Introduction to Data Science

Session 3: Programming II

Simon Munzert
Hertie School | GRAD-C11/E1339

Table of contents

Iteration

Iteration

The ubiquity of iteration

- Often we have to run the same task over and over again, with minor variations. Examples:
  - Standardize values of a variable
  - Recode all numeric variables in a dataset
  - Running multiple models with varying covariate sets
- A benefit of scripting languages in data (as opposed to point-and-click solutions) is that we can easily automate the process of iteration

Ways to iterate

- A simple approach is to copy-and-paste code with minor modifications (→ "duplicate code", → "copy-and-paste programming"). This is lazy, error-prone, not very efficient, and violates the "Don't repeat yourself" (DRY) principle.
- In R, vectorization, that is applying a function to every element of a vector at once, already does a good share of iteration for us.
- `for()` loops are intuitive and straightforward to build, but sometimes not very efficient.
- Finally, we learned about functions. Now, we learn how to unleash their power by applying them to anything we interact with in R at scale.

A simple example

Task

Say we want to double each element in a numeric vector, `x = c(1, 2, 3, 4, 5)`. Here are some different approaches to achieve this:

1. Manually (sometimes: copying and pasting code)

```
R> x ← c(1, 2, 3, 4, 5)
R> x_doubled ← c(2, 4, 6, 8, 10)
```

2. Vectorization

```
R> x ← c(1, 2, 3, 4, 5)
R> x_doubled ← x * 2
R> x_doubled
```

```
 [1]  2  4  6  8 10
```

3. `for()` loop

```
R> x ← c(1, 2, 3, 4, 5)
R> x_doubled ← numeric(length(x))
R> for (i in seq_along(x)) {
+   x_doubled[i] ← x[i] * 2
+ }
R> x_doubled
```

```
 [1]  2  4  6  8 10
```

4. Using `purrr`

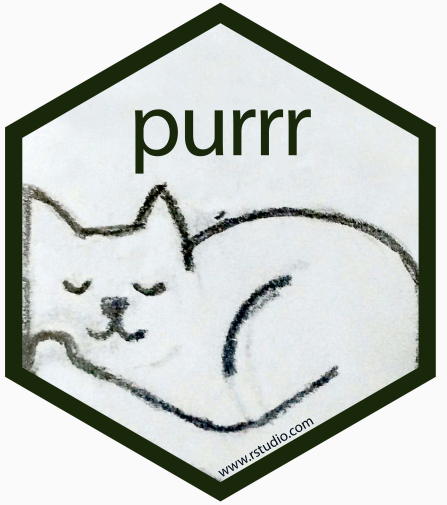
```
R> library(purrr)
R> x ← c(1, 2, 3, 4, 5)
R> x_doubled ← map_dbl(x, ~ .x * 2)
R> x_doubled
```

```
 [1]  2  4  6  8 10
```

Iteration with purrr

The tidyverse way to iterate

- For *real* functional programming in base R, we can use the `*apply()` family of functions (`lapply()`, `sapply()`, etc.). See here for an excellent summary.
- In the tidyverse, this functionality comes with the `purrr` package.
- At its core is the `map*()` family of functions.



How `purrr` works

- The idea is always to **apply** a function to **x**, where x can be a list, vector, data.frame, or something more complex.
- The output is then returned as output of a pre-defined type (e.g., a list).

Iteration with purrr: map()

The `map*()` functions all follow a similar syntax:

$$\text{map}(.\,\text{x}, .\,\text{f}, ...)$$

We use it to apply a function `.f` to each piece in `.x`. Additional arguments to `.f` can be passed on in `...`.

For instance, if we want to identify the object class of every column of a data.frame, we can write:

```
R> map(starwars, class)
```

```
$name
[1] "character"

$height
[1] "integer"

$mass
[1] "numeric"

$hair_color
[1] "character"

$skin_color
[1] "character"
```

Iteration with purrr: map() *cont.*

By default, `map()` returns a list. But we can also use other `map*()` functions to give us an atomic vector of an indicated type (e.g., `map_int()` to return an integer vector, or `map_vec()` to return a vector that is the simplest common type).

Going back to the previous example, we can also use `map_chr()`, which returns a character vector:

```
R> map_chr(starwars, class)
```

```
       name      height        mass  hair_color  skin_color   eye_color
"character"   "integer"   "numeric" "character" "character" "character"
 birth_year         sex      gender   homeworld     species       films
  "numeric" "character" "character" "character" "character"      "list"
   vehicles   starships
     "list"      "list"
```

The `purrr` function set is quite comprehensive. Be sure to check out the cheat sheet and the tutorials. You'll survive without `purrr` but you probably don't want to live without it. Together with `dplyr` it's easily the most powerful package for data wrangling in the tidyverse. If you master it, it will save you a lot of time and headaches.

Iteration with purrr: map() *cont.*

Apply functions with purrr : : CHEATSHEET

Map Functions

ONE LIST

map(.x, .f, …**)** Apply a function to each element of a list or vector, and return a list.

```
x <- list(a = 1:10, b = 11:20, c = 21:30)
l1 <- list(x = c("a", "b"), y = c("c", "d"))
map(l1, sort, decreasing = TRUE)
```

map_dbl(.x, .f, …**)**
Return a double vector.
map_dbl(x, mean)

map_int(.x, .f, ...**)**
Return an integer vector.
map_int(x, length)

map_chr(.x, .f, …**)**
Return a character vector.
map_chr(l1, paste, collapse = "")

map_lgl(.x, .f, …**)**
Return a logical vector.
map_lgl(x, is.integer)

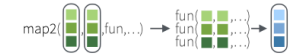
map_vec(.x, .f, ...**)**
Return a vector that is of the simplest common type.
map_vec(l1, paste, collapse = "")

walk(.x, .f, ...**)** Trigger side effects, return invisibly.
walk(x, print)

TWO LISTS

map2(.x, .y, .f, …**)** Apply a function to pairs of elements from two lists or vectors, return a list.

```
y <- list(1, 2, 3); z <- list(4, 5, 6); l2 <- list(x = "a", y = "z")
map2(x, y,\(x, y) x*y)
```



map2_dbl(.x, .y, .f, …**)** Return a double vector.
map2_dbl(y, z, ~ .x / .y)

map2_int(.x, .y, .f, …**)** Return an integer vector.
map2_int(y, z, `+`)

map2_chr(.x, .y, .f, …**)** Return a character vector.
map2_chr(l1, l2, paste, collapse = ",", sep = ":")

map2_lgl(.x, .y, .f, …**)** Return a logical vector.
map2_lgl(l2, l1, `%in%`)

map2_vec(.x, .f, ...**)**
Return a vector that is of the simplest common type.
map2_vec(l1, l2, paste, collapse = ",", sep = ":")

walk2(.x, .y, .f, ...**)** Trigger side effects, return invisibly.
walk2(objs, paths, save)

imap(.x, .f, ...) is shorthand for **map2(.x, names(.x), .f)** or **map2(.x, seq_along(.x), .f)** depending on whether **.x** is named or not.

MANY LISTS

pmap(.l, .f, …**)** Apply a function to groups of elements from a list of lists or vectors, return a list.

```
pmap(
  list(x, y, z),
  function(first, second, third) first * (second + third)
)
```

pmap_dbl(.l, .f, …**)**
Return a double vector.
pmap_dbl(list(y, z), ~ .x / .y)

pmap_int(.l, .f, …**)**
Return an integer vector.
pmap_int(list(y, z), `+`)

pmap_chr(.l, .f, …**)**
Return a character vector.
pmap_chr(list(l1, l2), paste, collapse = ",", sep = ":")

pmap_lgl(.l, .f, …**)**
Return a logical vector.
pmap_lgl(list(l2, l1), `%in%`)

pmap_vec(.l, .f, …**)**
Return a vector that is of the simplest common type.
pmap_vec(list(l1, l2), paste, collapse = ",", sep = ":")

pwalk(.l, .f, ...**)** Trigger side effects, return invisibly.
pwalk(list(objs, paths), save)

Function Shortcuts

Use **\(x)** with functions like **map()** that have single arguments.

map(l, \(x) x + 2)
becomes
map(l, function(x) x + 2)

Use **\(x, y)** with functions like **map2()** that have two arguments.

map2(l, p, \(x, y) x + y)
becomes
map2(l, p, function(l, p) l + p)

Use **\(x, y, z)** etc with functions like **pmap()** that have many arguments.

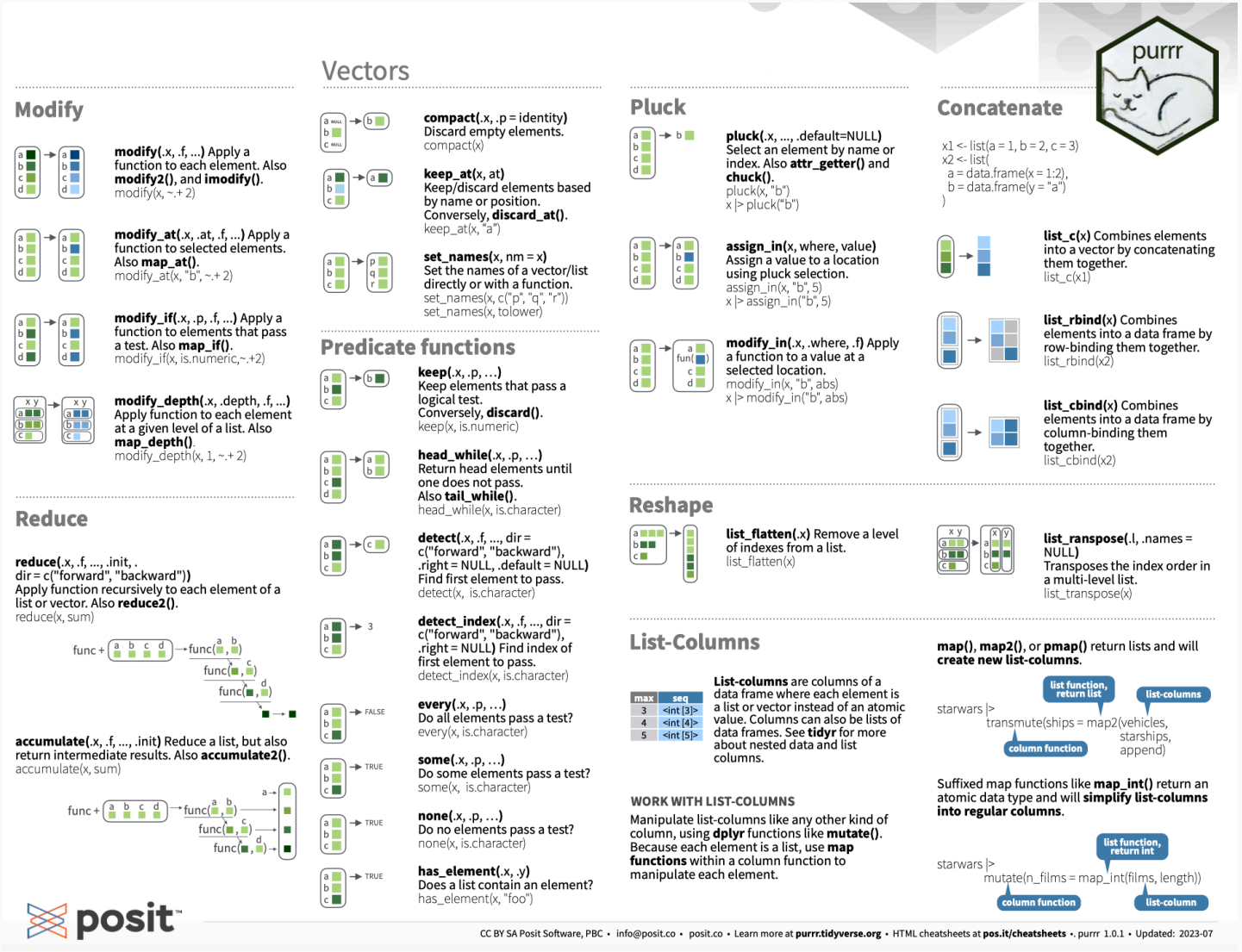
pmap(list(x, y, z), \(x, y, z) x + y / z)
becomes
pmap(list(x, y, z), function(x, y, z) x * (y + z))

Use **\(x, y)** with functions like **imap()**. .x will get the list value and .y will get the index, or name if available.

imap(list("a", "b", "c"), \(x, y) paste0(y, ": ", x))
outputs **"index: value"** for each item

Use a **string** or an **integer** with any map function to index list elements by name or position. **map(l, "name")** becomes **map(l, function(x) x[["name"]])**

Iteration with purrr: map() *cont.*



Another example

Task

Let's say we want to calculate the mean and standard deviation of height and mass for each species in the `starwars` dataset.

```
R> # Load the starwars dataset
R> data(starwars)
R>
R> # Custom function for calculations
R> calc_stats ← function(df) {
+   df %>%
+     summarise(
+       height_mean = mean(height, na.rm = TRUE),
+       height_sd = sd(height, na.rm = TRUE),
+       mass_mean = mean(mass, na.rm = TRUE),
+       mass_sd = sd(mass, na.rm = TRUE)
+     )
+ }
```

```
R> # Group by species and apply the custom function
R> species_stats ← starwars %>%
+   group_by(species) %>%
+   nest() # Nesting the data
R> species_stats
```

```
  # A tibble: 38 × 2
  # Groups:   species [38]
     species        data
     <chr>          <list>
   1 Human          <tibble [35 × 13]>
   2 Droid          <tibble [6 × 13]>
   3 Wookiee        <tibble [2 × 13]>
   4 Rodian         <tibble [1 × 13]>
   5 Hutt           <tibble [1 × 13]>
   6 <NA>           <tibble [4 × 13]>
   7 Yoda's species <tibble [1 × 13]>
   8 Trandoshan     <tibble [1 × 13]>
   9 Mon Calamari   <tibble [1 × 13]>
  10 Ewok           <tibble [1 × 13]>
  # ℹ 28 more rows
```

Another example

Task

Let's say we want to calculate the mean and standard deviation of height and mass for each species in the `starwars` dataset.

```
R> # Load the starwars dataset
R> data(starwars)
R>
R> # Custom function for calculations
R> calc_stats ← function(df) {
+   df %>%
+     summarise(
+       height_mean = mean(height, na.rm = TRUE),
+       height_sd = sd(height, na.rm = TRUE),
+       mass_mean = mean(mass, na.rm = TRUE),
+       mass_sd = sd(mass, na.rm = TRUE)
+     )
+ }
```

```
R> # Group by species and apply the custom function
R> species_stats ← starwars %>%
+   group_by(species) %>%
+   nest() %>% # Nesting the data
+   # purrr magic
+   mutate(stats = map(data, calc_stats))
R> species_stats
```

```
  # A tibble: 38 × 3
  # Groups:   species [38]
     species        data               stats
     <chr>          <list>             <list>
   1 Human          <tibble [35 × 13]> <tibble [1 × 4]>
   2 Droid          <tibble [6 × 13]>  <tibble [1 × 4]>
   3 Wookiee        <tibble [2 × 13]>  <tibble [1 × 4]>
   4 Rodian         <tibble [1 × 13]>  <tibble [1 × 4]>
   5 Hutt           <tibble [1 × 13]>  <tibble [1 × 4]>
   6 <NA>           <tibble [4 × 13]>  <tibble [1 × 4]>
   7 Yoda's species <tibble [1 × 13]>  <tibble [1 × 4]>
   8 Trandoshan     <tibble [1 × 13]>  <tibble [1 × 4]>
   9 Mon Calamari   <tibble [1 × 13]>  <tibble [1 × 4]>
  10 Ewok           <tibble [1 × 13]>  <tibble [1 × 4]>
```

Another example

Task

Let's say we want to calculate the mean and standard deviation of height and mass for each species in the `starwars` dataset.

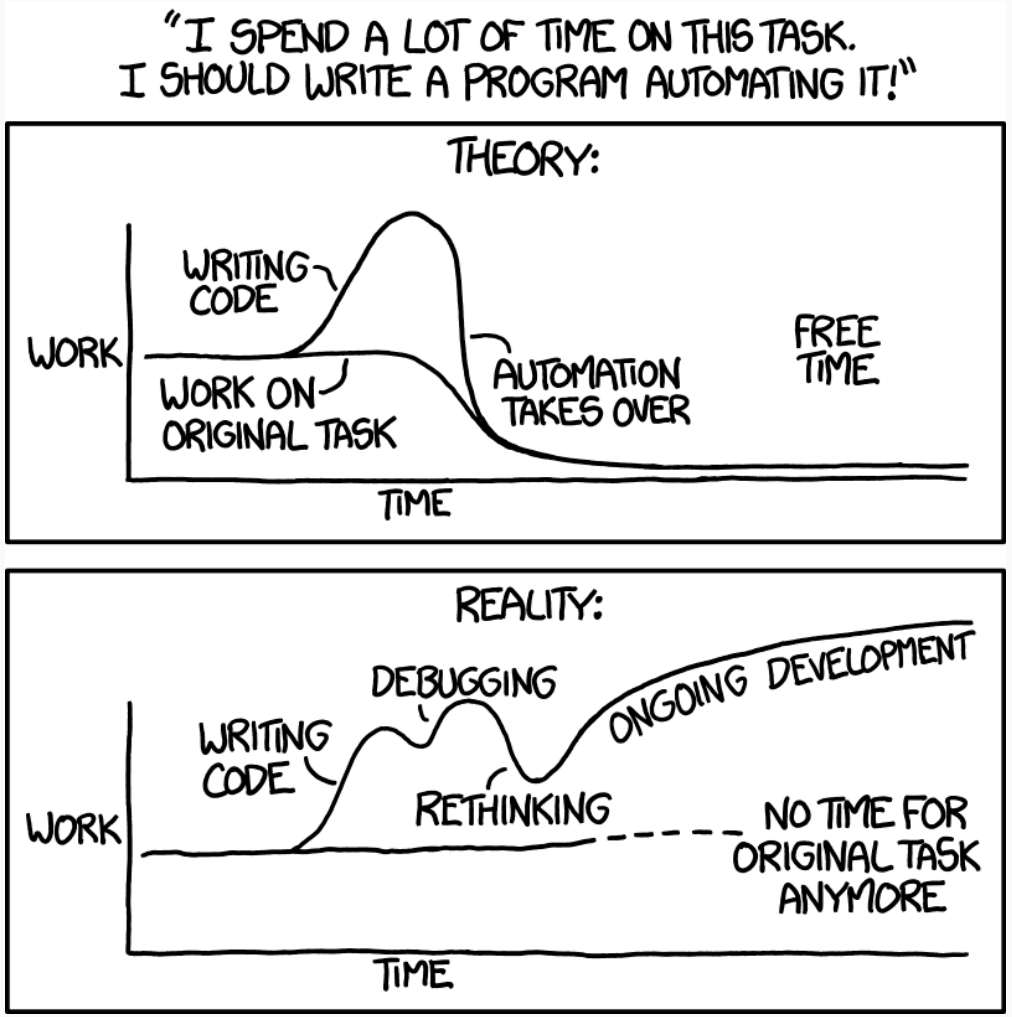
```
R> # Load the starwars dataset
R> data(starwars)
R>
R> # Custom function for calculations
R> calc_stats ← function(df) {
+   df %>%
+     summarise(
+       height_mean = mean(height, na.rm = TRUE),
+       height_sd = sd(height, na.rm = TRUE),
+       mass_mean = mean(mass, na.rm = TRUE),
+       mass_sd = sd(mass, na.rm = TRUE)
+     )
+ }
```

```
R> # Group by species and apply the custom function
R> species_stats ← starwars %>%
+   group_by(species) %>%
+   nest() %>% # Nesting the data
+   # purrr magic
+   mutate(stats = map(data, calc_stats)) %>%
+   select(species, stats) %>% # Select columns
+   unnest(stats) # Unnest the data
R> species_stats
```

```
  # A tibble: 38 × 5
  # Groups:   species [38]
     species        height_mean height_sd mass_mean mass_
     <chr>                <dbl>     <dbl>     <dbl>   <dl
   1 Human                178       12.0       81.3    1
   2 Droid                131.      49.1       69.8    5
   3 Wookiee              231        4.24     124      1
   4 Rodian               173       NA         74      NA
   5 Hutt                 175       NA       1358      NA
   6 <NA>                 175       12.4       81      3
   7 Yoda's species        66       NA         17      NA
   8 Trandoshan           190       NA        113      NA
```

Automation and scripting

Automation



`Credit` Randall Munroe/xkcd 1319

Automation

Motivation

- We spend too much time on repetitive tasks.
- We're already automating using scripts that bundle multiple commands! Next step: The pipeline as a series of scripts and commands.
- Good pipelines are modular. But you don't want to trigger 10 scripts sequentially by hand.
- Some tasks are to be repeated on a regular basis (schedule).

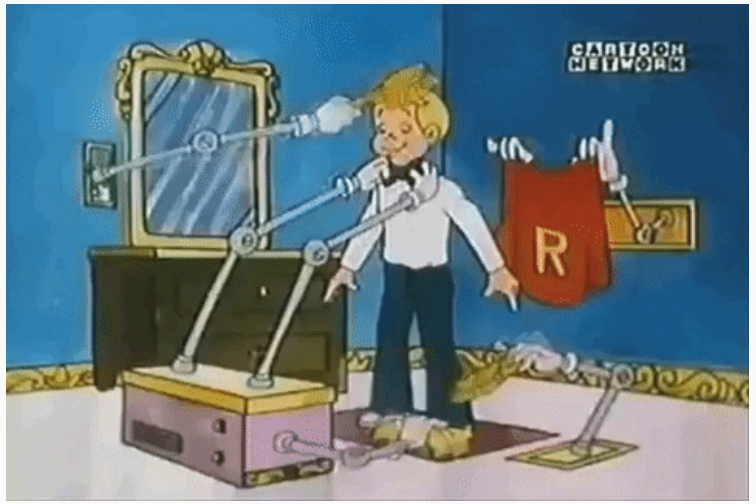
When automation makes sense

- The input is variable but the process of turning input into output is highly standardized.
- You use a diverse set of software to produce the output.
- Others (humans, machines) are supposed to run the analyses.
- Time saved by automation >> Time needed to automate.

Different ways of doing it

We will consider automation

- using **R**,
- using the **Shell** and **RScript**,
- using **make**, and
- using dedicated **scheduling tools**.



Thinking in pipelines

Key characteristics

- Pipelines make complex projects easier to handle because they break up a monolithic script into **discrete, manageable chunks**.
- If properly done, each stage of the pipeline defines its input and its outputs.
- Pipeline modules **do not modify their inputs** (*idempotence*). Rerunning one module produces the same results as the previous run.

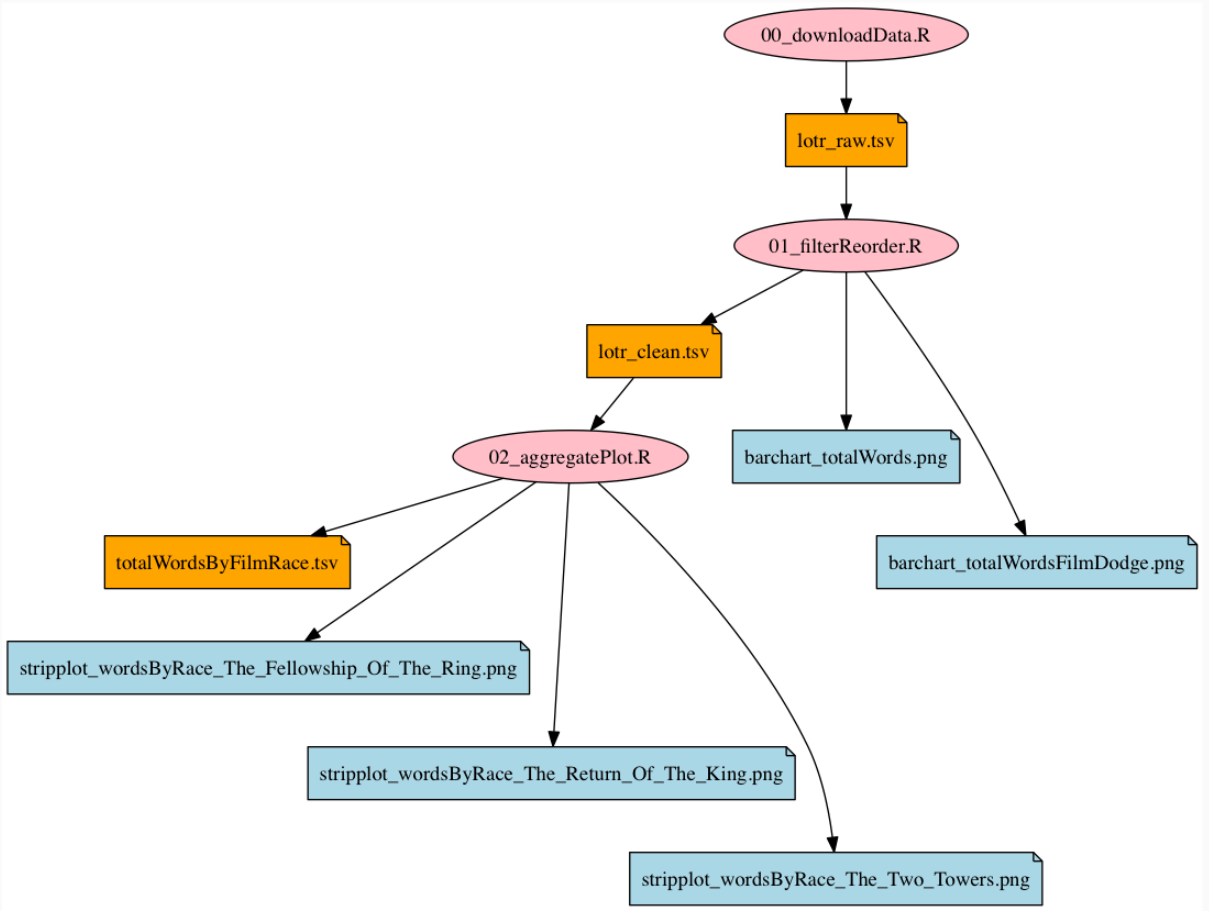
Key advantages

- When you modify one stage of the pipeline, you only have to rerun the downstream, dependent stages.
- Division of labor is straightforward.
- Modules tend to be a lot easier to debug.

A data science pipeline is a graph

Wait what

- Scripts and data files are vertices of the graph.
- Dependencies between stages are edges of the graph.
- Pipelines are not necessarily DAGS. Recursive routines are imaginable (but to be avoided?).
- Also, scripts are not necessarily hierarchical (e.g., multiple different modeling approaches of the same data in different scripts).
- An automation script gives *one* order in which you can successfully run the pipeline.



An example pipeline

In the following, we will work with this toy pipeline:[1]

[1] Courtesy of Jenny Bryan.

An example pipeline

In the following, we will work with this toy pipeline:

- `00-packages.R` loads the packages necessary for analysis,

`00-packages.R`:

```
R> # install packages from CRAN
R> p_needed ← c("tidyverse" # tidyverse packages
+ )
R> packages ← rownames(installed.packages())
R> p_to_install ← p_needed[!(p_needed %in% packages)]
R> if (length(p_to_install) > 0) {
+   install.packages(p_to_install)
+ }
R> lapply(p_needed, require, character.only = TRUE)
```

An example pipeline

In the following, we will work with this toy pipeline:

- `00-packages.R` loads the packages necessary for analysis,
- `01-download-data.R` downloads a spreadsheet, which is stored as `lotr_raw.tsv`,

`01-download-data.R`:

```
R> ## download raw data
R> download.file(url = "http://bit.ly/lotr_raw-tsv",
+                  destfile = "lotr_raw.tsv")
```

An example pipeline

In the following, we will work with this toy pipeline:

- `00-packages.R` loads the packages necessary for analysis,
- `01-download-data.R` downloads a spreadsheet, which is stored as `lotr_raw.tsv`,
- `02-process-data.R` imports and processes the data and exports a clean spreadsheet as `lotr_clean.tsv`, and

`02-process-data.R`:

```
R> ## import raw data
R> lotr_dat ← read_tsv("lotr_raw.tsv")
R>
R> ## reorder Film factor levels based on story
R> old_levels ← levels(as.factor(lotr_dat$Film))
R> j_order ← sapply(c("Fellowship", "Towers", "Return"),
+                   function(x) grep(x, old_levels))
R> new_levels ← old_levels[j_order]
R>
R> ## process data set
R> lotr_dat ← lotr_dat %>%
+   # apply new factor levels to Film
+     mutate(Film = factor(as.character(Film), new_levels),
+     # revalue Race
+     Race = recode(Race, `Ainur` = "Wizard", `Men` = "Man")) %>%
+ ## <skipping some steps here to avoid slide overflow>
+
+ ## write data to file
+ write_tsv(lotr_dat, "lotr_clean.tsv")
```

An example pipeline

In the following, we will work with this toy pipeline:

- `00-packages.R` loads the packages necessary for analysis,
- `01-download-data.R` downloads a spreadsheet, which is stored as `lotr_raw.tsv`,
- `02-process-data.R` imports and processes the data and exports a clean spreadsheet as `lotr_clean.tsv`, and
- `03-plot.R` imports the clean dataset, produces a figure and exports it as `barchart-words-by-race.png`.

`03-plot.R`:

```
R> ## import clean data
R> lotr_dat ← read_tsv("lotr_clean.tsv") %>%
+ # reorder Race based on words spoken
+ mutate(Race = reorder(Race, Words, sum))
R>
R> ## make a plot
R> p ← ggplot(lotr_dat, aes(x = Race, weight = Words)) + geom_bar()
R> ggsave("barchart-words-by-race.png", p)
```

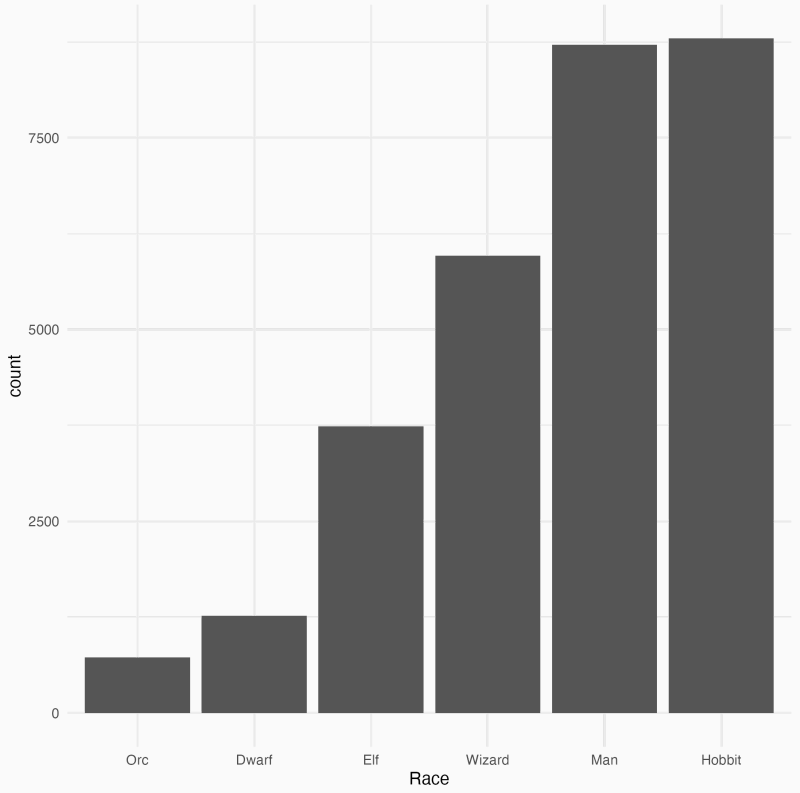
An example pipeline

```
R> slice_sample(lotr_dat, n = 10)
```

```
# A tibble: 10 × 5
   Film                      Chapter                       Character Race  Words
   <chr>                     <chr>                         <chr>     <chr> <dbl>
 1 The Return Of The King    64: The Mouth Of Sauron       Aragorn   Man      23
 2 The Fellowship Of The Ring 36: The Bridge Of Khazad-dûm Frodo     Hobb…     4
 3 The Two Towers            36: Isengard Unleashed        Saruman   Wiza…    50
 4 The Fellowship Of The Ring 42: The Great River          Sam       Hobb…    37
 5 The Return Of The King    42: Breaking The Gate Of Go… Gandalf   Wiza…    21
 6 The Two Towers            45: The Glittering Caves      Legolas   Elf      36
 7 The Two Towers            35: Helm's Deep               Rohan Wa… Man      22
 8 The Fellowship Of The Ring 33: Moria                    Aragorn   Man      31
 9 The Fellowship Of The Ring 43: Parth Galen              Aragorn   Man      79
10 The Return Of The King    24: Courage Is The Best Def… Gothmog   Orc       4
```

An example pipeline

```
R> p ← ggplot(lotr_dat, aes(x = Race, weight = Words)) +
+   geom_bar() + theme_minimal()
```



Automation using pipelines in R

Motivation and usage

- The `source()` function reads and parses R code from a file or connection.
- We can build a pipeline by sourcing scripts sequentially.
- This pipeline is usually stored in a "master/main" script.
- The removal of previous work is optional and maybe redundant. Often the data is overwritten by default.
- It is recommended that the individual scripts are (partial) standalones, i.e. that they import all data they need by default (loading the packages could be considered an exception).
- Note that as long as the environment is not reset, it remains intact across scripts, which is a potential source of error and confusion.

Example

The master script `master.R`:

```
R> ## clean out any previous work
R> outputs ← c("lotr_raw.tsv",
+               "lotr_clean.tsv",
+               list.files(pattern = "*.png$"))
R> file.remove(outputs)
R>
R> ## run scripts
R> source("00-packages.R")
R> source("01-download-data.R")
R> source("02-process-data.R")
R> source("03-plot.R")
```

Automation using the Shell and Rscript

Motivation and usage

- Alternatively to using an R master script, we can also run the pipeline from the command line.
- Note that here, the environments don't carry over across `Rscript` calls. The scripts definitely have to run in a standalone fashion (i.e., load packages, import all necessary data, etc.).
- The working directory should be set either in the script(s) or in the shell with `cd`.

Example

The master script `master.sh`:

```
#!/bin/sh
cd /Users/simonmunzert/github/examples/02-automation
set -eux
Rscript 01-download-data.R
Rscript 02-process-data.R
Rscript 03-plot.R
```

The `set` command allows to adjust some base shell parameters:

- `-e`: Stop at first error
- `-u`: Undefined variables are an error
- `-x`: Print each command as it is run

For more information on `set`, see here.

Automation using the Shell and Rscript

Motivation and usage

- Alternatively to using an R master script, we can also run the pipeline from the command line.
- Note that here, the environments don't carry over across `Rscript` calls. The scripts definitely have to run in a standalone fashion (i.e., load packages, import all necessary data, etc.).
- The working directory should be set either in the script(s) or in the shell with `cd`.
- One advantage of this approach is that it can be easily coupled with other command line tools, building a **polyglot pipeline**.

Example

The master script `master.sh`:

```
#!/bin/sh
cd /Users/simonmunzert/github/examples/02-automation
set -eux
curl -L http://bit.ly/lotr_raw-tsv > lotr_raw.tsv
Rscript 02-process-data.R
Rscript 03-plot.R
```

The `set` command allows to adjust some base shell parameters:

- `-e`: Stop at first error
- `-u`: Undefined variables are an error
- `-x`: Print each command as it is run

For more information on `set`, see here.

Automation using Make

Motivation and usage

- Make is an automation tool that allows us to specify and manage build processes.
- It is commonly run via the shell.
- At the heart of a make operation is the `makefile` (or `Makefile`, `GNUmakefile`), a script which serves as a recipe for the building process.
- A `makefile` is written following a particular syntax and in a declarative fashion.
- Conceptually, the recipe describes which files are built how and using what input.

Advantages of Make

- It looks at which files you have and automatically figures out how to create the files that you have. For complex pipelines this "automation of the automation process" can be very helpful.
- While shell scripts give *one* order in which you can successfully run the pipeline, Make will figure out the parts of the pipeline (and their order) that are needed to build a desired target.

Automation using Make (cont.)

Basic syntax

Each batch of lines indicates

- a file to be created (the target),
- the files it depends on (the prerequisites), and
- set of commands needed to construct the target from the dependent files.

Dependencies propagate.

- To create any of the `png` figures, we need `lotr_clean.tsv`.
- If this file changes, the `png`s change as well when they're built.

Example `makefile`

```
all: lotr_clean.tsv barchart-words-by-race.png words-histogram.png

lotr_raw.tsv:
    curl -L http://bit.ly/lotr_raw-tsv > lotr_raw.tsv

lotr_clean.tsv: lotr_raw.tsv 02-process-data.R
    Rscript 02-process-data.R

barchart-words-by-race.png: lotr_clean.tsv 03-plot.R
    Rscript 03-plot.R

words-histogram.png: lotr_clean.tsv
    Rscript -e 'library(ggplot2);
    qplot(Words, data = read.delim("$<"), geom = "histogram");
    ggsave("$@")'
    rm Rplots.pdf

clean:
    rm -f lotr_raw.tsv lotr_clean.tsv *.png
```

Automation using Make (cont.)

Getting Make to run

- Using the command line, go into the directory for your project.
- Create the `Makefile` file.
- The most basic Make commands are `make all` and `make clean` which builds (or deletes) all output as specified in the script.

Example `makefile`

```
all: lotr_clean.tsv barchart-words-by-race.png words-histogram.png

lotr_raw.tsv:
	curl -L http://bit.ly/lotr_raw-tsv > lotr_raw.tsv

lotr_clean.tsv: lotr_raw.tsv 02-process-data.R
	Rscript 02-process-data.R

barchart-words-by-race.png: lotr_clean.tsv 03-plot.R
	Rscript 03-plot.R

words-histogram.png: lotr_clean.tsv
	Rscript -e 'library(ggplot2);
	qplot(Words, data = read.delim("$<"), geom = "histogram");
	ggsave("$@")'
	rm Rplots.pdf

clean:
	rm -f lotr_raw.tsv lotr_clean.tsv *.png
```

Automation using Make - FAQ

Does it work on Windows?

To install an run `make` on Windows, check out these instructions.

Where can I learn more?

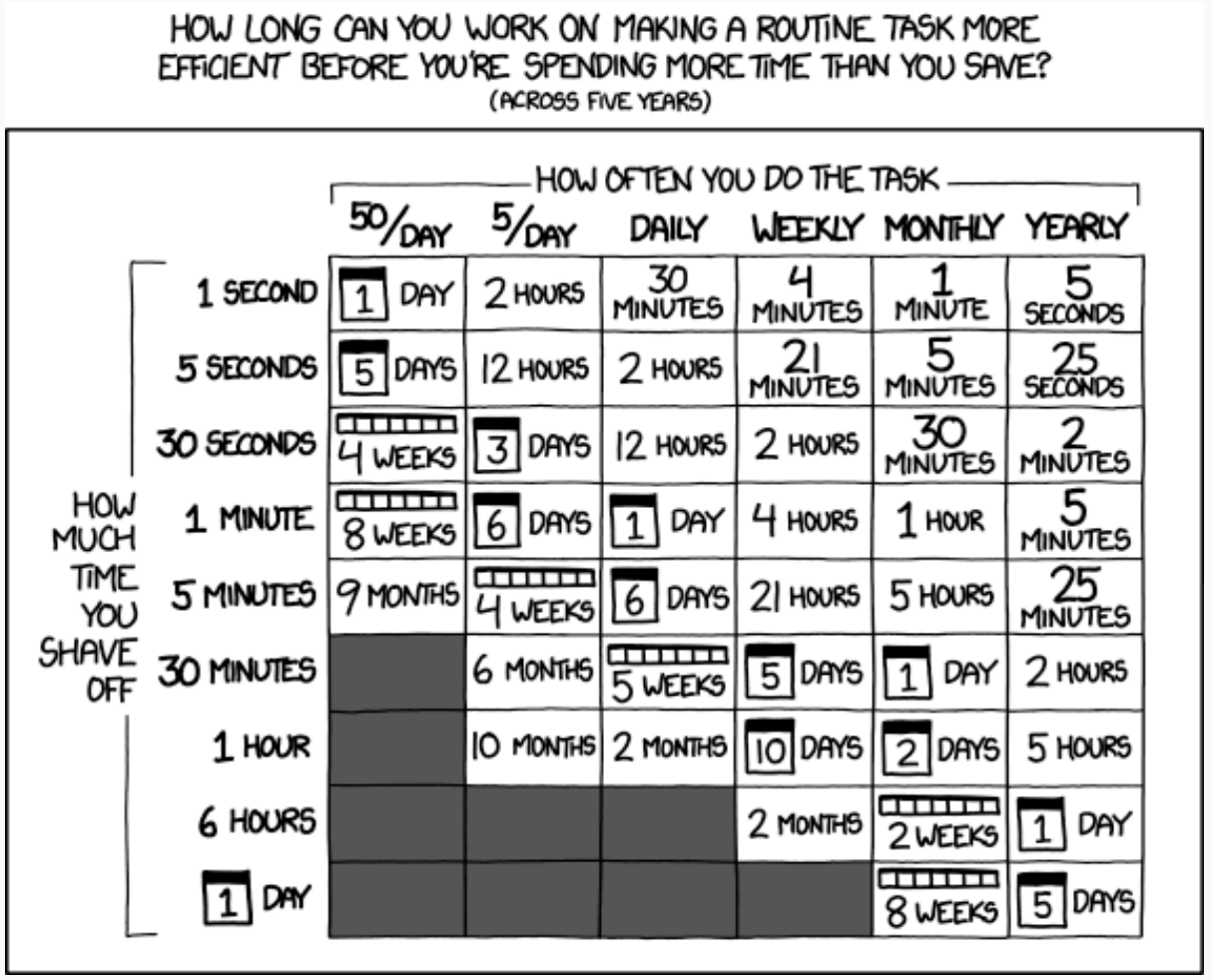
If you consider working with Make, check out the official manual, this helpful tutorial, Karl Broman's excellent minimal make introduction, or this Stat545 piece.

This is dusty technology. Are there alternatives?

In the context of data science with R, the `targets` package is an interesting option. It provides R functionality to define a Make-stype pipeline. Check out the overview and manual.

Scheduling

Scheduling



Credit Randall Munroe/xkcd 1205

Scheduling scripts and processes

Motivation

- So far, we have automated data science pipelines.
- But the execution of these pipelines still needs to be triggered.
- In some cases, it is desirable to also **automate the initialization** of R scripts (or any processes for that matter) **on a regular basis**, e.g. weekly, daily, on logon, etc.
- This makes particular sense when you have moving parts in your pipeline (most likely: data).

Common scenarios for scheduling

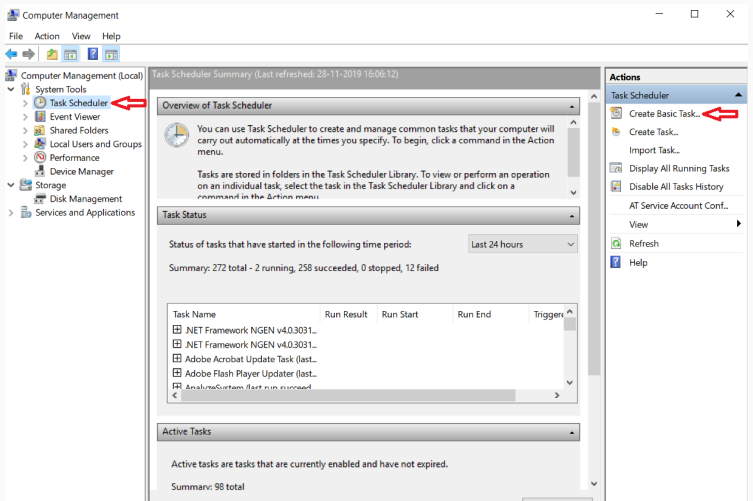
1. You fetch data from the web on a regular basis (e.g., via scraping scripts or APIs).
2. You generate daily/weekly/monthly reports/tweets based on changing data.
3. You build an alert control system informing you about anomalies in a database.

`Credit` Simone Giertz

Scheduling scripts and processes on Windows

Scheduling options

- Schedule tasks on Windows with Windows Task Scheduler.
- Manage them via a GUI (→ Control Panel) or the command line using `schtasks.exe`.
- The R package `taskscheduleR` provides a programmable R interface to the WTS.



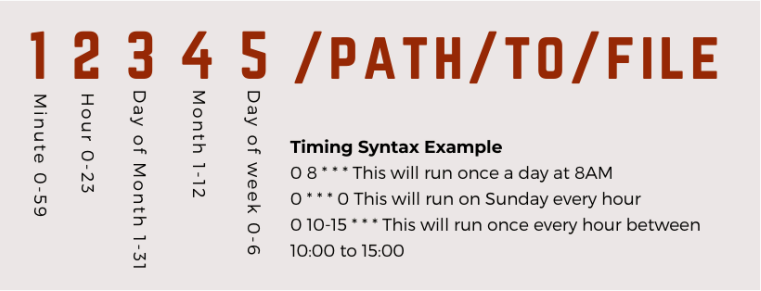
`taskscheduleR` example

```
R> library(taskscheduleR)
R> myscript ← "examples/scrape-wiki.R"
R> ## Run every 5 minutes, starting from 10:40
R> taskscheduler_create(
+    taskname = "WikiScraperR_5min", rscript = myscript,
+    schedule = "MINUTE", starttime = "10:40", modifier = 5)
R>
R> ## Run every week on Saturday and Sunday at 09:10
R> taskscheduler_create(
+    taskname = "WikiScraperR_SatSun", rscript = myscript,
+    schedule = "WEEKLY", starttime = "09:10",
+    days = c('SAT', 'SUN'))
R>
R> ## Delete task
R> taskscheduler_delete("WikiScraperR_SatSun")
R>
R> ## Get a data.frame of all tasks
R> tasks ← taskscheduler_ls()
R> str(tasks)
```

Scheduling scripts and processes on a Mac

Scheduling options

- On macOS you can schedule background jobs using `cron` and `launchd`.
- `launchd`[1] was created by Apple as a replacement for the popular Linux utility `cron` (deprecated but still usable).
- The R package `cronR` provides a programmable R interface.
- `cron` syntax for more complex scheduling:



`cronR` example

```
R> library(cronR)
R> myscript ← "examples/scrape-wiki.R"
R> # Create bash code for crontab to execute R script
R> cmd ← cron_rscript(myscript)
R>
R> ## Run every minute
R> cron_add(command = cmd, frequency = 'minutely',
+           id = 'ScraperR_1min', description = 'Every 1min')
R>
R> ## Run every 15 minutes (using cron syntax)
R> cron_add(cmd, frequency = '*/15 * * * *',
+           id = 'ScraperR_15min', description = 'Every 15 mins')
R>
R> ## Check number of running cronR jobs
R> cron_njobs()
R>
R> ## Delete task
R> cron_rm("WikiScraperR_1min", ask = TRUE)
```

[1]For more resources on scheduling with `launchd`, check out this and this.

Strategies for debugging

What's debugging?

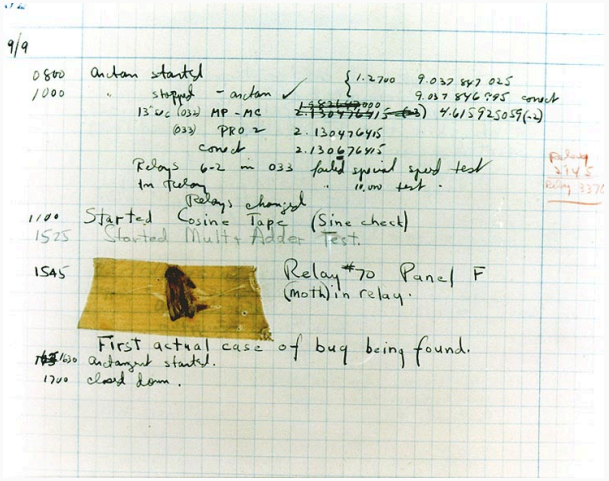
Straight from the Wikipedia

"Debugging is the process of finding and resolving bugs (defects or problems that prevent correct operation) within computer programs, software, or systems."

A famous (yet not the first) bug:

The term "bug" was used in an account by computer pioneer Grace Hopper (see on the right). While she was working on a Mark II computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system. This bug was carefully removed and taped to the log book (see on the right).

Above: Grace Hopper, *Below:* The bug



Why debugging matters

The Wikipedia list of software bugs with significant consequences is growing and you don't want to be on it.

NASA software engineers are famous for producing bug-free code. This was learned the hard and costly way though. Some highlights from space:

- 1962: A booster went off course during launch, resulting in the destruction of NASA Mariner 1 . This was the result of the failure of a transcriber to notice an overbar in a handwritten specification for the guidance program, resulting in an incorrect formula the FORTRAN code.
- 1999: NASA's Mars Climate Orbiter was destroyed, due to software on the ground generating commands based on parameters in pound-force (lbf) rather than newtons (N)
- 2004: NASA's Spirit rover became unresponsive on January 21, 2004, a few weeks after landing on Mars. Engineers found that too many files had accumulated in the rover's flash memory (the problem could be fixed though by deleting unnecessary files, and the Rover lived happily ever after. Until it froze to death in 2011).

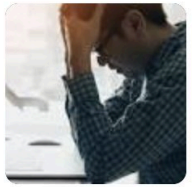
Why debugging matters (cont.)

ZDNet

Microsoft Excel blunder: Developers blamed for loss of thousands of COVID-19 test results

The error has hampered the UK's contact-tracing program at a time when the country is undergoing a second wave of coronavirus infections. By using the XLS ...

1 month ago



ABC News

Excel glitch leads to nearly 16,000 confirmed coronavirus cases going unreported in United Kingdom

For the test and trace program to work well, contacts should be notified as soon as possible. Health Secretary Matt Hancock told MPs that the problem related to ...

1 month ago

Metro

How were 16,000 Test and Trace coronavirus cases lost on Excel?

The cases were lost due to a technical error on a Microsoft Excel spreadsheet. ... Trace 'immediately' after the issue was resolved and thanked contact tracers for ...

1 month ago

Does Contact Tracing Work? Quasi-Experimental Evidence from an Excel Error in England*

Thiemo Fetzer† Thomas Graeber‡

November 24, 2020

Abstract

Contact tracing has been a central pillar of the public health response to the COVID-19 pandemic. Yet, contact tracing measures face substantive challenges in practice and well-identified evidence about their effectiveness remains scarce. This paper exploits quasi-random variation in COVID-19 contact tracing. Between September 25 and October 2, 2020, a total of 15,841 COVID-19 cases in England (around 15 to 20% of all cases) were not immediately referred to the contact tracing system due to a data processing error. Case information was truncated from an Excel spreadsheet after the row limit had been reached, which was discovered on October 3. There is substantial variation in the degree to which different parts of England areas were exposed – by chance – to delayed referrals of COVID-19 cases to to the contact tracing system. We show that more affected areas subsequently experienced a drastic rise in new COVID-19 infections and deaths alongside an increase in the positivity rate and the number of test performed, as well as a decline in the performance of the contact tracing system. Conservative estimates suggest that the failure of timely contact tracing due to the data glitch is associated with more than 125,000 additional infections and over 1,500 additional COVID-19-related deaths. Our findings provide strong quasi-experimental evidence for the effectiveness of contact tracing.

Keywords: Health, Coronavirus
JEL Classification: I31, Z18

Why debugging matters (cont.)

Technology

Facebook made big mistake in data it provided to researchers, undermining academic work

Company accidentally left out half of all of its U.S. users in providing data to a research consortium

The error resulted from Facebook accidentally excluding data from U.S. users who had no detectable political leanings — a group that amounted to roughly half of all of Facebook's users in the United States. Data from users in other countries was not affected.

"It's data. Of course, there are errors," said Gary King, a Harvard professor who co-chairs Social Science One. "This, of course, was a big error."

King, director of the university's Institute for Quantitative Social Science, said dozens of papers from researchers affiliated with Social Science One had relied on the data since Facebook shared the flawed set in February 2020, but he said the impact could be determined only after Facebook provided corrected data that could be reanalyzed. He said some of the errors may cause little or no problems, but others could be serious.

Social Science One shared the flawed data with at least 110 researchers, King said.

An Italian researcher, Fabio Giglietto, discovered data anomalies last month and brought them to Facebook's attention. The company contacted researchers in recent days with news that they had failed to include roughly half of its U.S. users — a group that likely is less politically polarized than Facebook's overall user base. The New York Times first reported Facebook's error.

`Source` Washington Post

`Source` Solomon Messing / Twitter

A general strategy for debugging

1. Google
2. Reset
3. Debug
4. Deter

Google

According to this analysis, the most common error types in R are:[1]

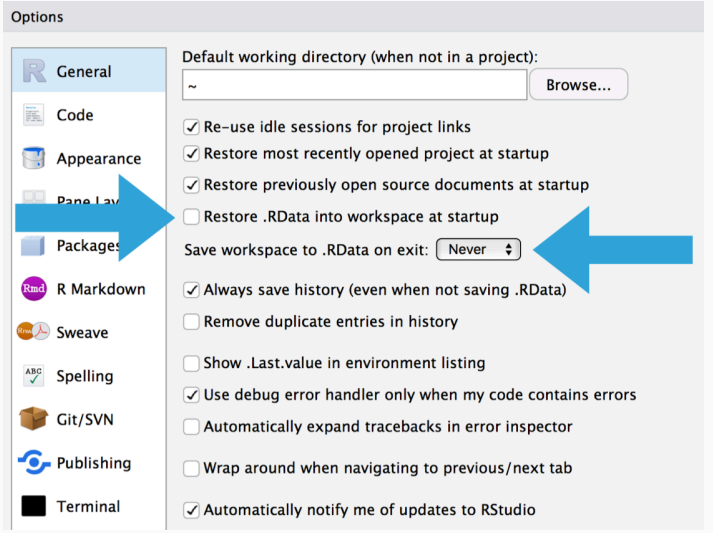
1. `Could not find function` errors, usually caused by typos or not loading a required package.
2. `Error in if` errors, caused by non-logical data or missing values passed to R's `if` conditional statement.
3. `Error in eval` errors, caused by references to objects that don't exist.
4. `Cannot open` errors, caused by attempts to read a file that doesn't exist or can't be accessed.
5. `no applicable method` errors, caused by using an object-oriented function on a data type it doesn't support.
6. `subscript out of bounds` errors, caused by trying to access an element or dimension that doesn't exist
7. Package errors caused by being unable to install, compile or load a package.

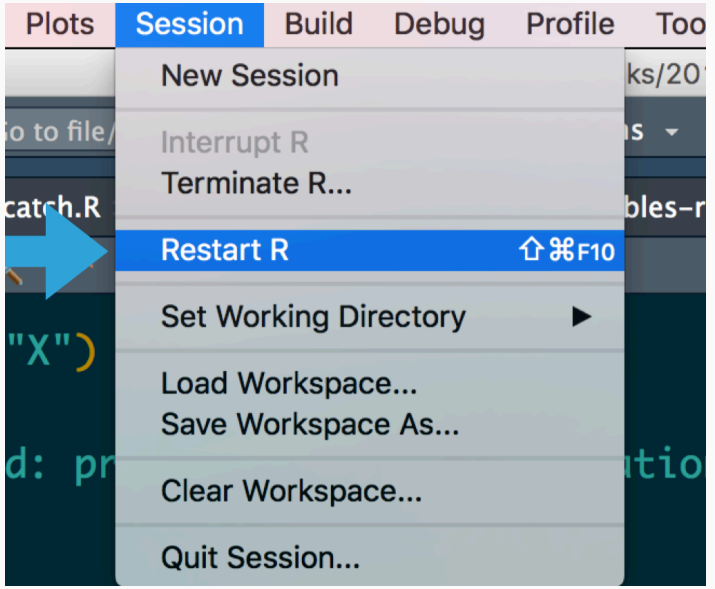
Whenever you see an error message, start by googling it. Improve your chances of a good match by removing any variable names or values that are specific to your problem. Also, look for Stack Overflow posts and list of answers.

[1] Do you get an error message you don't understand? That's good news actually, because the really nasty bugs come without errors.

Reset

- If at first you don't succeed, try exactly the same thing again.
- Have you tried turning it off and on again?
- Do you use `rm(list = ls())`? Don't. Packages remain loaded, options and environment variables set, ... all possible sources of error!
- A fresh start clears the workspace, resets options, environment variables, and the path.
- While we're at it, check out James Wade's advice "How I set up RStudio for Efficient Coding" (YouTube).





Debug

Make the error repeatable.

- Execute the code many times as you consider and reject hypotheses. To make that iteration as quick possible, it's worth some upfront investment to make the problem both easy and fast to reproduce.
- Work with reproducible and minimal examples by removing innocuous code and simplifying data.
- Consider automated testing. Add some nearby tests to ensure that existing good behaviour is preserved.

Track the error down.

- Execute code step by step and inspect intermediate outputs.
- Adopt the scientific method: Generate hypotheses, design experiments to test them, and record your results.

Once found, fix the error and test it.

- Ensure you haven't introduced any new bugs in the process.
- Make sure to carefully record the correct output, and check against the inputs that previously failed.
- Reset and run again to make sure everything still works.

Deter

Defensive programming

- **Pay attention.** Do results make sense? Do they look different from previous results? Why?
- **Know what you're doing**, and what you're expecting.
  - Avoid functions that return different types of output depending on their input, e.g., `[]` and `sapply()`.
  - Be strict about what you accept (e.g., only scalars).
  - Avoid functions that use non-standard evaluation (e.g., `with()`)
- **Fail fast**.
  - As soon as something wrong is discovered, signal an error.
  - Add tests (e.g., with the `testthat` package).
  - Practice good condition/exception handling, e.g., with `try()` and `tryCatch()`.
  - Write error messages for humans.

Transparency

- Collaborate! Pair programming is an established software development technique that increases code robustness. It also works from remote.
- Be transparent! Let others access your code and comment on it.

Debugging R: What you get

```
Error : .onLoad failed in loadNamespace() for 'rJava', details:
call: dyn.load(file, DLLpath = DLLpath, ...)
error: unable to load shared object '/Users/janedoe/Library/R/3.6/library/rJava/libs/rJava.so':
libjvm.so: cannot open shared object file: No such file or directory
Error: loading failed
Execution halted
ERROR: loading failed
* removing '/Users/janedoe/Library/R/3.6/library/rJava/'
Warning in install.packages :
installation of package 'rJava' had non-zero exit status
```

`Credit` Jenny Bryan

Debugging R: What you see

```
Error : blah failed blah blah() blah 'blah', blah:
call: blah.blah(blah, blah = blah, ...)
error: unable to blah blah blah '/blah/blah/blah/blah/blah/blah/blah/blah/blah.so':
blah.so: cannot open blah blah blah: No blah blah blah blah
Error: blah failed
blah blah
ERROR: blah failed
* removing '/blah/blah/blah/blah/blah/blah/blah/'
Warning in blah.blah :
blah of blah 'blah' blah blah-blah blah blah
```

`Credit` Jenny Bryan

Strategies to debug your R code

Sometimes the mistake in your code is hard to diagnose, and googling doesn't help. Here are a couple of strategies to debug your code:

- Use `traceback()` to determine where a given error is occurring.
- Output diagnostic information in code with `print()`, `cat()` or `message()` statements.
- Use `browser()` to open an interactive debugger before the error
- Use `debug()` to automatically open a debugger at the start of a function call.
- Use `trace()` to make temporary code modifications inside a function that you don't have easy access to.

Locating errors with traceback()

Motivation and usage

- When an error occurs with an unidentifiable error message or an error message that you are in principle familiar with but cannot locate its sources, the `traceback()` function comes in handy.
- The `traceback()` function prints the sequence of calls that led to an uncaught error.
- The `traceback()` output reads from bottom to top.
- Note that errors caught via `try()` or `tryCatch()` do not generate a traceback!
- If you're calling code that you `source()`d into R, the traceback will also display the location of the function, in the form `filename.r#linenumber`.

Example

In the call sequence below, the execution of `g()` triggers an error:

```
R> f ← function(x) x + 1
R> g ← function(x) f(x)
R> g("a")
```

```
#> Error in x + 1 : non-numeric argument to binary o
```

Doing the traceback reveals that the function call f(x) is what lead to the error:

```
R> traceback()
```

```
#> 2: f(x) at #1
#> 1: g("a")
```

Interactive debugging with browser()

Motivation and usage

- Sometimes, you need more information than the precise location of an error in a function to fix it.
- The interactive debugger lets you pause the run of a function and interactively explore its state.
- Two options to enter the interactive debugger:
  1. Through RStudio's "Rerun with Debug" tool, shown to the right of an error message.
  2. You can insert a call to `browser()` into the function at the stage where you want to pause, and re-run the function.
- In either case, you'll end up in an interactive environment inside the function where you can run arbitrary R code to explore the current state. You'll know when you're in the interactive debugger because you get a special prompt, `Browse[1]>`.

Example

```
R> h ← function(x) x + 3
R> g ← function(b) {
+   browser()
+   h(b)
+ }
R> g(10)
```

Some useful things to do are:

1. Use `ls()` to determine what objects are available in the current environment.
2. Use `str()`, `print()` etc. to examine the objects.
3. Use `n` to evaluate the next statement.
4. Use `s`: like `n` but also step into function calls.
5. Use `where` to print a stack trace (→ traceback).
6. Use `c` to exit debugger and continue execution.
7. Use `Q` to exit debugger and return to the R prompt.

Debugging other peoples' code

Motivation

- Sometimes the error is outside your code in a package you're using, you might still want to be able to debug.
- Two options:
  1. Get a local version of the package code and debug as if it were your own.
  2. Use functions which which allow you to start a browser in existing functions, including `recover()` and `debug()`.

Debugging other peoples' code (cont.)

Motivation

- `recover()` serves as an alternative error handler which you activate by calling `options(error = recover)`.
- You can then select from a list of current calls to browse.
- `options(error = NULL)` turns off this debugging mode again.
- A simpler alternative is `options(error = browser)`, but this only allows you to browse the call where the error occurred.

Example

- Activate debugging mode; then execute (flawed) function:

```
R> options(error = recover)
R> lm(mpg ~ wt, data = "mtcars")
```

```
Error in model.frame.default(formula = mpg ~ wt, data = "mtcars", drop
 'data' must be a data.frame, environment, or list

Enter a frame number, or 0 to exit

1: lm(mpg ~ wt, data = "mtcars")
2: eval(mf, parent.frame())
3: eval(mf, parent.frame())

Selection:
```

- Deactivate debugging mode:

```
R> options(error = NULL)
```

Debugging other peoples' code (cont.)

Motivation

- `debug()` activates the debugger on any function, including those in packages (see on the right). `undebug()` deactivates the debugger again.
- Some functions in another package are easier to find than others. There are
  - *exported* functions which are available outside of a package and
  - *internal* functions which are only available within a package.
- To find (and debug) exported functions, use the `::` syntax, as in `ggplot2::ggplot`.
- To find un-exported functions, use the `:::` syntax, as in `ggplot2:::check_required_aesthetics`.

Example

- Activate debugging mode for `lm()` function; then execute function:

```
R> debug(stats::lm)
R> lm(mpg ~ weight, data = "mtcars")
```

- Interactive debugging mode for `lm()` is entered; use the common `browser()` functionality to navigate:

```
debugging in: lm(mpg ~ weight, data = mtcars)
debug: {
    ret.x ← x
    ...
Browse[2]>
```

- Deactivate debugging mode:

```
R> undebug(stats::lm)
```

Debug Mode

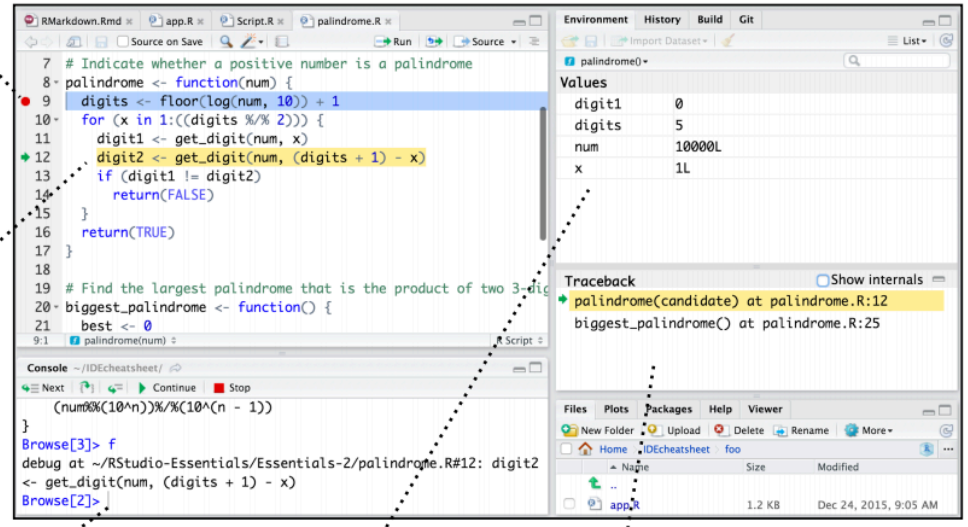
Open with **debug(), browser(),** or a breakpoint. RStudio will open the debugger mode when it encounters a breakpoint while executing code.

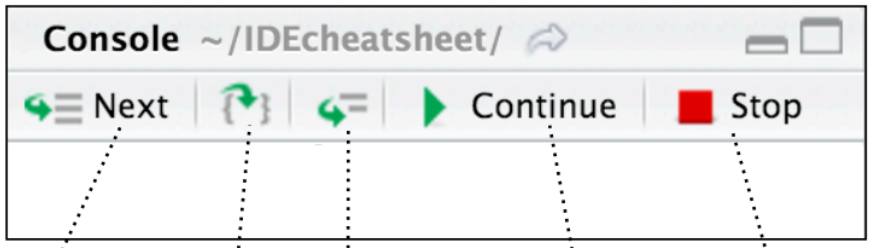
Launch debugger mode from origin of error

Open traceback to examine the functions that R called before the error occurred

Click next to line number to add/remove a breakpoint.

Highlighted line shows where execution has paused





Run commands in environment where execution has paused

Examine variables in executing environment

Select function in traceback to debug

Step through code one line at a time

Step into and out of functions to run

Resume execution

Quit debug mode

More on debugging R

Further reading

- 12-minute video on debugging in R
- Jenny Bryan's talk on debugging at rstudio::conf 2020
- Jenny Bryan and Jim Hester's "What They Forgot to Teach You About R", Chapter 11: Debugging R code
- Jonathan McPherson's Debugging with RStudio

Next steps

Assignment 1 and Quiz 1

Don't forget to submit your solutions for Assignment 1. Also, Quiz 1 is online!

Next lecture

We're going to dig into the world wide web...

Important: The lecture is going to take place on Wednesday, 8-10am, in the Forum! If your regular lab is schedule for that slot, please visit the lab session at the ordinary lecture slot instead (Mon, 10-12h). The lecture will be recorded and made available in case you cannot attend.