subsetting: subsetting a tibble always return a tibble. With dataframes, it can sometimes output vectors or other types.

`dplyr` (with `tidyr`)

As we will see, `dplyr` is not meant to be a standalone package. It is merged with all the tidyverse packages.

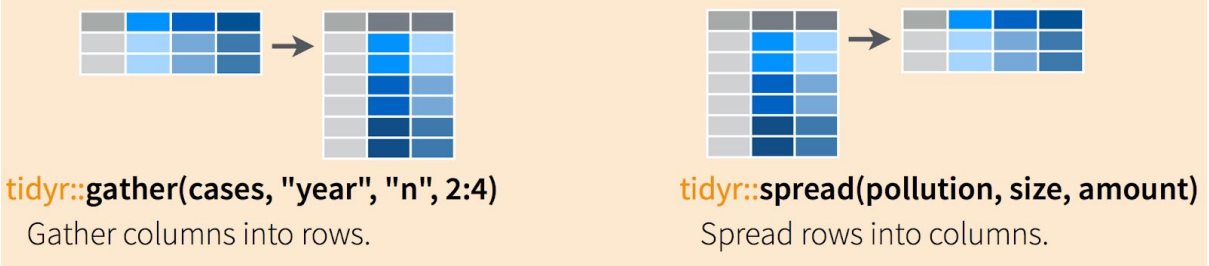


The Tidyverse: a collection of synergic packages to help data import, wrangling, exploration and communication.

`dplyr` automatically import `magrittr` and `tibble` packages, respectively for piping and tibble data formats.

Using `dplyr` and `tidyr` to handle a dataframe

Being part of the Tidyverse, dplyr integrates with other packages to perform quick operations on table datasets. In this section we will see some useful tool to reorganize data in a tibble.



images from https://jules32.github.io/2016-07-12-Oxford/dplyr_tidyr/

`tidyr::gather()` and `tidyr::spread()`

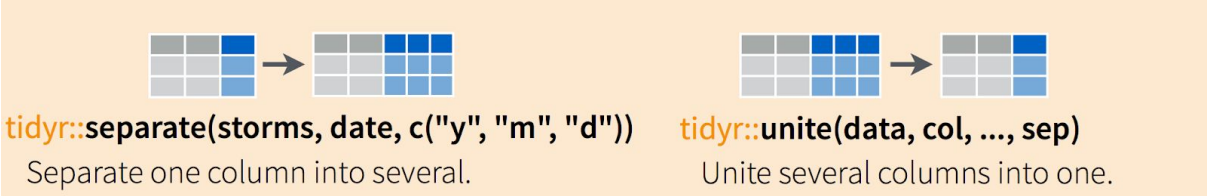
`gather()` takes multiple values, and collapse them in key-value pairs, making “wide” tables longer.
`spread()` does the opposite, taking 2 pair-value columns and spreading them in multiple columns.
Similar operations can be found in different context, with different names. In the following table a comparison of such diverse nomenclature.

tidyr	gather	spread

reshape	`melt`	`cast`
spreadsheets	`unpivot`	`pivot`
databases	`fold`	`unfold`

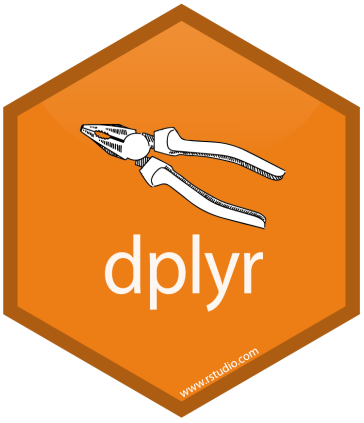
Correspondence of concepts between tidyverse, reshape, spreadsheets and databases operation nomenclature.

`tidyr::separate()` and `tidyr::unite()`



These two functions are useful to respectively subdivide a column in more columns, by specifying a separating substring, and uniting different columns by concatenating their content.

`dplyr` verbs: the “golden six”



The main core of `dplyr` are six functions, often called “verbs”, implying tibbles to be “names”, thus building a grammar for data manipulation. Let’s have quick overview of these functions.

`filter()`

This function is used to choose rows/cases where conditions are true. Unlike base subsetting with [, rows where the condition evaluates to NA are dropped. Like all single verbs, the first argument is the tibble (or data frame). The second and subsequent arguments refer to variables within that data frame, selecting rows where the expression is TRUE.

ex.

```
filter(starwars, species == "Human")
```

`group_by()`

This function takes an existing tibble and converts it into a grouped tibble where operations are performed "by group". `ungroup()` removes grouping.

ex.

```
by_cyl <- mtcars %>% group_by(cyl)
```

`select()`

This function subset a tibble by columns, keeping only the variable(s) you mention in the arguments. As well as using existing R functions like `:` and `c()`, there are a number of special functions that only work inside `select`

- `starts_with(), ends_with(), contains()`
- `matches()`
- `num_range()`
- `one_of()`
- `everything()`

To drop variables, use "`-`". Note that except for `:`, `-` and `c()`, all complex expressions are evaluated outside the data frame context. This is to prevent accidental matching of data frame variables when you refer to variables from the calling context.

ex.

```
select(iris, starts_with("Petal"))
```

There is a similar function, named `dplyr::pull()`, that can be used to extract raw column content (vectors) from a selection, instead of a tibble.

`arrange()`

This function simply sort values of its argument, by one or more columns. By default, this is done in ascending order, a different order can be specified in arguments.

ex.

```
arrange(mtcars, desc(disp))
```

`mutate()` / `transmute()`

`mutate()` adds new variables and preserves existing ones; `transmute()` adds new variables and drops existing ones. Both functions preserve the number of rows of the input. New variables overwrite existing variables of the same name.

When working on grouped tibbles, remember that all mutate/transmute operations are done *within* groups.

ex.

```
mtcars %>% as_tibble() %>% mutate(
  mpg = NULL,              # drop column
  disp = disp * 0.0163871 # convert to litres, overwriting
)
```

`summarise()`[3]

Create one or more scalar variables summarizing the variables of an existing tbl. Tibbles with groups created by `group_by()` will result in one row in the output for each group.

Useful functions

- Center: `mean(), median()`
- Spread: `sd(), IQR(), mad()`
- Range: `min(), max(), quantile()`
- Position: `first(), last(), nth(),`
- Count: `n(), n_distinct()`
- Logical: `any(), all()`

Window functions

Useful functions available in calculations of variables, that can be used as arguments:

- `+, -, log(), etc.,` for their usual mathematical meanings
- `lead(), lag()`
- `dense_rank(), min_rank(), percent_rank(), row_number(), cume_dist(), ntile()`
- `cumsum(), cummean(), cummin(), cummax(), cumany(), cumall()`
- `na_if(), coalesce()`
- `if_else(), recode(), case_when()`

scoped aggregated functions: `_at` , `_if` , `_all`

Scoped function are written as `X_Y()`, where `X : {arrange, distinct, filter, group_by, mutate, select, and summarise}` and `Y : {at, if, all}`

Adding one of this suffixes modify the function by specifying their target. In detail:

[3] To satisfy both UK and US, `summarize()` and `summarise()` are synonyms.

`_all`: do something to all variables;
`_if`: do something to a set of variables selected from their characteristics. Instead of a predicate function, you can also supply a logical vector;
`_at`: do something to a set of variables selected from their variable name specified with the quoting function `vars()`.

It's impossible to consider `dplyr` out of the tidyverse: `magrittr` and piping in `dplyr`

Piping is the process of chaining functions in R. If you have $f: A \to B$ and $g: B \to C$, chaining together these functions means taking the output of f and using it as input of g. Mathematically speaking, it means computing $f \circ g$, or $f(g(x))$. Similar processes can be found in other languages. In Bash, the pipe operator " | " is used in the same way. Similar tools can be found in Haskell (" `(.)` ") and F# (" `|>` ").

The history of piping in R can be traced back to this Stackoverflow post: (https://stackoverflow.com/questions/8896820/how-to-implement-fs-forward-pipe-operator-in-r), when someone asked if a piping system, inspired from F#, could be implemented in R. The first reply came from prof. Ben Bolker (McMaster University), who proposed a short syntax:

```
"%>%" <- function(x,f) do.call(f,list(x))
```

9 months later `dplyr` was proposed, but just in 2013 the `magrittr` package implemented piping as we use it. As often in R, sometimes we find inconsistencies between different libraries: in `ggplot2`, the symbol "+" is still used for piping.

Additional piping operators:

Beyond `%>%`, other less known operators can be used:

- *The compound assignment operator `%<>%`*
  After computing the pipe, it assign the computed value back to the original variable.
- *The Tee operator `%T>%`*
  It returns the left-hand side value rather than the potential result of the right-hand side operations, effectively branching the pipe.
- *The exposition pipe operator `%$%`*
  It exposes function variables to the next function in the pipe.

do() in verbs

This is a general purpose complement to the specialised manipulation functions `filter()`, `select()`, `mutate()`, `summarise()` and `arrange()`. You can use `do()` to perform arbitrary computation, returning either a data frame or arbitrary objects which will be stored in a list. This is particularly useful when working with models: you can fit models per group with `do()` and then flexibly extract components with either another `do()` or `summarise()`. For an empty data frame, the expressions will be evaluated once, even in the presence of a grouping. This makes sure that the format of the resulting data frame is the same for both empty and non-empty input.

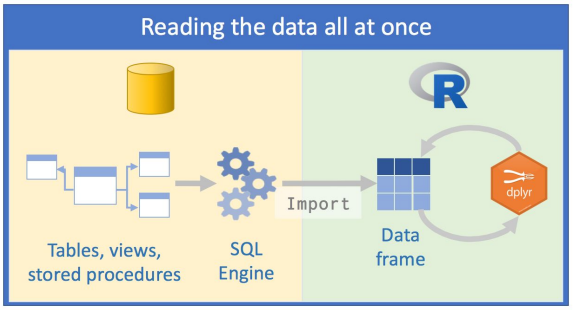
Ex.

```
models <- by_cyl %>% do(mod = lm(mpg ~ disp, data = .))
```

Using `dplyr` grammar on SQL databases and data.tables

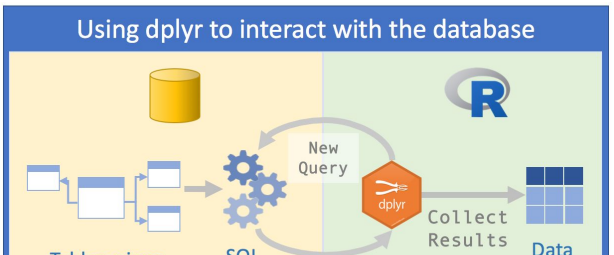
`dplyr` is designed to abstract over how the data is stored. That means as well as working with local data frames, you can also work with remote database tables, using exactly the same R code. Install the `dbplyr` package, and R will produce the SQL queries needed to perform those operations. With `dtplyr` instead we can extend our capabilities to data.tables, a popular and fast data format.
Without `dplyr`, when connecting to an external database, all the tables used to be pulled in R in a single, huge chunk, then usually analyzed with `stats::aggregate()`. Connecting R to the database with dplyr and performing the same operations can have speedup of 20-30x, as seen below. Both these packages use special version of tibble structures, compatible with `dplyr` functions.

When working with a remote database, our first instinct is to import all the data, and perform operations in memory. This can be done efficiently for small databases, but does not scale well when the data are too big.



A better approach involves using the database optimized SQL engine to perform queries, delegating as much of the data exploration as possible, with the aim of bringing in R the better dataset for the analysis. This is anyway ineffective, because we should switch from R to SQL many times.

An even better way can be achieved with `dplyr`. Since `dplyr` verbs can be used to



automatically generate SQL queries, we can take advantage of the grammar to access the database, clean and refine data, using the SQL engine to do heavy data transformations and exploration, while returning a clean R tibble, without writing a single SQL query: `dplyr`, with an added database driver (like ODBC), will take charge of writing the queries. Useful packages are `odbc, dbplyr and DBI.`

When working with databases, dplyr tries to be as lazy as possible:

- It never pulls data into R unless you explicitly ask for it.
- It delays doing any work until the last possible moment: it collects together everything you want to do and then sends it to the database in one step. Only calling `collect()` will effectively perform the full query, otherwise just few lines are collected.

Combining datasets: joins

Sometimes our data may reside in more than one table. It's possible that, as in a relational database, our data is split in more tables, to avoid repetitions, and only indexes or pointers to many values are included in a table.
When joining tables, R will look for common names, and create a new table that includes all the data. Nonetheless, it's possible that the tables do not contain the same keys. In these cases, the user must specify the table priority, and how to join tables. These choice is implemented when choosing over different join methods, whose arguments are two dataframes/tibbles.

`left_join()`

Priority: first argument. R will output all rows from first argument, adding columns from second argument.

`right_join()`

Priority: second argument. R will output all rows from second argument, adding columns from the first one.

`inner_join()`

Priority: intersection. R will output rows if present in both dataframes.

`full_join()`

Priority: union. R will output any row that is present in at least one of the two dataframes.

`semi_join()`

Filtering join. It will return all rows from first argument where there are matching values in the second one, but keeping just columns from the first one.

A semi join differs from an inner join because an inner join will return one row of the first argument for each matching row of the second, where a semi join will never duplicate rows of the first.

`anti_join()`

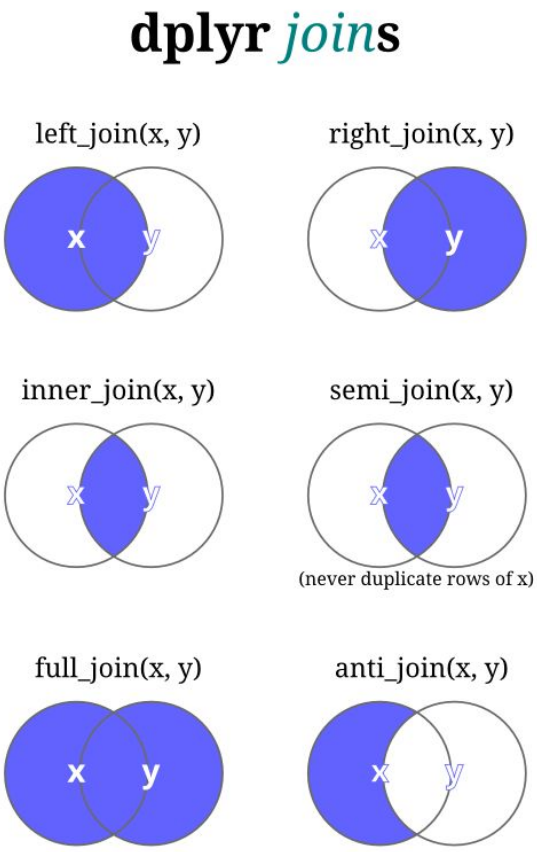
Another filtering join. It's basically the complement of a semi join, returning all rows from the first argument where there are not matching values in the second one, but keeping just columns from the first.

overridden functions

Importing `dplyr` overrides many operations on sets: the following functions are affected.

- `intersect(x, y, ...)`
- `union(x, y, ...)`
- `union_all(x, y, ...)`
- `setdiff(x, y, ...)`
- `setequal(x, y, ...)`

The main effect is to make them more efficient on data frames and tibbles. The default methods call the base version.



Local speed up using `dplyr`: a comparative benchmark

A clean, neat syntax is not enough to justify the heavy diffusion of this package. Being able to vectorize functions brings a massive speedup in operations. A comparative study between base R and different libraries showed the magnitude of this speedup, showing that, despite the diffusion of Python, `dplyr` in R still remains among the most effective tools to handle millions-row datasets[4].

	base R	dplyr
filter	2	1

[4] For more informations about the python-R war, check https://dataconomy.com/2015/05/r-vs-python-the-data-science-wars/

sort	30-60	20-30
new column	1	1
aggregation	8-100	4-30
join	>100	4-15

From Szilard Pafka (Uni. of California), relative execution times of simple functions on large number of rows (10^8-10^{10}) performed with basic R functions and `dplyr` vectorized functions. Data, details and code from https://github.com/szilard/benchm-dplyr-dt

Appendix: dplyr tricks: a (small) toolbox of useful code snippets

you can use dplyr inside your own functions

You can use dplyr functions inside custom functions:

```
yourFun <- function (data, some_string){
    data %>%
    select_(lazyeval::interp(~contains(some_string)))-> data
    return(data)
}
```

selecting the same rows multiple times

Use !! to quickly select a pre-defined list of variables.

```
cols <- c("col1", "col2", "col3)

mydataframe %>%
    select(!!cols)
```

reordering column with everything()

everyting() will consider "everything else".

```
mydataframe %>%
    select( thirdVariable, everything() )
```

rename all variables at once

Apply a rename function to all variable names.

```
library(stringr)
myDataFrame %>%
    rename_all(tolower) %>%    # to lower case
    rename_all(~str_replace_all(., "\\", "_"))
```

cleaning all observations at once

Apply value cleaning functions to all observations in some columns:

```
library(stringr)
myDataFrame %>%
    select(columnA, columnB) %>%
    mutate_all(tolower) %>%
    mutate_all(~str_replace_all(., " ", "_"))
```

finding top 5 values in your dataset

Use `top_n()` to find the n rows who are top ranked for a variable.

```
mtcars %>%
    top_n(5, hp)
```

creating new discrete variables

`case_when()` can be used in place of complex `for` cycles to create new categorical discrete variables.

```
people %>%
    select(name, sex, age, height, hair_color) %>%
    mutate(group = case_when(
          age < 25 & sex == "F" &
                    hair_color == "blonde" ~ "young blond girl",
          sex == "F" & age > 90 ~ "granma",
          age < 25 & sex == "F" ~ "young non-blond girl",
          TRUE ~ "other human being"))
```

working on data by row instead of column

By default, operations like `summarise` and `mutate` work by column. One can force them to work by row by piping `rowwise()`

```
iris %>%
    select(contains("Length")) %>%
    rowwise() %>%
    mutate(Avg.Length = mean(c(Petal.Length, Sepal.Length)))
```