Drake – Reproducible Workflows at Scale

Dr Andrew J. Stewart

E: drandrewjstewart@gmail.com
T: @ajstewart_lang
G: ajstewartlang

Introducing Drake

“Data analysis can be slow. A round of scientific computation can take several minutes, hours, or even days to complete. After it finishes, if you update your code or data, your hard-earned results may no longer be valid. How much of that valuable output can you keep, and how much do you need to update? How much runtime must you endure all over again?

For projects in R, the `drake` package can help. It analyzes your workflow, skips steps with up-to-date results, and orchestrates the rest with optional distributed computing. At the end, `drake` provides evidence that your results match the underlying code and data, which increases your ability to trust your research."

https://books.ropensci.org/drake/index.html

Introducing Drake

Too many data science projects follow a Sisyphean loop:

- Launch the code.
- Wait while it runs.
- Discover an issue.
- Restart from scratch.

For projects with long runtimes, people tend to get stuck. But with `drake`, you can automatically:

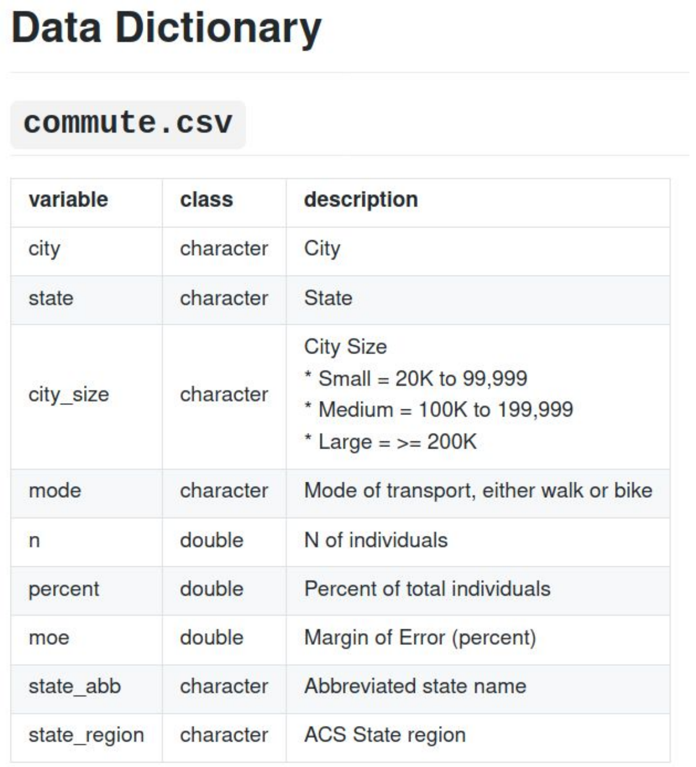
- Launch the parts that changed since last time.
- Skip the rest.

https://books.ropensci.org/drake/index.html

Working with a dataset

We're going to use a TidyTuesday dataset to examine how the `drake` package works. We'll be using the November 5th, 2019, dataset of commuting via walking/biking in US Cities 2008-2012. It can be accessed from here:

`https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2019/2019-11-05/commute.csv`



Data Dictionary

`commute.csv`

variable	class	description
city	character	City
state	character	State
city_size	character	City Size * Small = 20K to 99,999 * Medium = 100K to 199,999 * Large = >= 200K
mode	character	Mode of transport, either walk or bike
n	double	N of individuals
percent	double	Percent of total individuals
moe	double	Margin of Error (percent)
state_abb	character	Abbreviated state name
state_region	character	ACS State region

Getting started with drake

We first need to install the drake package with
`install.packages("drake")`

When writing our script for visualising and understanding the dataset, we need to write our code chunks as functions.When we write our drake plan (which details the steps of our analysis) we list the various steps in our workflow - these are called *targets*.

Associated with each target is the command that is executed. These commands can be simple R commands, chunks of code or functions that you have written for each analysis step.

Developing our workflow...

Let's develop an workflow plan that does the following:

- Reads in our data.
- Generates a plot to visualise the top 10 States in the US with the highest % of people commuting via walking.
- Generates a plot to visualise the % of people commuting via walking for small vs. medium vs. large sized cities.
- Generates summary descriptives of the mean and sd of % people commuting via walking for small vs. medium vs. large sized cities.
- Builds a linear model to examine how % of people commuting via walking is predicted by the size of a US city.
- Produces a summary of the output of this model.

Reading in our data...

Below I create a function called read_my_data() which simply reads in a datafile. You don't need to pass any parameters when you call the function, as in this case the location of where the data are to be loaded in from is hardwired into the body of the function itself.

```
read_my_data <- function(x){
   read_csv("https://raw.githubusercontent.com/rfordatascienc
   e/tidytuesday/master/data/2019/2019-11-05/commute.csv")
  }
```

Calling the function with `read_my_data()` will return a tibble of the dataset.

```
read_my_data()
Parsed with column specification:
cols(
  city = col_character(),
  state = col_character(),
  city_size = col_character(),
  mode = col_character(),
  n = col_double(),
  percent = col_double(),
  moe = col_double(),
  state_abb = col_character(),
  state_region = col_character()
)
# A tibble: 3,496 x 9
   city              state          city_size mode      n percent   moe state_abb state_region
   <chr>             <chr>          <chr>     <chr> <dbl>   <dbl> <dbl> <chr>     <chr>
 1 Aberdeen city     South Dakota  Small      Bike 110  0.8   0.5 SD        North Centr…
 2 Acworth city      Georgia        Small     Bike      0     0     0.4 GA        South
 3 Addison village   Illinois       Small     Bike      43    0.2   0.3 IL        North Centr…
 4 Adelanto city     California     Small     Bike      0     0     0.5 CA        West
 5 Adrian city       Michigan       Small     Bike 121  1.5   1   MI        North Centr…
 6 Agawam Town city Massachusetts Small      Bike      0     0     0.2 MA        Northeast
 7 Agoura Hills ci… California     Small     Bike      84    0.8   1.1 CA        West
 8 Aiken city        South Caroli… Small      Bike      23    0.2   0.3 SC        South
 9 Alabaster city    Alabama        Small     Bike      0     0     0.2 AL        South
10 Alameda city      California     Small     Bike 576  1.5   0.4 CA        West
# … with 3,486 more rows
```

Generate a plot to visualise the top 10 States in the US with the highest % of people commuting via walking

The following function takes a tibble, and generates the plot:

```
my_overall_plot <- function(x) {
  x %>%
      group_by(state, mode) %>%
      summarise(mean_percent = mean(percent)) %>%
      ungroup() %>%
      filter(mode == "Walk") %>%
      arrange(-mean_percent) %>%
      top_n(10, mean_percent) %>%
      ggplot(aes(x = fct_reorder(state, mean_percent, median),
                 y = mean_percent, fill = state)) +
      geom_col() +
      guides(fill = FALSE) +
      coord_flip() +
      labs(x = "State", y = "Percentage of Walkers",
      title = "States with the Highest Percentage of Walkers") +
      theme(text = element_text(size = 10)) +
      theme_minimal()
}
```

Generate a plot to visualise the % of people commuting via walking for small vs. medium vs. large sized cities

```
my_walk_plot <- function(x) {
  x %>%
    filter(mode == "Walk") %>%
    group_by(city_size) %>%
    ggplot(aes(x = city_size, y = percent, colour = city_size)) +
    geom_jitter(width = .1, size = 3, alpha = .25) +
    guides(colour = FALSE) +
    labs(title = "% of Walkers by City Size",
         x = "City Size",
         y = "Percent of Walkers") +
    theme(text = element_text(size = 12))
}
```

Generate summary descriptives of the mean and sd of % people commuting via walking for small vs. medium vs. large sized cities

```
desc_stats <- function(x) {
  x %>%
    filter(mode == "Walk") %>%
    group_by(city_size) %>%
    summarise(mean_walk = mean(percent), sd_walk = sd(percent))
  }
```

We can then build our drake plan for our 6 analysis steps and call the functions we have just written…

```
my_plan <- drake_plan(commute_mode = read_my_data(),
                      show_overall_plot = my_overall_plot(commute_mode),
                      show_walk_plot = my_walk_plot(commute_mode),
                      show_stats = desc_stats(commute_mode),
                      fit = lm(percent ~ city_size,
                              data = filter(commute_mode, mode == "Walk")),
                      summary_fit = summary(fit))
```

We have 6 targets - `commute_mode, show_overall_plot, show_walk_plot, show_stats, fit, and summary_fit`

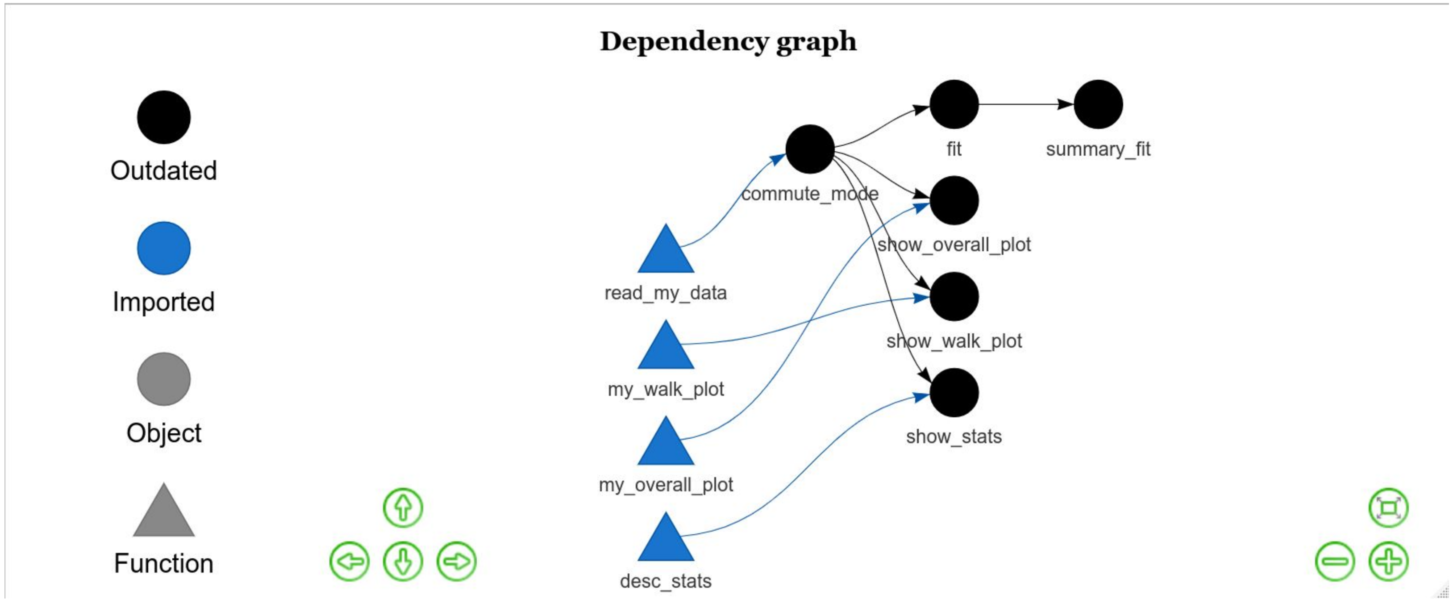
Associated with each target is the appropriate code - for targets 1, 2, 3, and 4 this is for functions we have written. For targets 5 and 6 this is for in-built R functions.

Our drake plan...

```
> my_plan
# A tibble: 6 x 2
  target            command
  <chr>             <expr>
1 commute_mode      read_my_data()
2 show_overall_plot my_overall_plot(commute_mode)
3 show_walk_plot    my_walk_plot(commute_mode)
4 show_stats        desc_stats(commute_mode)
5 fit               lm(percent ~ city_size, data = filter(commute_mode, mode == "Walk"))
6 summary_fit       summary(fit)
```

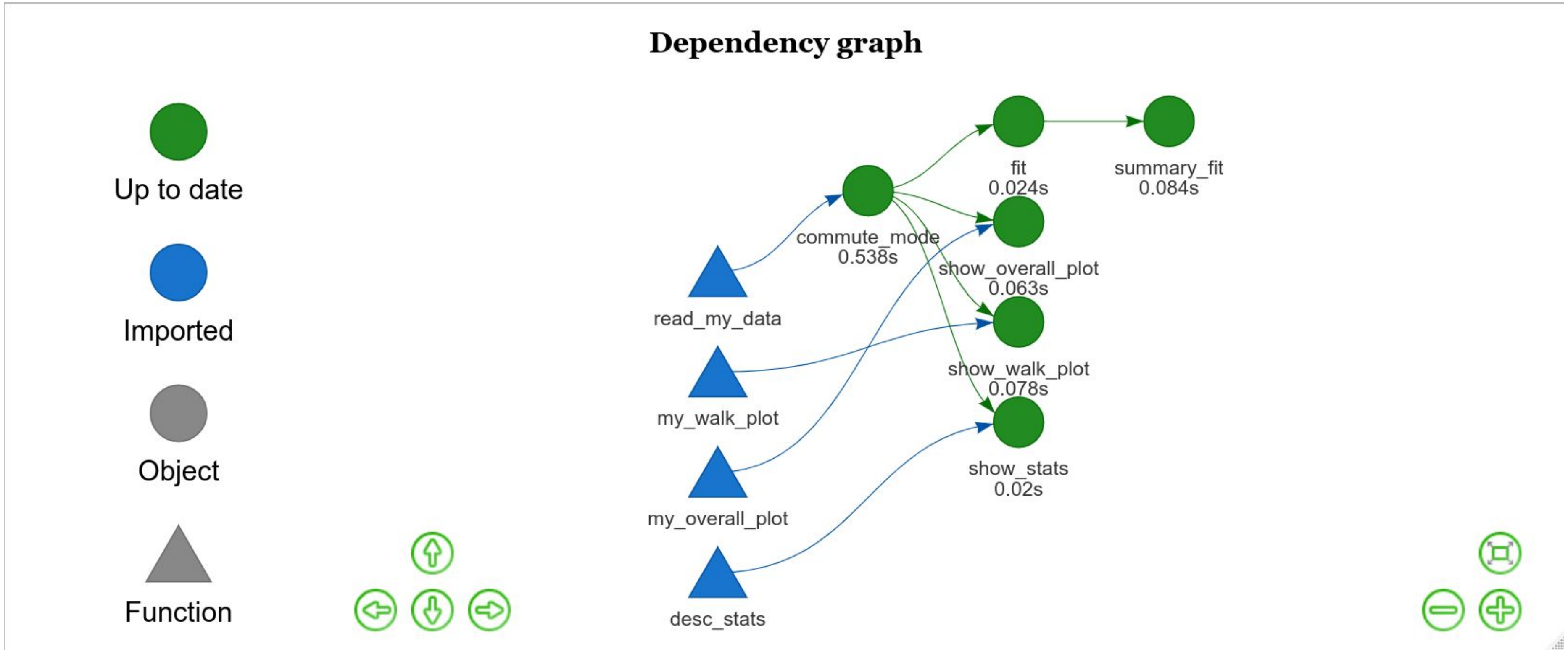
Our drake plan...

```
vis_drake_graph(my_plan)
```



Executing our plan

```
> make(my_plan)
▶️ target commute_mode
Parsed with column specification:
cols(
  city = col_character(),
  state = col_character(),
  city_size = col_character(),
  mode = col_character(),
  n = col_double(),
  percent = col_double(),
  moe = col_double(),
  state_abb = col_character(),
  state_region = col_character()
)
▶️ target fit
▶️ target show_overall_plot
`summarise()` regrouping output by 'state' (override with `.groups` argument)
▶️ target show_walk_plot
▶️ target show_stats
`summarise()` ungrouping output (override with `.groups` argument)
▶️ target summary_fit
```


Dependency graph
Up to date
Imported
Object
Function
read_my_data
my_walk_plot
my_overall_plot
desc_stats
commute_mode
0.538s
fit
0.024s
summary_fit
0.084s
show_overall_plot
0.063s
show_walk_plot
0.078s
show_stats
0.02s

Analysing your drake history

```
> drake_history(analyze = TRUE)
# A tibble: 6 x 8
  target   current built        exists hash     command                        seed runtime
  <chr>    <lgl>   <chr>        <lgl>  <chr>    <chr>                         <int>   <dbl>
1 commute_m… TRUE   2020-09-28 1… TRUE   4444ca6… "read_my_data()"         2.76e8 0.525
2 fit     TRUE 2020-09-28 1… TRUE   b2d8814… "lm(percent ~ city_size,…  1.11e9 0.00500
3 show_over… TRUE   2020-09-28 1… TRUE   f8793d5… "my_overall_plot(commute…  1.85e9 0.028
4 show_stats TRUE   2020-09-28 1… TRUE   7d80d90… "desc_stats(commute_mode…  1.08e9 0.0160
5 show_walk… TRUE   2020-09-28 1… TRUE   8340ac7… "my_walk_plot(commute_mo…  1.67e9 0.033
6 summary_f… TRUE   2020-09-28 1… TRUE   741674a… "summary(fit)"           2.09e9 0.00100
```

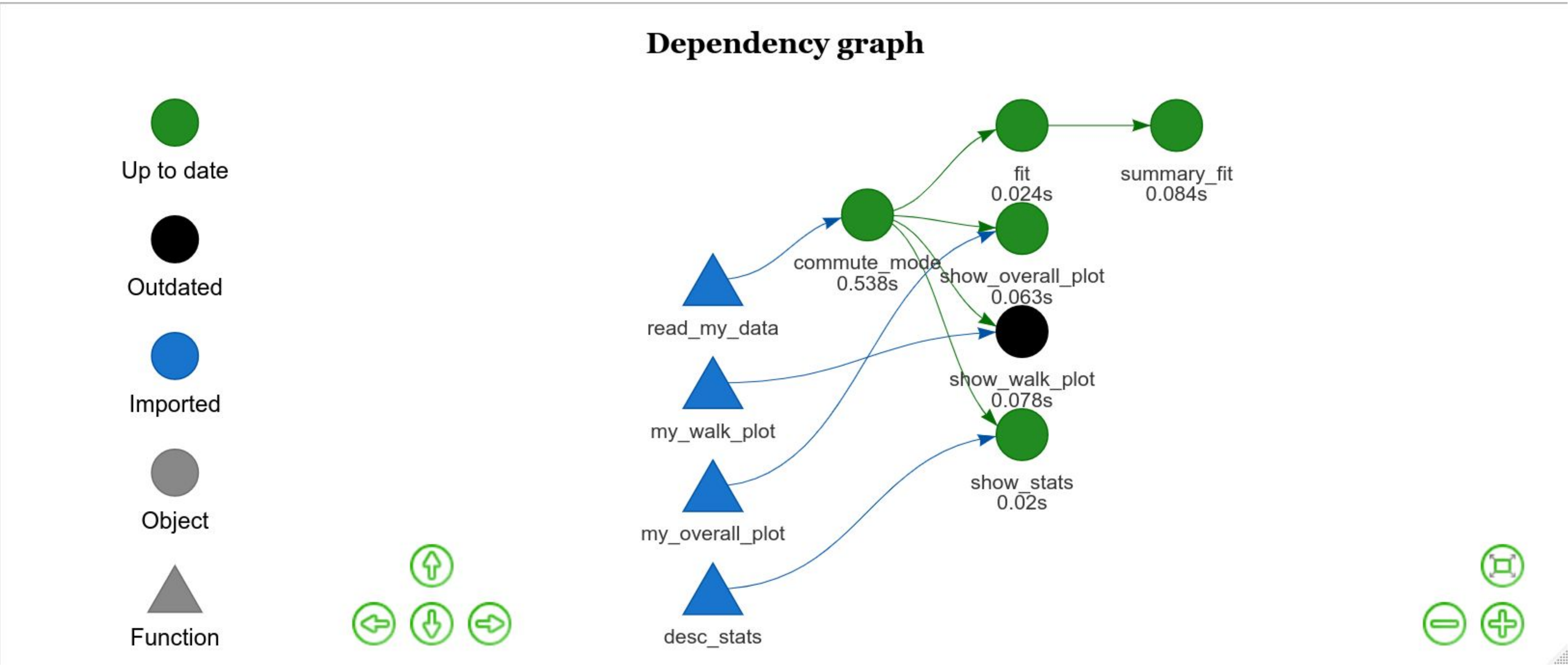
`drake` keeps track of which components have been run so if you were to update your code and (re)make your drake plan, only the bits of code that have changed will be run. You can look at the runtime column to see which bit of code took the longest to execute.This is super useful as you don't want to have to re-run computationally intensive bits of code again - and only the bits you have changed.

Let’s change a small bit of our code...

Let's change one of our plot functions - let's keep is simple and just add a `coord_flip()` to the `my_walk_plot()` function:

```
my_walk_plot <- function(x) {
  x %>%
    filter(mode == "Walk") %>%
    group_by(city_size) %>%
    ggplot(aes(x = city_size, y = percent, colour = city_size)) +
    geom_jitter(width = .1, size = 3, alpha = .25) +
    guides(colour = FALSE) +
    labs(title = "% of Walkers by City Size",
        x = "City Size",
        y = "Percent of Walkers") +
    theme(text = element_text(size = 12)) +
    coord_fpip()
}
```

Let's run that function again and then look at the configuration of our plan with `vis_drake_graph(my_plan)`



The `show_walk_plot()` function is in black to indicate it is outdated.

If we now (re)make our drake plan, only that component is actually run!

```
> make(my_plan)
target show_walk_plot
```

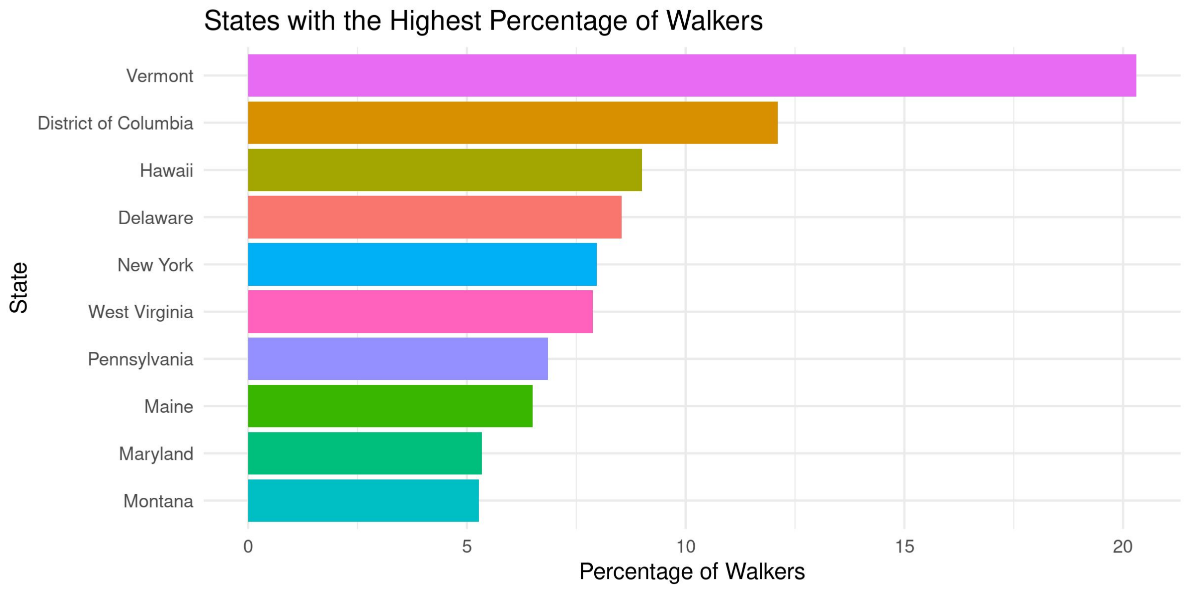
And if we look at `drake_history()` we see that the `show_walk_plot()` function has now been run twice - with the most recent version being marked a the current one.

```
drake_history(analyze = TRUE)
# A tibble: 7 x 8
  target        current built        exists hash     command                          seed runtime
  <chr>         <lgl>   <chr>        <lgl>  <chr>    <chr>                           <int>   <dbl>
1 commute_m…    TRUE 2020-09-28 1… TRUE   4444ca6… "read_my_data()"           2.76e8 0.525
2 fit           TRUE 2020-09-28 1… TRUE   b2d8814… "lm(percent ~ city_size,…  1.11e9 0.00500
3 show_over…    TRUE 2020-09-28 1… TRUE   f8793d5… "my_overall_plot(commute…  1.85e9 0.028
4 show_stats    TRUE 2020-09-28 1… TRUE   7d80d90… "desc_stats(commute_mode…  1.08e9 0.0160
5 show_walk…    FALSE   2020-09-28 1… TRUE   8340ac7… "my_walk_plot(commute_mo…  1.67e9 0.033
6 show_walk…    TRUE 2020-09-28 1… TRUE   8b4cf9e… "my_walk_plot(commute_mo…  1.67e9 0.0100
7 summary_f…    TRUE 2020-09-28 1… TRUE   741674a… "summary(fit)"             2.09e9 0.00100
```

When we run `make()`, drake stores the targets in a cache in the `.drake` folder.

Read and return a target from the drake cache

We can use the `readd()` function to read and return the contents of a target in our cache. So, `readd(show_overall_plot)` will return:



And `readd(summary_fit)` will return the output of the linear model summary.

```
> readd(summary_fit)

Call:
lm(formula = percent ~ city_size, data = filter(commute_mode, 
     mode == "Walk"))

Residuals:
   Min     1Q Median      3Q    Max 
-2.847 -1.847 -1.047  0.353 39.553 

Coefficients:
               Estimate Std. Error t value Pr(>|t|)    
(Intercept)    3.4422    0.3338  10.311   <2e-16 ***
city_sizeMedium  -0.4354      0.4248  -1.025   0.3056    
city_sizeSmall   -0.5950      0.3460  -1.720   0.0857 .  
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.485 on 1745 degrees of freedom
Multiple R-squared:  0.001796,    Adjusted R-squared:  0.0006519 
F-statistic:  1.57 on 2 and 1745 DF,  p-value: 0.2084
```

A consistent and reliable workflow using `r_make()`

If you change your code interactively (i.e., as you work in RStudio), you might end up making changes that accidentally invalidate targets - but you won't necessarily notice at the time.

One way to avoid this issue is to build your workflow in a new, temporary, R session. This will result in a more consistent and reliable workflow.

Setting things up for `r_make()`

You need to have a configuration script (the default is called `_drake.R` sitting above another folder (which we're calling R) which contains separate files for each bit of your make plan (and you need the drake plan itself as a separate file):

```
_drake.R
R/
   ├── packages.R
   ├── functions.R
   └── plan.R
```

Here we have 3 .R files saved in a folder I've called R.

Setting things up for `r_make()`

Our `_drake.R` file contains the following 4 lines of code:

```
source("R/packages.R")
source("R/functions.R")
source("R/plan.R")
drake_config(my_plan, verbose = 2)
```

By setting `verbose = 2`, we get a progress bar and update as the plan targets are made. The `source()` function reads the code from the named location - this means that with `source("R/functions.R")` you can then use the functions contained within this file in the current R session.

Setting things up for `r_make()`

Our packages.R file contains these 2 lines of code:

```
library(drake)
library(tidyverse)
```

our `functions.R` file contains the code for all our functions such as the `desc_stats()` function here:

```
desc_stats <- function(x) {
    x %>%
        filter(mode == "Walk")
        %>% group_by(city_size)
        %>% summarise(mean_walk = mean(percent), sd_walk = sd(percent))
}
```

Setting things up for `r_make()`

Our `plan.R` file is simply the code for our drake plan:

```
my_plan <- drake_plan(commute_mode = read_my_data(),
                      show_overall_plot = my_overall_plot(commute_mode),
                      show_walk_plot = my_walk_plot(commute_mode),
                      show_stats = desc_stats(commute_mode),
                      fit = lm(percent ~ city_size,
                              data = filter(commute_mode, mode == "Walk")),
                      summary_fit = summary(fit))
```

With our config file set up and your .R files associated with loading your packages, functions etc. all in the right place you can simply type `r_make()` and your plan etc. will be loaded and run in your new temporary environment.

```
> r_make()
── Attaching packages ─────────────────────────────────────── tidyverse 1.3.0 ──
✓ ggplot2 3.3.2     ✓ purrr   0.3.4
✓ tibble  3.0.1     ✓ dplyr   1.0.0
✓ tidyr   1.1.0     ✓ stringr 1.4.0
✓ readr   1.3.1     ✓ forcats 0.5.0
── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
x dplyr::filter() masks stats::filter()
x dplyr::lag() masks stats::lag()
/
Attaching package: 'drake'

The following objects are masked from 'package:tidyr':

     expand, gather

✓ All targets are already up to date.
```

A few other things...

If you want to force targets to be out of date in the cache (maybe you want to re-run everything) you can use the `clean()` function.

If you accidentally delete targets that you didn't mean to, you can try to recover them using

```
make(my_plan, recover = TRUE)
```

You can find out the dependencies for your targets like this:

```
deps_target("show_walk_plot", my_plan)

# A tibble: 2 x 3
  name         type       hash
  <chr>        <chr>      <chr>
1 my_walk_plot globals    e865098e79741023
2 commute_mode globals    4444ca621fdd0dd5
```

A few other things...

For analyses that can take hours (or days) to run locally, you can throw your drake plan (and hence analysis) up to an HPC cluster, run it as a persistent background process, run targets in parallel, or use multiple cores on your own machine.

`drake` is really good in terms of scaleability.

More info. about these aspects here:

https://books.ropensci.org/drake/hpc.html