Module 2.2

The Tidyverse and Data Wrangling in R

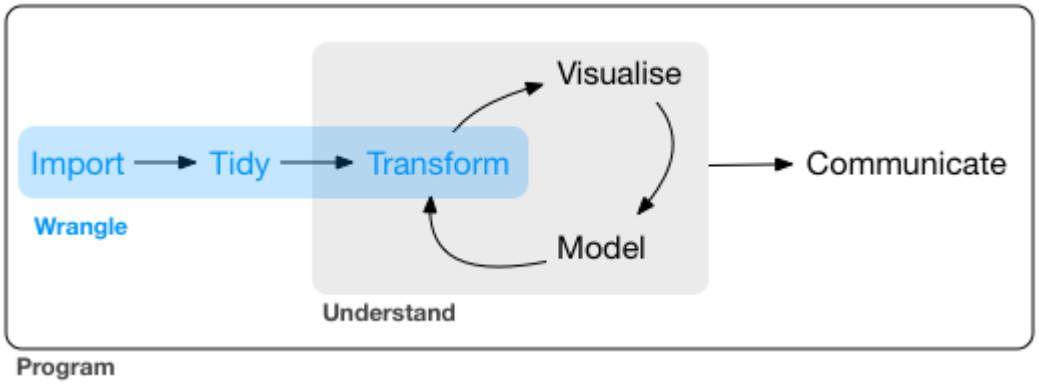
Fall 2020

Thanks to Christine Albano, who pulled this module together!

Load script for module #2.2

1. Click here to download the script! Save the script to a convenient folder on your laptop.
2. Load your script in RStudio.

In this tutorial we introduce the 'Tidyverse' set of packages, which were designed to facilitate everyday data management and visualization tasks in R using a consistent data structure and syntax. Much of the content in this tutorial is based on the online resource R for Data Science by Garrett Grolemund and Hadley Wickham.



The core tidyverse set of packages includes all of the below (plus a handfull of others). For now, just focused on the first four for data wrangling:

- dplyr, for data manipulation
- tidyr, for data tidying
- readr, for data import
- tibble, for tibbles, a modern re-imagining of data frames
- ggplot2, for data visualisation
- purrr, for functional programming
- stringr, for strings
- forcats, for factors

Learning goals

- Use *Tidyverse* grammar (piping) and data structures to perform basic data manipulation tasks
- Use *tidyr* and *dplyr* data transformation verbs to wrangle data
- Work with and parse dates using *lubridate*

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

Using the pipe operator %>% in R

The 'tidyverse' set of packages takes advantage of the pipe operator **%>%**, which provides a clean and intuitive way to structure code and perform sequential operations in R.

Key advantages include:

- code is more readable and intuitive – reads left to right, rather than inside out as is the case for nested function
- perform multiple operations without creating a bunch of intermediate (temporary) datasets

This operator originally comes from the *magrittr* package, which is included in the installation of all of the tidyverse packages. The shortcut for the pipe operator is ctrl-shift-m. When reading code out loud, use 'then' for the pipe. For example the command here:

```
x %>% log() %>% round(digits=2)
```

can be interpreted as follows:

> Take the object "x", THEN take its natural logarithm, THEN round that value to 2 significant figures

The structure is simple. Start with the object you want to manipulate, and apply actions (e.g., functions) to that object in the order in which you want to apply them.

Here is a quick example.

```
####
####  Using the pipe operator %>% (ctrl-shift-m)
####

# start with a simple example
x <- 3

# calculate the log of x
log(x) # form f(x) is equivalent to
```

```
## [1] 1.098612
```

```
x %>% log() # form x %>% f
```

```
## [1] 1.098612
```

```
# example of multiple steps in pipe
round(log(x), digits=2) # form g(f(x)) is equivalent to
```

```
## [1] 1.1
```

```
x %>% log() %>% round(digits=2) # form x %>% f %>%  g
```

```
## [1] 1.1
```

Importing data into *R* using *readr*

Here, we import meteorological data from Hungry Horse and Polson Kerr Dams in Montana as a *tibble* dataframe. The *readr* package imports data as a special type of dataframe called a *tibble*. The *tibble* dataframe has several advantages over the 'regular' dataframe, including nicely formatted printing, more useful defaults, faster imports, and it also flags potential issues in your dataset early on – which can save you a lot of time later! All this said, tibble and regular dataframes are readily converted to one type vs. the other if the need arises.

Download an example dataset

The next few examples use meteorological data from Hungry Horse (HH) and Polson Kerr (PK) dams. You can download this data set by clicking here.

Import as a *tibble*

```
####
####  Import data as a Tibble dataframe and take a quick glance
####

# import meteorological data from Hungry Horse (HH) and Polson Kerr (PK) dams as tibble dataframe using
clim_data <- read_csv("MTMetStations.csv")

# display tibble - note nice formatting and variable info, entire dataset is not displayed as is case i
clim_data

 # display the last few lines of the data frame
tail(clim_data)
```

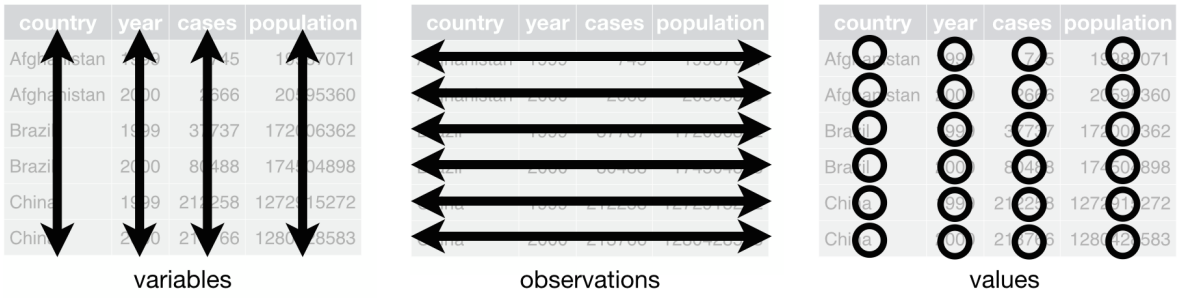
Making data tidy using *tidyr*

All of the tidyverse set of packages are designed to work with *Tidy* formatted data.

This means:

- Each variable must have its own column.
- Each observation must have its own row.
- Each value must have its own cell.

This is what it looks like:



The *tidyr* package has several functions to help you get your data into this format.

Use "gather()" to get all of the values and variables into a single column.

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

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

table4

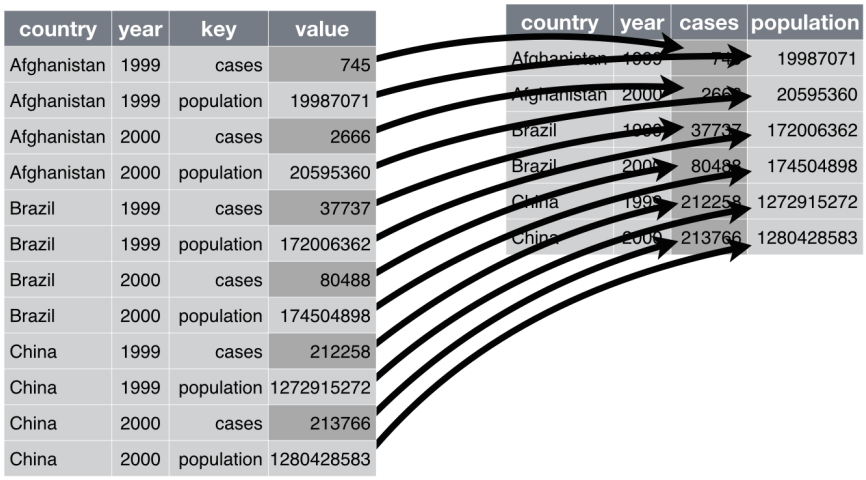
Use "separate()" to separate a single column into two separate columns. "unite()" does the opposite

country	year	rate
Afghanistan	1999	**745** / 19987071
Afghanistan	2000	**2666** / 20595360
Brazil	1999	**37737** / 172006362
Brazil	2000	**80488** / 174504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

table3

country	year	cases	population
Afghanistan	1999	**745**	19987071
Afghanistan	2000	**2666**	20595360
Brazil	1999	**37737**	172006362
Brazil	2000	**80488**	174504898
China	1999	**212258**	1272915272
China	2000	**213766**	1280428583

Use “spread()” to distribute two variables in a single (‘key’) column into separate columns, with their data values(‘value’)



country	year	key	value
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

table2

```
####
####  Use Tidyr verbs to make data 'tidy'
####

# look at clim_data -- is it in tidy format? What do we need to do to get it there?
head(clim_data)

# gather column names into a new column called 'climvar_station', and all of the numeric precip and tem
gather_clim_vars <- gather(clim_data,
                           key = climvar_station,
                           value = value,
                           -Date)

gather_clim_vars

# separate the climvar_station column into two separate columns that identify the climate variable and
separate_clim_vars <- gather_clim_vars %>%
  separate(climvar_station,
           into = c("Station","climvar"))

separate_clim_vars

# spread distributes the clim_var column into separate columns, with the data values from the 'value' c
tidy_clim_data <- spread(separate_clim_vars,
                         key = climvar,
                         value = value)

tidy_clim_data


# repeat above as single pipe series without creation of intermediate datasets
```

```
tidy_clim_data <- clim_data %>%
  gather(key = climvar_station,
         value = value,
         -Date) %>%
  separate(climvar_station,
           into = c("Station","climvar")) %>%
  spread(key = climvar,
         value = value)

tidy_clim_data
```

Using dplyr – the data wrangling workhorse

dplyr is by far one of the most useful packages in all of the tidyverse as it allows you to quickly and easily summarize and manipulate your data once you have it in tidy form. The basic dplyr verbs include:

```
  * grouping data with *group_by()*

  * filtering rows with *filter()*

  * creating new variables with *mutate()*

  * summarizing with *summarize()*

  * selecting columns with *select()*

  * sorting columns by row with *arrange()*
```

you can add variants _all (affects all variables), _at (affects selected variables), _if (affects variables that meet criteria) to most of these verbs

```
####
####  Use dplyr verbs to wrangle data
####

# example of simple data selection and summary using group_by, summarize, and mutate verbs

# take tidy_clim_data, then
# group data by station, then
# calculate summaries and put in columns with names mean.precip.in, mean.TMax.F, and mean.Tmin.F, then
# transform to metric and put in new columns mean.precip.in, mean.TMax.F, and mean.Tmin.F

station_mean1 <- tidy_clim_data %>%
  group_by(Station) %>%
  summarize(
    mean.precip.in = mean(PrcpIN, na.rm=TRUE),
    mean.TMax.F = mean(TMaxF, na.rm=TRUE),
    mean.TMin.F = mean(TMinF, na.rm=TRUE)) %>%
  mutate(
    mean.precip.mm = mean.precip.in * 25.4,
    mean.TMax.C = (mean.TMax.F - 32) * 5 / 9,
```

```
    mean.TMin.C = (mean.TMin.F - 32) * 5 / 9
  )

station_mean1

# using variants

# take tidy_clim_data, then
# group data by station, then
# calculate summary (mean of all non-NA values) for numeric data only, then
# transform temp data (.) from F to C, then
# transform precip data (.) from in to mm


station_mean2 <- tidy_clim_data %>%
  group_by(Station) %>%
  summarize_if(is.numeric, mean, na.rm=TRUE) %>%
  mutate_at(vars(TMaxF, TMinF), funs(C=(.-32)*5/9)) %>%
  mutate_at(vars(PrcpIN), funs(Prcp.mm=.*25.4))

station_mean2
```

Working with Dates with *lubridate* - Date from character string

Dates can be tricky to work with and we often want to eventually get to the point that we can summarize our data by years, seasons, months, or even day of the year. The *lubridate* package in R (also part of the tidyverse) is very useful for creating and parsing dates for this purpose. Date/time data often comes as strings in a variety of orders/formats. Lubridate is useful because it automatically works out the date format once you specify the order of the components. To to do this, identify the order in which year, month, and day appear in your dates, then arrange "y", "m", and "d" in the same order. That gives you the name of the lubridate function that will parse your date. For example:

```
####
####  Using lubridate to format and create date data types
####

library(lubridate)

date_string <- ("2017-01-31")

# convert date string into date format by identifing the order in which year, month, and day appear in

date_dtformat <- ymd(date_string)

# note the different formats of the date_string and date_dtformat objects in the environment window.

# a variety of other formats/orders can also be accommodated. Note how each of these are reformatted to

mdy("January 31st, 2017")
```

```
## [1] "2017-01-31"
```

```
dmy("31-Jan-2017")
```

```
## [1] "2017-01-31"
```

```
ymd(20170131)
```

```
## [1] "2017-01-31"
```

```
ymd(20170131, tz = "UTC")
```

```
## [1] "2017-01-31 UTC"
```

```
# can also make a date from components. this is useful if you have columns for year, month, day in a da
year<-2017
month<-1
day<-31
make_date(year, month, day)
```

```
## [1] "2017-01-31"
```

```
# times can be included as well. Note that unless otherwise specified, R assumes UTC time

ymd_hms("2017-01-31 20:11:59")
```

```
## [1] "2017-01-31 20:11:59 UTC"
```

```
mdy_hm("01/31/2017 08:01")
```

```
## [1] "2017-01-31 08:01:00 UTC"
```

```
# we can also have R tell us the current time or date

now()
```

```
## [1] "2020-09-03 12:29:04 PDT"
```

```
now(tz = "UTC")
```

```
## [1] "2020-09-03 19:29:04 UTC"
```

```
today()
```

```
## [1] "2020-09-03"
```

Working with Dates with lubridate - Parsing Dates

Once dates are in date format, we can easily pull out their individual components like this:

```
####
####  Parsing dates with lubridate
####

datetime <- ymd_hms("2016-07-08 12:34:56")

# year
year(datetime)
```

```
## [1] 2016
```

```
# month as numeric
month(datetime)
```

```
## [1] 7
```

```
# month as name
month(datetime, label = TRUE)
```

```
## [1] Jul
## Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < Oct < Nov < Dec
```

```
# day of month
mday(datetime)
```

```
## [1] 8
```

```
# day of year (julian day)
yday(datetime)
```

```
## [1] 190
```

```
# day of week
wday(datetime)
```

```
## [1] 6
```

```
wday(datetime, label = TRUE, abbr = FALSE)
```

```
## [1] Friday
## Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < Friday < Saturday
```

Working with Dates - Parsing Dates Example

We can use lubridate and dplyr to make new columns from a date for year, month, day of month, and day of year

first we convert the character string date into date format. Here we are naming the new column "Date", so R will replace character string with date formatted dates

```
####
####  Using lubridate with dataframes and dplyr verbs
####

# going back to our tidy_clim_data dataset we see that the date column is formatted as character, not d
head(tidy_clim_data)

# change format of date column
tidy_clim_data <- tidy_clim_data %>%
  mutate(Date = mdy(Date))
tidy_clim_data
```

now we can use mutate to create individual columns for the date components

```
# parse date into year, month, day, and day of year columns
tidy_clim_data <- tidy_clim_data %>% mutate(
  Year = year(Date),
  Month = month(Date),
  Day = mday(Date),
  Yday = yday(Date))

tidy_clim_data

# calculate total annual precipitation by station and year
annual_sum_precip_by_station <- tidy_clim_data %>%
  group_by(Station, Year) %>%
  summarise(PrecipSum = sum(PrcpIN))

annual_sum_precip_by_station
```

Challenge Exercises

Exercise 1 - make data tidy

- Create a tibble dataframe using the code below and make it tidy. Do you need to spread or gather? What are the variables? what are the observations?

```
pregnancy <- tribble(
    ~pregnant, ~male, ~female,
    "yes",     NA,    10,
    "no",      20,    12
)
```

Exercise 2 - data summary with dplyr

- use the dplyr verbs and the tidy_clim_data dataset you created above to calculate monthly average Tmin and Tmax for each station

Exercise 3 - make and parse a date with lubridate

make a date object out of a character string of your birthday and find the day of year it occurs on.

–go to next module–