REPRODUCIBLE PROGRAMMING

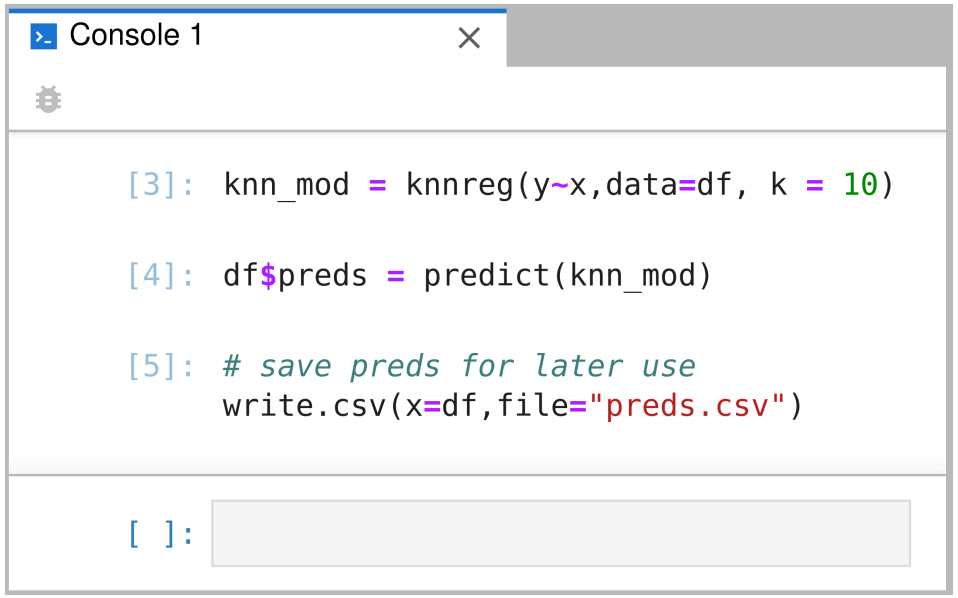
EVERYDAY PRACTICES FOR REPRODUCIBLE PROGRAMMING

- **Theory:** follow a comprehensive set of coding guidelines.
- **Practice:** time-pressure, rapid prototyping, etc. competes with code quality and reproducibility.
- In this module, we cover **five** idioms that can help enhance reproducibility everyday:

1. write it in code, not the console
2. don’t repeat yourself, use functions
3. avoid magic numbers, expose them
4. cache intermediate results
5. seed random numbers

1. AS MUCH AS POSSIBLE, WRITE IT IN CODE.

Code written in the console is wiped upon restart, and not particularly reproducible.

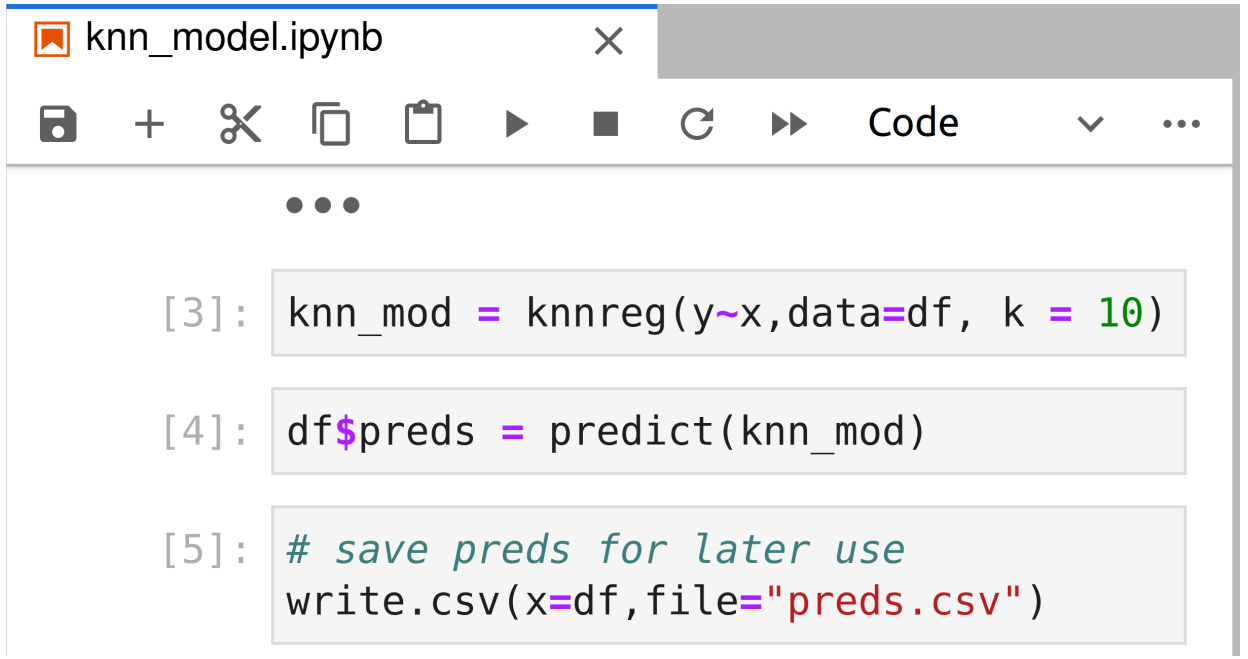


For example, I may save model predictions in “preds.csv” but I don’t **exactly** know how they were produced.

1. AS MUCH AS POSSIBLE, WRITE IT IN CODE.

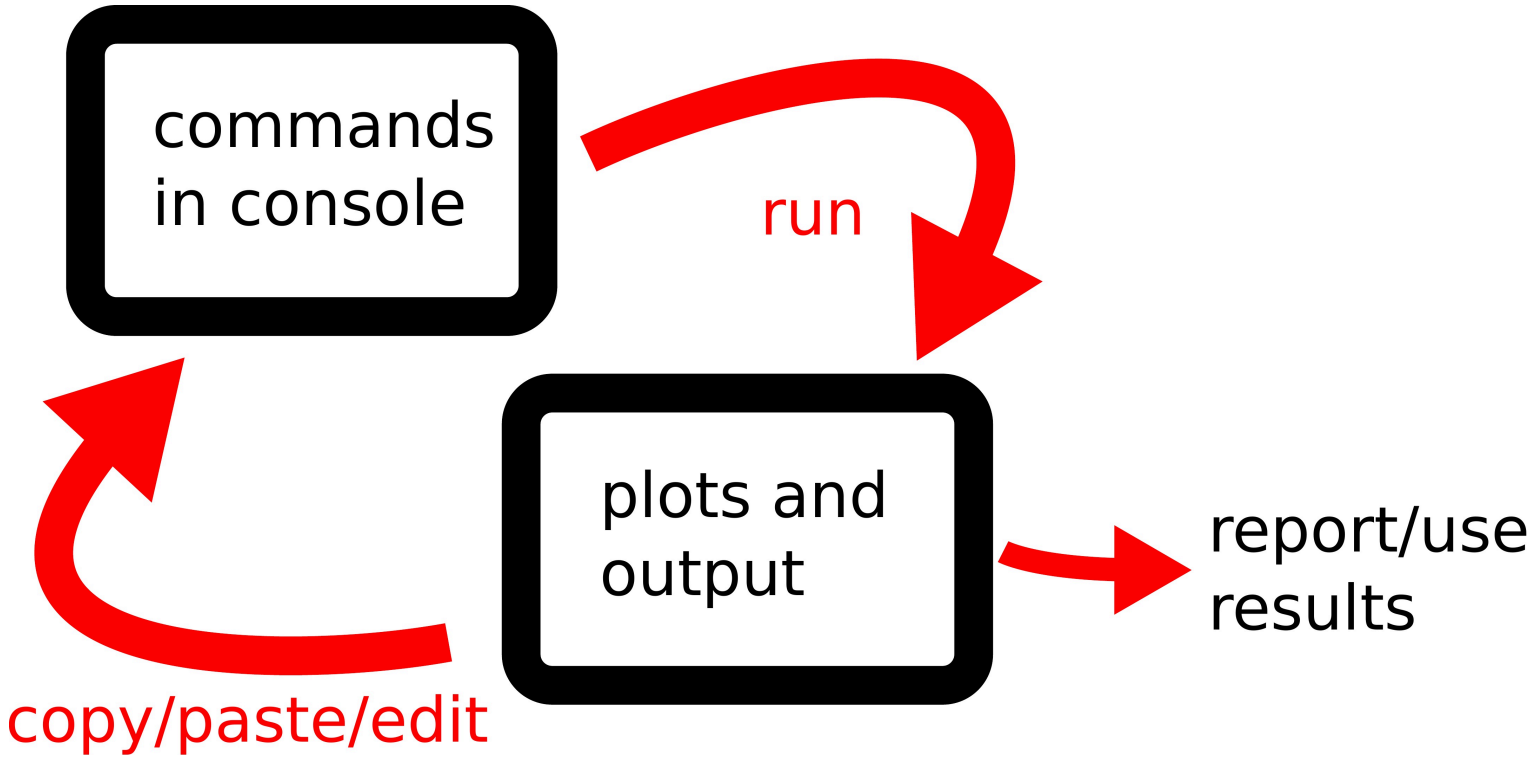
Shared analyses should have a *instruction list*.

Scripts and notebooks are ideal for this:



1. AS MUCH AS POSSIBLE, WRITE IT IN CODE.

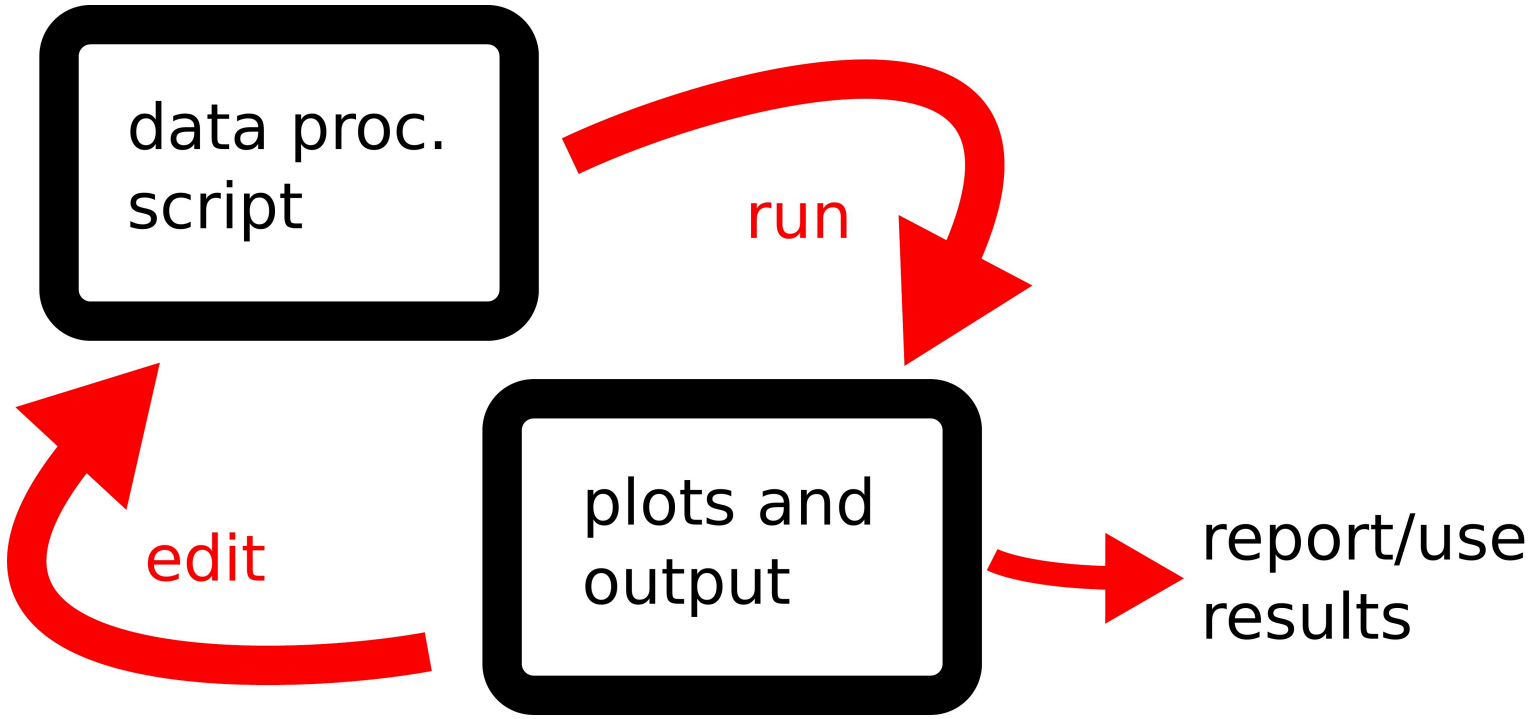
A **less** reproducible process:



After closing down the console, I **don't** have a reproducible record of what I did to get results.

1. AS MUCH AS POSSIBLE, WRITE IT IN CODE.

A **more** reproducible process:



Afterwards, I have a **more reproducible script** that shows exactly what was run to get results.

1. AS MUCH AS POSSIBLE, WRITE IT IN CODE.

This applies to all stages of analysis, including low-level processing of data before more *traditional* statistical analyses.

Example, aligning RNA-seq data:

```
$> samtools sort -@ 8 -o UHR_Rep1.bam UHR_Rep1.sam
$> samtools sort -@ 8 -o UHR_Rep2.bam UHR_Rep2.sam
$> cd $RNA_HOME/alignments/hisat2
$> java -Xmx2g -jar $PICARD MergeSamFiles -OUTPUT
UHR.bam -INPUT UHR_Rep1.bam -INPUT UHR_Rep2.bam -INPUT
UHR_Rep3.bam
$> ls -l *.bam | wc -l
$> hisat2 --very-sensitive --no-spliced-alignment -x
grch38 -U SRR1806626.fastq.gz > SRR1806626.fastq.sam
```

should be wrapped-up in a script:

```
$> bash process_rnaseq.sh
```

1. AS MUCH AS POSSIBLE, WRITE IT IN CODE.

In a `linux` environment `makefile`s can be helpful for unifying and self-documenting this process. For a wonderful tutorial see: Karl Broman's minimal make.

`makefile`s are a way of combining multiple commands specifying the **target** and the **dependencies**.

1. AS MUCH AS POSSIBLE, WRITE IT IN CODE.

Run with the command `make` and pass in a named argument

```
.PHONY: clean all

raw.csv: raw_analysis.R
    Rscript raw_analysis.R

proc.csv: raw.csv proc_analysis.R
    Rscript proc_analysis.R

plot.pdf: proc.csv plotting.R
    Rscript plotting.R

hist.pdf: proc.csv plot_hist.R
    Rscript plot_hist.R

```

1. AS MUCH AS POSSIBLE, WRITE IT IN CODE.

Run with the command `make` and pass in a named argument

```
15  all:
16      make hist.pdf
17      make plot.pdf
18
19  clean:
20      rm raw.csv || true
21      rm proc.csv || true
22      rm plot.pdf || true
23      rm hist.pdf || true
```

2. DON'T REPEAT YOURSELF (DRY), USE FUNCTIONS.

Copying / pasting can create code that is difficult to maintain / understand.

For example, we often see code like this:

```
knn_mod10 = knnreg(y~x,data=df, k = 10)
```

```
knn_mod5 = knnreg(y~x,data=df, k = 5)
```

```
knn_mod1 = knnreg(y~x,data=df, k = 1)
```

2. DON'T REPEAT YOURSELF, USE FUNCTIONS.

Refactoring as a function is a much more scalable solution:

```
fit_knn = function(K){
    knn_mod = knnreg(y~x,data=df, k = K)
    return(knn_mod)
}
```

```
K_seq = c(10,5,1)
```

```
knn_mods = lapply(K_seq,fit_knn)
```

2. DON’T REPEAT YOURSELF, USE FUNCTIONS.

Advantages of **DRY**:

1. makes your code easier to change / maintain (avoiding errors)
2. makes your code easier to understand
3. removes some clutter from code

2. DON’T REPEAT YOURSELF, USE FUNCTIONS.

Can over-do it:

- First time, write it.
- Second time, copy it.
- Third* time, refactor it.
- * May not actually be the *third* time.

2. DON'T REPEAT YOURSELF, USE FUNCTIONS.

"Premature optimization is the root of all evil." – Don Knuth

```
fit_model = function(df, fit_fn, ...){
    mdl = fit_fn(y~x,data=df,...)
    return(mdl)
}
```

```
mod_lm = fit_model(df,lm)
mod_knn = fit_model(df,knnreg,K=5)
```

In this particular case, factoring out the call to `lm` and `knnreg` increased the number of lines of code and made it harder to read.

3. MAGIC NUMBERS SHOULD BE VARIABLES, VARIABLES SHOULD OFTEN BE ARGUMENTS.

An `R` example:

```
run_sim=function(){
    x = sort(runif(200,-4*pi,4*pi)) # simulated data
    eps = rnorm(200,0,.25)
    y = sin(x)+eps

    smoothed = locPolSmootherC(x=x,y=y,xeval=x,deg=0,
                               kernel=gaussK,bw=.5)
    preds = smoothed$beta0
    return(preds)
}
```

```
run_sim()
```

• • •

3. MAGIC NUMBERS SHOULD BE VARIABLES, VARIABLES SHOULD OFTEN BE ARGUMENTS.

Magic numbers: sample size, kernel type, bandwidth, …

```
run_sim=function(){
    x = sort(runif(200,-4*pi,4*pi)) # simulated data
    eps = rnorm(200,0,.25)
    y = sin(x)+eps

    smoothed = locPolSmootherC(x=x,y=y,xeval=x,deg=0,
                               kernel=gaussK,bw=.5)
    preds = smoothed$beta0
    return(preds)
}
```

```
run_sim()
```

• • •

3. MAGIC NUMBERS SHOULD BE VARIABLES, VARIABLES SHOULD OFTEN BE ARGUMENTS.

Better: expose those magic numbers as parameters

```
run_sim=function(N=200,sig=.25,bandw=.5,kern=gaussK){
    x = sort(runif(N,-4*pi,4*pi)) # simulated data
    eps = rnorm(N,0,sig)
    y = sin(x)+eps

    smoothed = locPolSmootherC(x=x,y=y,xeval=x,deg=0,
                               kernel=kern,bw=bandw)
    preds = smoothed$beta0
    return(preds)
}
```

```
run_sim()
```

• • •

```
run_sim(bandw=1,sig=2)
```

3. MAGIC NUMBERS SHOULD BE VARIABLES, VARIABLES SHOULD OFTEN BE ARGUMENTS.

Naming **magic numbers** and other parameter choices

1. makes the code easier to read and more self-documenting
2. enhances reproducibility by
    - flagging these analysis choices,
    - exposing them via an interface for easy (third party) experimentation

4. CACHE INTERMEDIATE RESULTS

Ideally, reproducible analysis takes data from (e.g.)

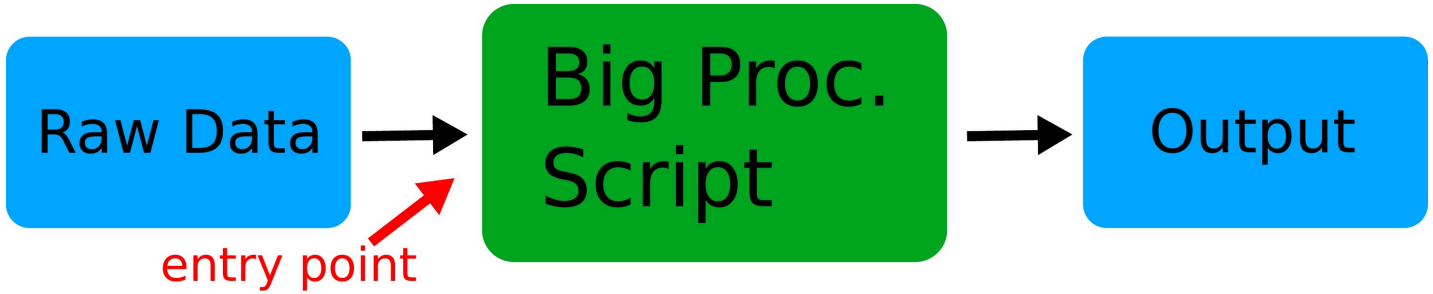
- raw sources, to
- cleaned-up processed data, to
- final results/plots/output

If this is all one long script that processes data without saving any intermediate results along the way, it can be difficult to reproduce the analysis.

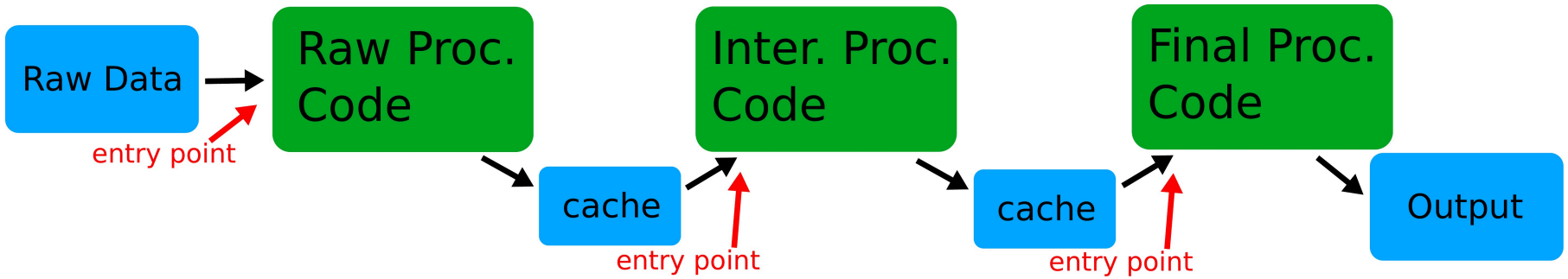
4. CACHE INTERMEDIATE RESULTS

It is good practice to **cache** intermediate results to enhance reproducibility by creating multiple entry-points into the analysis.

Don’t structure analysis like this:



Structure it like this:



4. CACHE INTERMEDIATE RESULTS

A very simplified version in `R` uses the idioms `saveRDS(object,file)` and `object = readRDS(file)`:

```
raw_data = read.csv('raw_data.csv')
# ... do some analysis to produce `basic_data`
```

```
saveRDS(object=basic_data,file='basic_data_cache.rds')
```

```
basic_data = readRDS(file='basic_data_cache.rds')
# .. do some analysis to produce `intermed_data`
```

```
saveRDS(object=intermed_data,file='intermed_data_cache.rds')
```

```
intermed_data = readRDS(file='intermed_data_cache.rds')
#...
```

4. CACHE INTERMEDIATE RESULTS

A useful idiom if I have a single function's results I want to cache:

```
read_or_run = function(cache_file,func){
    if(!file.exists(cache_file)){
        cat("Running func...");flush.console()
        obj = func()
        saveRDS(object=obj,file=cache_file)
    } else {
        cat("Reading from cache...");flush.console()
        obj = readRDS(file=cache_file)
    }
    return(obj)
}
```

4. CACHE INTERMEDIATE RESULTS

An example of this idiom:

```
proc_df = read_or_run('proc_cache.rds',proc_data)
```

```
Running func...
```

```
proc_df = read_or_run('proc_cache.rds',proc_data)
```

```
Reading from cache...
```

5. SEED RANDOM NUMBERS

For statistical analyses, our results often depend on **randomness**. To make randomness **reproducible** we need to identically set the pseudo-random number generators (PRNG) state each time.

In `R` we can do this with `set.seed(number)` or `set.seed(NULL)`

Example: a simple MC estimate of the mean of a U(0,1)

```
x = runif(10)
mean(x^2)
```

0.504305202849137

```
x = runif(10)
mean(x^2)
```

0.432056724361968

5. SEED RANDOM NUMBERS

For statistical analyses, our results often depend on **randomness**. To make randomness **reproducible** we need to identically set the pseudo-random number generators (PRNG) state each time.

In `R` we can do this with `set.seed(number)` or `set.seed(NULL)`

Example: a simple MC estimate of the mean of a U(0,1)

```
set.seed(887561)
x = runif(10)
mean(x^2)
```

0.261476556288769

```
set.seed(887561)
x = runif(10)
mean(x^2)
```

0.261476556288769

5. SEED RANDOM NUMBERS

In `R` the PRNG state is saved in `.Random.seed`

```
head(.Random.seed)
```

10403 · 30 · -1808316273 · -469104443 · 837650556 · 1438237906

A useful idiom:

```
get_rseed = function(){
    if(!exists(".Random.seed"))
        set.seed(NULL)
    return(.Random.seed)
}
```

```
head(get_rseed())
```

10403 · 624 · -1849901807 · 114181022 · 809053063 · -1333479908

5. SEED RANDOM NUMBERS

We can use this state to cache the PRNG state reproducibly without explicitly setting it

```
rseed = read_or_run("random_seed", get_rseed)
.Random.seed = rseed

x = runif(10)
mean(x^2)
```

```
 Running func...
0.280758108258287
```

```
rseed = read_or_run("random_seed", get_rseed)
.Random.seed = rseed

x = runif(10)
mean(x^2)
```

```
 Reading from cache...
0.280758108258287
```

5. SEED RANDOM NUMBERS

If we want to generate a new seed we just remove the cache

```
system('rm random_seed')
```

```
rseed = read_or_run("random_seed", get_rseed)
.Random.seed = rseed

x = runif(10)
mean(x^2)
```

```
 Running func...
0.376071728383279
```

5. BEWARE: FUNCTIONS

Careful with functions:

```
mc_est = function(){
    set.seed(887561)
    x = runif(10)
    mean(x^2)
}
```

```
N_rep = 5
var(replicate(N_rep,mc_est()))
```

0

```
replicate(N_rep,mc_est())
```

1. 0.261476556288769
2. 0.261476556288769
3. 0.261476556288769
4. 0.261476556288769

5. BEWARE: FUNCTIONS

Set the seed *outside* the function

```
set.seed(887561)

mc_est = function(){
    x = runif(10)
    mean(x^2)
}

N_rep = 5
var(replicate(N_rep,mc_est()))
```

0.0177441652674178

5. BEWARE: PARALLELIZATION

As instructed, we set the seed outside the function call:

```
library(parallel)
cl <- makeCluster(5)
```

```
mc_est = function(i){
    x = runif(10)
    mean(x^2)
}
```

```
set.seed(887561)
out = clusterApply(cl, 1:5, mc_est)
out
```

1. 0.437248451074948
2. 0.394221360173399
3. 0.177918863126196
4. 0.311512514403605
5. 0.246496561791708

5. BEWARE: PARALLELIZATION

Re-running it, it's not the same.

```
set.seed(887561)
clusterApply(cl, 1:5, mc_est)
```

1. 0.394305944271141
2. 0.454497342889293
3. 0.398544956729978
4. 0.582426639255505
5. 0.41182331119249

Each parallel instance starts its own session, and thus resets the PRNG.

5. BEWARE: PARALLELIZATION

An ok solution:

```
set.seed(887561)
seeds = sample.int(100000000,5)
seeds
```

1. 33786065
2. 90525536
3. 40106588
4. 15296196
5. 84095680

```
mc_est = function(i){
    set.seed(seeds[i])
    x = runif(10)
    mean(x^2)
}

cl <- makeCluster(5)
clusterExport(cl=cl, varlist="seeds", envir=environment())

clusterApply(cl, 1:5, mc_est)
```

5. BEWARE: PARALLELIZATION

The built-in solution:

```
cl <- makeCluster(5)
clusterSetRNGStream(cl, iseed = 887561)
mc_est = function(i){
    x = runif(10)
    mean(x^2)
}

clusterApply(cl, 1:5, mc_est)
```

1. 0.390381338040286
2. 0.353646776895175
3. 0.338523872182183
4. 0.261878172684988
5. 0.610991600994895

5. BEWARE: PARALLELIZATION

This will fail if I change parallelization parameters e.g. the number of workers.

```
cl <- makeCluster(2)
clusterSetRNGStream(cl, iseed = 887561)
mc_est = function(i){
    x = runif(10)
    mean(x^2)
}

clusterApply(cl, 1:5, mc_est)
```

1. 0.390381338040286
2. 0.353646776895175
3. 0.327037352615337
4. 0.30747298222943
5. 0.218492073469147

5. BEWARE: PARALLELIZATION

A very reproducible way is to use the `future.apply` package:

```
mc_est = function(i){
    x = runif(10)
    mean(x^2)
}
```

```
library('future.apply')
plan(multisession,workers=5)
future_lapply(1:5,FUN=mc_est,future.seed=887561)
```

```
Loading required package: future
```

1. 0.392439730744121
2. 0.288707857966789
3. 0.442395305412029
4. 0.568664612085185
5. 0.408235991591878

This will still work even if we change parallelization parameters.

BONUS: LINTING AND STYLING

linting is checking your code's adherence to a stylistic guidelines

- depending on the language, there are packages that will do this automatically for you
- e.g. `lintr` in `R`
- there are also packages that will automatically find and **fix** these issues for you
- e.g. `styler` in `R`

BONUS: LINTING AND STYLING

This is ugly, let's fix it.

```
# ---
# jupyter:
#   kernelspec:
#     display_name: R
#     language: R
#     name: ir
# ---

f = function(x )
{
    x = x+1
    return(x+1)
}

```

BONUS: LINTING AND STYLING

We can use the `styler` package in `R` to automatically format.

[1]:

```
library('styler')
```

[2]:

```
style_file('test.R')
```

```
Styling  1  files:
 test.R ℹ
──────────────────────────────────────────────────
Status  Count   Legend
✔       0       File unchanged.
ℹ       1       File changed.
✖       0       Styling threw an error.
──────────────────────────────────────────────────
Please review the changes carefully!
```

BONUS: LINTING AND STYLING

Now its pretty.

```r
# ---
# jupyter:
#   kernelspec:
#     display_name: R
#     language: R
#     name: ir
# ---

f <- function(x) {
  x <- x + 1
  return(x + 1)
}

```

BONUS: LINTING AND STYLING

Notice that `styler` dealt with the `yaml` header nicely.

- However it will not be able to directly style `.ipynb` files.
- Solution: `.ipynb` + `jupytext` → `.R`
- `styler` + `.R` → pretty `.R`
- then `jupytext` will automatically propagate back to `.ipynb`

EVERY-SO-OFTEN PRACTICES:

Every-so-often practices:

1. cleaning up your pipeline. Go back periodically, and do things like:
    - delete those commented out lines
    - refactor copy-and-pasted code chunks,
    - rename your poorly named variables,
    - break apart code into better logically structured chunks/scripts
2. testing your pipeline (from soup to nuts)
    - delete all your cached intermediate results
    - clear your notebook outputs
    - remove plots/data produced
    - re-run your whole analysis (ideally via a `makefile`)
3. code review
    - have someone else look at your code
4. avoid proprietary software

DISCUSSION

- How often do you go back and clean-up code?
- What practices do you find most helpful for creating good, reproducible code?
- What do you find gets in the way of applying good practices?