DSCI 6607 – Programmatic Data Analysis Using Python and R

Module 4: Data frame with Tibble

Dr. Armin Hatefi*

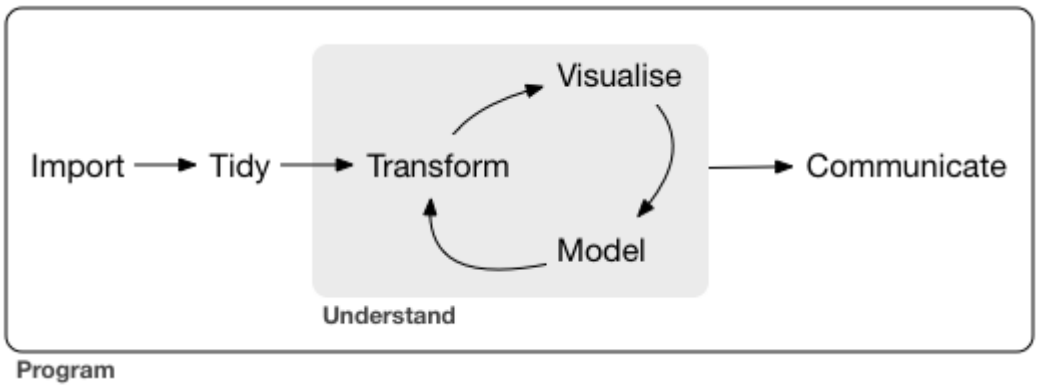
September 30, 2024

Contents

Data Science Workflow

Data science is an exciting discipline that allows you to turn raw data into understanding, insight, and knowledge.

1. Import
2. Wrangle (tidy & transform)
3. Visualize
4. Model
5. Communicate



An Introduction to `tidyverse`

The `tidyverse` is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures. [1]

`tidyverse` includes packages for **importing, wrangling, exploring and modeling data**.

The system is intended to make data scientists more productive. To use `tidyverse` do the following:

```
# Install the package
install.packages("tidyverse")
# Load it into memory
library("tidyverse")
```

[1] Tidyverse website

The `tibble` Package

The `tibble` package is part of the core `tidyverse`.

Tibbles are a modern take on data frames. They keep the features that have stood the test of time, and drop the features that used to be convenient but are now frustrating.

`tibbles` are data frames, tweaked to make life a little easier. Unlike regular `data.frames` they:

- never change the type of the inputs (e.g. do not convert strings to factors!)
- never changes the names of variables

- never creates row.names()
- only recycles inputs of length 1

Using `tibbles`

To use functions from `tibble` and other `tidyverse` packages:

```
# load it into memory
library(tidyverse)
```

Printing `tibble` is much nicer, and always fits into your window:

```
# e.g. a built-in dataset 'diamonds' is a tibble:
class(diamonds)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
head(diamonds)
```

```
## # A tibble: 6 x 10
##   carat cut       color clarity depth table price     x     y     z
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
## 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
## 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
## 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
## 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
## 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
```

```
diamonds
```

```
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
## # i 53,930 more rows
```

Creating `tibbles` is similar to `data.frames`, but no strict rules on column names:

```
tb <- tibble(x = 1:5, y = 1,z = x ^ 2 + y, `:)` =  "smile")
tb
```

```
## # A tibble: 5 x 4
##       x     y     z `:)`
##   <int> <dbl> <dbl> <chr>
## 1     1     1     2 smile
## 2     2     1     5 smile
## 3     3     1    10 smile
## 4     4     1    17 smile
## 5     5     1    26 smile
```

Subsetting `tibbles` is stricter than subsetting `data.frames`, and ALWAYS returns objects with expected class: a single `[` returns a `tibble`, a double`[[` returns a vector.

```
class(diamonds$carat)
```

```
## [1] "numeric"
```

```
class(diamonds[["carat"]])
```

```
## [1] "numeric"
```

```
class(diamonds[, "carat"])
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

More on `tibbles`

Convert a regular data frame to tibble:

```
# a regular data frame
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
 as_tibble(iris)
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # i 140 more rows
```

Convert a tibble to data frame:

```
tb
```

```
## # A tibble: 5 x 4
##       x     y     z `:)`
##   <int> <dbl> <dbl> <chr>
## 1     1     1     2 smile
## 2     2     1     5 smile
## 3     3     1    10 smile
## 4     4     1    17 smile
## 5     5     1    26 smile
```

```
as.data.frame(tb)
```

```
##   x y  z    :)
## 1 1 1  2 smile
## 2 2 1  5 smile
## 3 3 1 10 smile
## 4 4 1 17 smile
## 5 5 1 26 smile
```

Transposed tibbles:

```
tribble(
~x, ~y, ~z,
#--/--/----
"a", 2, 3.6,
"b", 1, 8.5
)
```

```
## # A tibble: 2 x 3
##   x         y     z
##   <chr> <dbl> <dbl>
## 1 a         2   3.6
## 2 b         1   8.5
```

By default, tibble prints the first 10 rows and all columns that fit on screen.

```
my_df <- tibble(
  a = lubridate::now() + runif(1e3) * 86400,
  b = lubridate::today() + runif(1e3) * 30,
  c = 1:1e3,
  d = runif(1e3),
  e = sample(letters, 1e3, replace = TRUE)
)
print(my_df,n=12)
```

```
## # A tibble: 1,000 x 5
##    a                   b              c      d e
##    <dttm>              <date>     <int>  <dbl> <chr>
##  1 2024-10-01 19:20:50 2024-10-09     1 0.550  t
##  2 2024-10-01 22:45:15 2024-10-15     2 0.275  t
##  3 2024-10-02 04:28:06 2024-10-11     3 0.0542 z
##  4 2024-10-02 01:00:40 2024-10-16     4 0.519  u
##  5 2024-10-02 15:58:42 2024-10-18     5 0.467  f
##  6 2024-10-02 03:17:10 2024-10-17     6 0.368  r
##  7 2024-10-01 22:05:37 2024-10-26     7 0.0275 u
##  8 2024-10-02 12:29:10 2024-10-21     8 0.367  b
##  9 2024-10-02 00:47:05 2024-10-01     9 0.376  x
## 10 2024-10-02 09:34:25 2024-10-11    10 0.612  z
## 11 2024-10-02 18:06:36 2024-10-10    11 0.794  d
## 12 2024-10-02 11:40:58 2024-10-04    12 0.569  u
## # i 988 more rows
```

To change number of rows and columns to display:

```
# install.packages(nycflights13)
library(nycflights13)
  print(flights,n = 10, width = Inf)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
##    arr_delay carrier flight tailnum origin dest  air_time distance  hour minute
##        <dbl> <chr>    <int> <chr>   <chr>  <chr>    <dbl>    <dbl> <dbl>  <dbl>
##  1        11 UA        1545 N14228  EWR    IAH        227     1400     5     15
##  2        20 UA        1714 N24211  LGA    IAH        227     1416     5     29
##  3        33 AA        1141 N619AA  JFK    MIA        160     1089     5     40
##  4       -18 B6         725 N804JB  JFK    BQN        183     1576     5     45
##  5       -25 DL         461 N668DN  LGA    ATL        116      762     6      0
##  6        12 UA        1696 N39463  EWR    ORD        150      719     5     58
##  7        19 B6         507 N516JB  EWR    FLL        158     1065     6      0
##  8       -14 EV        5708 N829AS  LGA    IAD         53      229     6      0
##  9        -8 B6          79 N593JB  JFK    MCO        140      944     6      0
## 10         8 AA         301 N3ALAA  LGA    ORD        138      733     6      0
##    time_hour
##    <dttm>
##  1 2013-01-01 05:00:00
##  2 2013-01-01 05:00:00
##  3 2013-01-01 05:00:00
##  4 2013-01-01 05:00:00
##  5 2013-01-01 06:00:00
##  6 2013-01-01 05:00:00
##  7 2013-01-01 06:00:00
##  8 2013-01-01 06:00:00
##  9 2013-01-01 06:00:00
## 10 2013-01-01 06:00:00
## # i 336,766 more rows
```

To change the default print behavior:

Subsetting

```
df <- tibble(
  x = runif(5),
  y = rnorm(5)
)
```

```
# Extract by name
df$x
```

```
## [1] 0.3590573 0.8052988 0.9279510 0.7939550 0.5708227
```

```
df[["x"]]
```

```
## [1] 0.3590573 0.8052988 0.9279510 0.7939550 0.5708227
```

```
# Extract by position
df[[1]]
```

```
## [1] 0.3590573 0.8052988 0.9279510 0.7939550 0.5708227
```

An Introduction to Pipe

- Tidyverse functions are at their best when composed together using the pipe operator
- It looks like this: `%>%`. **Shortcut**: use `ctrl + shift + m` in RStudio
- This operator actually comes from the `magrittr` package (automatically included in `dplyr`)
- **Piping** at its most basic level:
  - Take one return value and automatically feed it in as an input to another function, to form a flow of results

How to read pipes: single arguments

Passing a single argument through pipes, we interpret something like:

```
x %>% f %>% g %>% h
```

as `h(g(f(x)))`

Key takeaway: in your mind, when you see `%>%`, read this as **"and then"**

Simple example

We can write `exp(1)` with pipes as `1 %>% exp`, and `log(exp(1))` as `1 %>% exp %>% log`

```
exp(1)
```

```
## [1] 2.718282
```

```
1 %>% exp
```

```
## [1] 2.718282
```

```
1 %>% exp %>% log
```

```
## [1] 1
```

How to read pipes: multiple arguments

Now for multi-arguments functions, we interpret something like:

```
x %>% f(y)
```

as `f(x,y)`

Simple example

```
mtcars %>% head(4)
```

And what's the "old school" (base R) way?

```
head(mtcars, 4)
```

Notice that, with pipes:

- Your code is more readable (arguably)
- You can run partial commands more easily

The dot

The command `x %>% f(y)` can be equivalently written in **dot notation** as:

```
x %>% f(., y)
```

What's the advantage of using dots? Sometimes you want to pass in a variable as the *second* or *third* (say, not first) argument to a function, with a pipe. As in:

```
x %>% f(y, .)
```

which is equivalent to `f(y,x)`

More details about `tibble`?

You can read more about other `tibble` features by calling on your R console:

```
vignette("tibble")
```