The `tidyverse` collection of packages

Alberto Garfagnini

Università di Padova

R lecture 6

`tidyverse`

- it's an opinionated collection of R packages designed for data science.
- all packages share an underlying design philosophy, grammar, and data structures.
- Web Site: https://www.tidyverse.org/

R packages for data science

The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Install the complete tidyverse with:

```
install.packages("tidyverse")
```

tidyverse packages

ggplot2	gplot2 is a system for declaratively creating graphics, based on The Grammar of Graphics
dplyr	it provides a grammar of data manipulation, providing a consistent set of verbs that solve the most common data manipulation challenges
tidyr	it provides a set of functions that help you get to tidy data. Tidy data is data with a consistent form: in brief, every variable goes in a column, and every column is a variable
readr	it provides a fast and friendly way to read rectangular data (csv, tsv, and fwf)
purrr	it enhances R's functional programming (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors
tibble	a modern re-imagining of the data frame
stringr	it provides a cohesive set of functions designed to make working with strings
forcats	it provides a suite of useful tools that solve common problems with factors

readr : https://readr.tidyverse.org/

- it provides a fast and friendly way to read rectangular data: csv, tsv, and fwf

- `read_csv()` : read and import comma separated (CSV) files
- `read_tsv()` : read and import tab separated (TSV) file
- `read_delim()` : read and import general delimited files
- `read_fsw()` : read and import fixed width files
- `read_log()` : read and import web log files

Alternatives

- in `baseR` : the `read.table()` function
- in `data.table` : the function `fread()` is similar to `read_csv()`

- dplyr is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges:

function	description	SQL equivalent
`select()`	select on columns (i.e. variables)	SELECT
`filter()`	filter a subset of rows	WHERE
`group_by()`	group the data	GROUP BY
`summarise()`	reduces multiple values down to a single summary	-
`arrange()`	changes the ordering of the rows	ORDER BY
`join()`		JOIN
`mutate()`	adds new variables that are functions of existing variables	COLUMN ALIAS

- all function operate on a `data frame` and the result is a new `data frame`
  - dplyr functions never modify their input

dplyr : filter()

- it allows to subset observations based on their values
- the first argument is the name of the data frame
- the second and subsequent arguments are the expressions that filter the data frame

```
filter(flights, month == 1, day == 1)
#> # A tibble: 842 x 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time
                                                    sched_arr_time
#>   <int> <int> <int>   <int>          <int>     <dbl>    <int>
#> 1  2013     1     1     517            515         2      830
#> 2  2013     1     1     533            529         4      850
#> 3  2013     1     1     542            540         2      923
#> 4  2013     1     1     544            545        -1     1004
#> 5  2013     1     1     554            600        -6      812
#> 6  2013     1     1     554            558        -4      740
#> # with 836 more rows, and 11 more variables: ...
```

- `nycflights13::flights` is a data frame that contains all 336,776 flights that departed from New York City in 2013
- it's available in the `library(nycflights13)`

dplyr : arrange()

- it works similarly to `filter()` except that instead of selecting rows, it changes their order
- 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:

```
> arrange(flights, month,day, sched_dep_time)
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time
   <int> <int> <int>    <int>          <int>
 1  2013     1     1      517            515
 2  2013     1     1      533            529
 3  2013     1     1      542            540
 4  2013     1     1      544            545
```

- to rearrange a column in descending row, use `desc()`

```
> arrange(flights, month,day, desc(sched_dep_time))
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time
   <int> <int> <int>    <int>          <int>
 1  2013     1     1     2353           2359
 2  2013     1     1     2353           2359
 3  2013     1     1     2356           2359
 4  2013     1     1     2250           2255
```


dplyr : select()

- it allows to select a subrange of columns in the data frame

```
> select(flights, year, month, day, dep_time)
# A tibble: 336,776 x 4
    year month   day dep_time
   <int> <int> <int>    <int>
 1  2013     1     1      517
 2  2013     1     1      533
 3  2013     1     1      542
```

- usual selection rules apply: we can select a range of columns

```
> select(flights, dep_time:dep_delay)
# A tibble: 336,776 x 3
   dep_time sched_dep_time dep_delay
      <int>          <int>     <dbl>
 1      517            515         2
 2      533            529         4
 3      542            540         2
```

- we can remove columns with the - (minus) sign

```
> select(flights, -(year:day))
# A tibble: 336,776 x 16
   dep_time sched_dep_time dep_delay arr_time
      <int>          <int>     <dbl>    <int>
 1      517            515         2      830
 2      533            529         4      850
 3      542            540         2      923
```

dplyr : mutate()

- besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns

```
flights_sml <- select(flights, year:day, ends_with("delay"),
                      distance, air_time)

> mutate(flights_sml, gain = dep_delay - arr_delay,
         speed = distance / air_time * 60)
# A tibble: 336,776 x 9
    year month  day dep_delay arr_delay distance air_time  gain speed
   <int> <int> <int>   <dbl>   <dbl>   <dbl>   <dbl> <dbl> <dbl>
 1  2013     1     1       2      11    1400     227    -9  370.
 2  2013     1     1       4      20    1416     227   -16  374.
 3  2013     1     1       2      33    1089     160   -31  408.
 4  2013     1     1      -1     -18    1576     183    17  517.
```

- if we want to keep only the new variables, we use `transmute()`:

```
transmute(flights, gain = dep_delay - arr_delay,
          hours = air_time / 60, gain_per_hour = gain / hours)
# A tibble: 336,776 x 3
    gain hours gain_per_hour
   <dbl> <dbl>         <dbl>
 1    -9 3.78          -2.38
 2   -16 3.78          -4.23
 3   -31 2.67         -11.6
 4    17 3.05           5.57
```


dplyr : summarise() and group_by()

- summarise() collapses a data frame to a single row
- it is very useful ewhen combined with group_by()
- group_by() takes an existing data frame and converts it into a grouped data frame where operations are performed by group

```
not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))

not_cancelled %>% group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))
 A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day  mean
   <int> <int> <int> <dbl>
 1  2013     1     1 11.4
 2  2013     1     2 13.7
 3  2013     1     3 10.9
 4  2013     1     4  8.97
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.56
 9  2013     1     9  2.30
10  2013     1    10  2.84
```

The PIPE operator %>%

- it process a data-object with a sequence of operations by passing the result of one step as input for the next step using infix-operators rather than the more typical R method of nested function calls
- it is defined in the `magrittr` package, but it gained huge visibility and popularity with the `dplyr` package

Syntax

```
lhs %>% rhs # pipe syntax for rhs(lhs)

lhs %>% rhs(a = 1) # pipe syntax for rhs(lhs, a = 1)

lhs %>% rhs(a = 1, b = .) # pipe syntax for rhs(a = 1, b = lhs)

lhs %<>% rhs # pipe syntax for lhs <- rhs(lhs)

lhs %$% rhs(a) # pipe syntax for with(lhs, rhs(lhs$a))

lhs %T>% rhs # pipe syntax for { rhs(lhs); lhs }
```

- `lhs` = a value or the magrittr placeholder
- `hhs` = a function call using the magrittr semantics

The PIPE operator %>% - examples

Basic use

```
library(magrittr)

1:10 %>% mean
# [1] 5.5

# is equivalent to
mean(1:10)
# [1] 5.5

years <- factor(2008:2012)
as.numeric(as.character(years))

# piping equivalent
years %>% as.character %>% as.numeric

grepl("Wo", substring("Hello World", 7, 11))
#> [1] TRUE

"Hello World" %>% substring(7, 11) %>% grepl(pattern = "Wo")
#> [1] TRUE
"Hello World" %>% substring(7, 11) %>% grepl("Wo", .)
#> [1] TRUE
"Hello World" %>% substring(7, 11) %>% { c(paste(. ,'Hi', .)) }
#> [1] "World Hi World"
```

Combining multiple operation with the pipe

- the pipe, %>%, can be used to rewrite multiple operations in a compact way; it can be read left-to-right, top-to-bottom
- piping improves code readability

```
select(flights, year:day, ends_with("delay"),
       distance , air_time) %>%
       transmute(gain = dep_delay - arr_delay,
       speed = distance / air_time * 60)
# A tibble: 336,776 x 2
    gain speed
   <dbl> <dbl>
 1    -9  370.
 2   -16  374.
 3   -31  408.
 4    17  517.
 5    19  394.
 6   -16  288.
 7   -24  404.
 8    11  259.
 9     5  405.
10   -10  319.
# ... with 336,766 more rows
```

- behind the scenes, `x %>% f(y)` turns into `f(x, y)`, and `x %>% f(y) %>% g(z)` turns into `g(f(x, y), z)` and so on

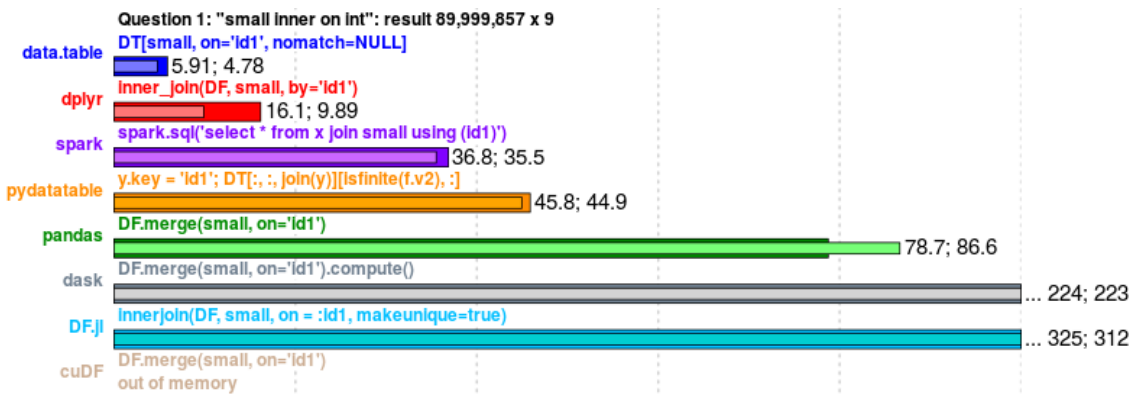
data.table

https://github.com/Rdatatable/data.table/wiki

- it provides a high-performance version of base R's `data.frame`
- `data.table` is created using the `fread()` function for reading data on disk, or provided on the fly with the `data.table()` function

```
DT = data.table(
  id = c("b","a","a","c","c","b"),
  val = c(4,2,3,1,5,6)
)
```

- existing objects can be converted to `data.table` using the `setDT()` and the `as.data.table()` functions
- it is a optimized and runs faster for large data sets (example plot: 10^8 rows with 7 columns ➜5 GB data) https://h2oai.github.io/db-benchmark/



data.frame - 1

- have a 2D matrix like structure: rows and columns. We can:
- subset rows

```
X[X$id != "a"]
```

- select columns

```
X[, "val"]
```

- and do it at the same time:

```
X[X$id != "a", "val"]
```

X

	id	val
1	b	4
2	a	2
3	a	3
4	c	1
5	c	5
6	b	6

data.frame - 2

- we can compute on columns:
- sum column `valA` only for the rows where `code != "abd"`

```
sum(DF[DF$code != "abd", "valA"])
1.9
```

- we can perform operations on aggregated groups
- sum `valA` and `valB` columns for `code != "abd"` and group by `id`

```
aggregate(cbind(valA, valB) ~ id,
          DF[DF$code != "abd", ], sum)
```

- we can update values

```
DF[DF$code == "abd", "valA"] <- NA
```

DF

	id	code	valA	valB
1	1	abc	0.1	11
2	1	abc	0.6	7
3	1	abd	NA	5
4	2	apq	0.9	10
5	2	apq	0.3	13

	id	valA	valB
1	1	0.7	18
2	2	1.2	23

data.table

- they allow column names to be seen as variables within the [...]
- and computations can be done with them directly
- an additional argument, `by` is introduced
- a `data.table` has a row/column data structure, as `data.frames`

- subset rows

```
X[id != "a", ]
```

- select columns

```
X[, val]
```

- and compute on columns

```
X[, mean(val)]
```

- subset rows and select/compute on columns

```
X[X$id != "a", mean(val)]
```

- and with a 'virtual 3rd dimension, `group by`

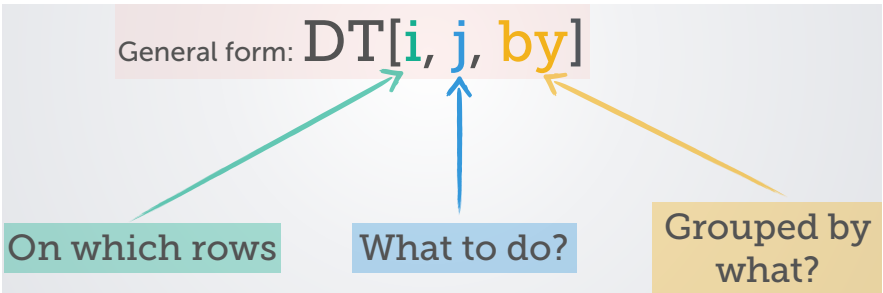
```
X[X$id != "a", .sum(valA), sum(valB), by=id]
```

X

	id	val
1:	b	4
2:	a	2
3:	a	3
4:	c	1
5:	c	5
6:	b	6

equivalence data.frame vs data.table

- think in terms of basic units: rows, columns and groups




```
sum(DF[DF$code != "abd", "valA"])
```

```
DT[code != "abd", sum(valA)]
```

```
aggregate(cbind(valA, valB) ~ id,
          DF[DF$code != "abd", ],
          sum)
```

```
DT[code != "abd",
   .(sum(valA), sum(valB)),
   by = id]
```

```
DF[DF$code == "abd", "valA"] <- NA
```

```
DT[code == "abd", valA := NA]
```