SciViews :: CHEAT SHEET

SciViews::R

SciViews::R offers additional functions on top of base R and tidyverse. To use them just type:

```
SciViews::R

# For a better help, replace ?topic by
.?topic or about("topic")
```

Read datasets

read() unifies the data importation methods and also loads datasets from R packages.

```
ub <- read("urchin_bio", package = "data.io",
lang = "fr") – Load data from a package

ub1 <- read("file.csv") – Import local data

ub1 <- read$csv2("file.csv")  or
ub1 <- read("file.csv", type = "csv2") – Import
local data with explicit format specification
```

write() for data exportation (always explicit).

```
write$csv(x, file = "filename.csv")
```

read() and **write()** support many formats : .txt, .rds, .xls(x), .sas, …

```
data_types() – List supported data formats
```

Workflow

Functions are building blocks. They can be *nested*, *piped* (**%>.%** operator), or used in *successive statements*. A pipeline is usually more readable.

```
ub <- read("urchin_bio", package = "data.io")
```

- Successive statements: select then filter data

```
ub1 <- select_(ub, 1:5)
ub2 <- filter_(ub1, ~origin == "Farm")
```

- Nesting functions

```
ub2 <- filter_(select_(ub, 1:5),
   ~origin == "Farm")
```

- Pipeline with {svFlow}

```
ub %>.%
  select_(., 1:5) %>.%
  filter_(., ~origin == "Farm") ->
  ub2
```

%>.% is an explicit pipe (dot must be specified). The base R pipe **|>** can also be used.

Data visualisation

chart() uses four rules against ggplot()

```
ub <- read("urchin_bio", package = "data.io")
ggplot(data = ub, mapping = aes(x = weight, y = height,
   colour = origin) +
   geom_point()
```

1. Replace **ggplot()** by **chart()**

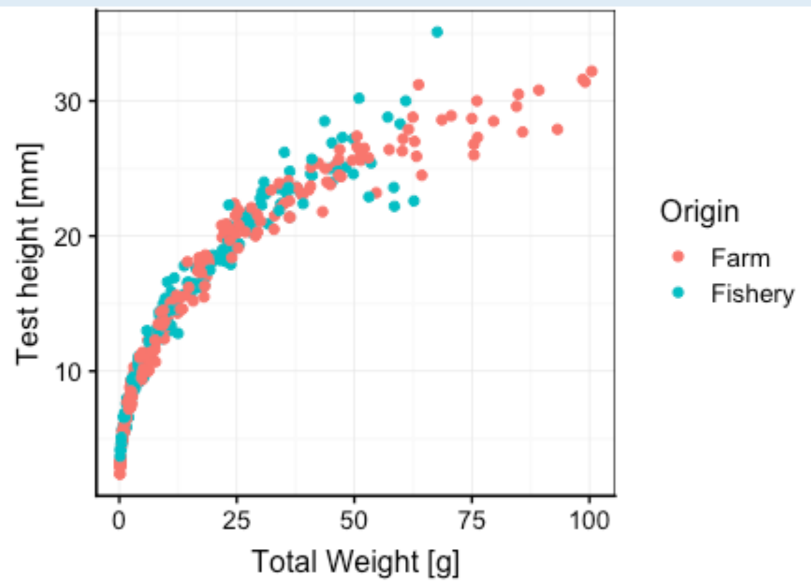
```
chart(data = ub, aes(x = weight, y = height, colour = origin)) +
  geom_point()
```

2. Replace argument **aes()** by **f_aes()** to use a formula instead

```
ggplot(data = ub, f_aes(height ~ weight %col=% origin)) +
  geom_point()
```

3. Even better: use **chart()** with formula syntax directly

```
chart(data = ub, height ~ weight %col=% origin) +
  geom_point()
```



In addition **chart()** uses associated metadata (labels and units) to provide a plot close to publication ready.

```
ggplot(data = ub, mapping = aes(x = weight, y = height) +
   geom_point() +
   facet_grid(~ origin)
```

4. Use facets (https://ggplot2-book.org/facet.html) in the formula

```
chart(data = ub, height ~ weight | origin) +
   geom_point()
```

chart() provides a unified interface for base plots, lattice and ggplot2 with argument **type =** or with **$**

```
chart(data = ub, height ~ weight | origin, type =
"geom_point") – type = e.g. "xyplot" for lattice plot

chart$xyplot(data = ub, height ~ weight | origin)
```

Reproductible research

Respect the three rules below for reproductible works:

1. Organise your analyses in **(RStudio) Project** with a README(.md) file and all other files in dedicated directories, e.g.:
- **data** sub-dir: all datasets (also **data-raw)**
- **R** sub-dir: all R scripts
- Main directory: all notebooks, reports, presentations, …

2. Make your project **portable**:
- Use only *relative* paths, or use **here::here()**

3. Use a **version control** system like git (and GitHub, Gitlab, …)

Data manipulation

The **svTidy** package provides an alternative to Tidyverse's dplyr and tidyr. The syntax is slightly different for good reasons (more explicit use of non-standard evaluation through formula) and these functions are also faster. Rules to convert from Tidyverse to svTidy:

1. Use svTidy's functions ending with '_', e.g., `select_()` instead of `select()`, `filter_()` instead of `filter()`… With these functions, `data = .` or `.data = .` is facultative in *all* contexts (not only with pipe operators).

2. Use standard evaluation (specify `df$var` for variable `var` in the data frame `df`), or place a non-standard evaluation in a formula by prepending it by ~

3. Use "fast" stat functions in your calculations (especially if you perform calculations over groups). For instance, replace `mean()` by `fmean()`. Use `list_fstat_functions()` to get a list of all existing fast stat functions

4. Use a two-sided formula instead of `varname := value`

5. Use the "bullet-point" `.=` inside brackets `{…}` for a group of successive instructions instead of long pipelines `|>`, `%>%` or `%>.%` : they are easier to debug

All five rules are applied in the following example:

Tidyverse

```
varname <- 'hp'
summ <-
  mtcars %>%
    group_by(cal, vs) %>%
    summarise(
      disp = mean(disp),
      varname := mean(hp)
    )
```

svTidy

```
varname <- 'hp'
summ <- {
  .= mtcars
  .= group_by_('cal', 'vs')
  .= summarise_(
    disp = ~fmean(disp),
    varname ~ fmean(hp))
}
```

6. **Bonus:** no need to embrace variables to pass their values within functions

```
varsum <- function(data, var)
  summarise(data,
    min = min({{ var }}),
     max = max({{ var }}))
```

```
varsum <- function(data, var)
  summarise_(data,
    min = ~fmin(var),
     max = ~fmax(var))
```

Updated: 2025-09