

# **REPRODUCIBLE PROGRAMMING**

# MODULE 3: EVERYDAY PRACTICES FOR REPRODUCIBLE PROGRAMMING

- **Theory:** follow a comprehensive set of coding guidelines.

# MODULE 3: 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.

# MODULE 3: 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:

# MODULE 3: 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

# MODULE 3: 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

# MODULE 3: 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

# MODULE 3: EVERYDAY PRACTICES FOR REPRODUCIBLE PROGRAMMING

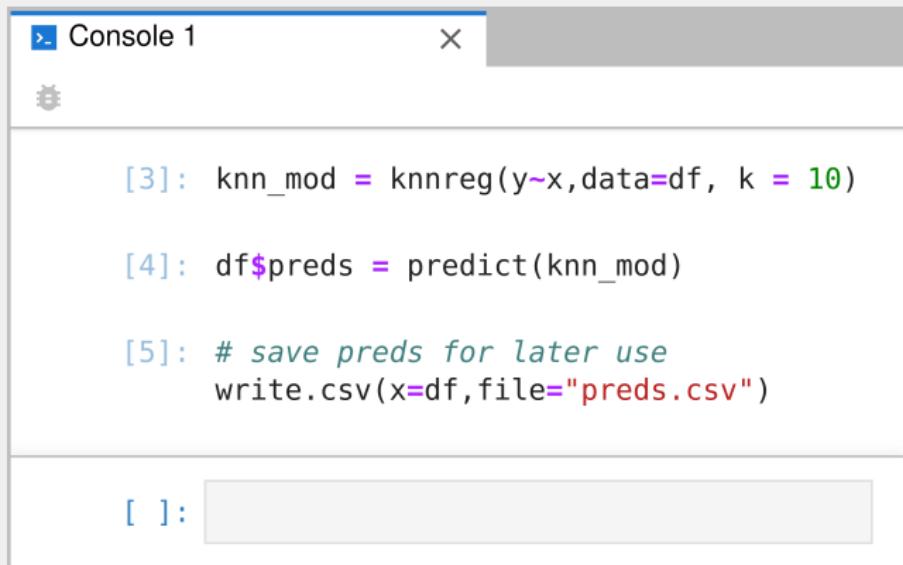
- **Theory:** follow a comprehensive set of coding guidelines.
- **Practice:** time-pressure, rapid prototyping, etc. competes with code quality and reproducibility.
- In this module, we cover **five** idioms that can help enhance reproducibility everyday:
  1. write it in code, not the console
  2. don't repeat yourself, use functions
  3. avoid magic numbers, expose them
  4. cache intermediate results

# MODULE 3: 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.



The screenshot shows the RStudio interface with a "Console 1" tab open. The console window contains the following R code:

```
[3]: knn_mod = knnreg(y~x,data=df, k = 10)

[4]: df$preds = predict(knn_mod)

[5]: # save preds for later use
      write.csv(x=df,file="preds.csv")
```

The code performs the following steps:

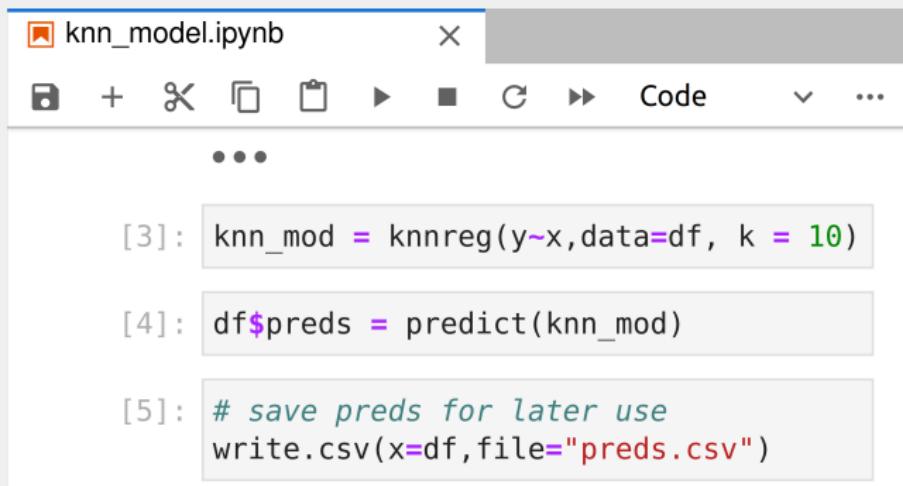
- Creates a KNN regression model (`knn_mod`) using the `knnreg` function from the `kknn` package, with `y` as the dependent variable and `x` as the independent variables, using data from `df` and setting `k` to 10.
- Saves the predicted values from the model into the `df` data frame, adding a new column named `preds`.
- Saves the entire `df` data frame to a CSV file named `"preds.csv"`.

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.

Following Ellis and Leek (2018), shared analyses should have a “instruction list” documenting how the analysis was performed.

Generally, we think that documenting analysis in a notebook/script helps enhance reproducibility. e.g. the previous code could be put in a notebook:



The screenshot shows a Jupyter Notebook interface with the title "knn\_model.ipynb". The notebook has three code cells:

```
[3]: knn_mod = knnreg(y~x,data=df, k = 10)

[4]: df$preds = predict(knn_mod)

[5]: # save preds for later use
      write.csv(x=df,file="preds.csv")
```

# 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.

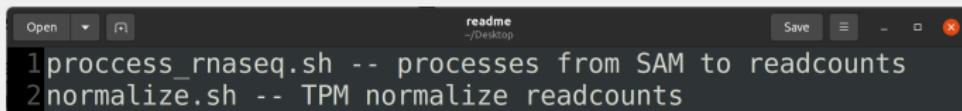
Example, aligning RNA-seq data:

```
$> 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
```

Write scripts:

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

and readmes:

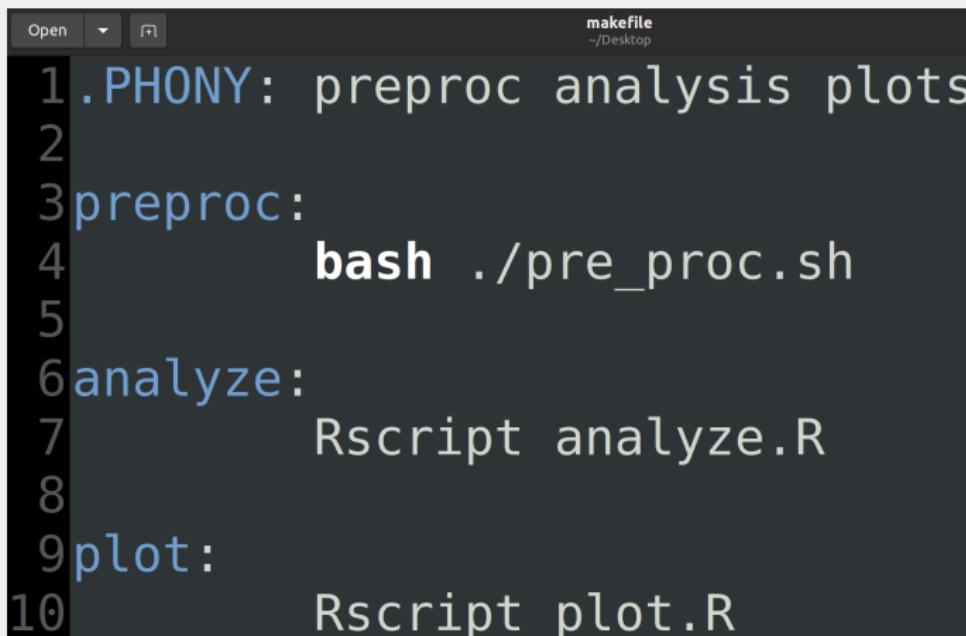


A screenshot of a code editor window titled "readme". The window shows two lines of text:  
1 process\_rnaseq.sh -- processes from SAM to readcounts  
2 normalize.sh -- TPM normalize readcounts

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

In a **linux** environment makefiles can be helpful for unifying and self-documenting this process.

This is a file called makefile that is run with the command make



The image shows a terminal window with a dark background and light-colored text. At the top left is a menu bar with 'Open' and a dropdown arrow. To the right of the menu is the file name 'makefile' and the path '~/Desktop'. The main area of the terminal contains the following text:

```
1 .PHONY: preproc analysis plots
2
3 preproc:
4     bash ./pre_proc.sh
5
6 analyze:
7     Rscript analyze.R
8
9 plot:
10    Rscript plot.R
```

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

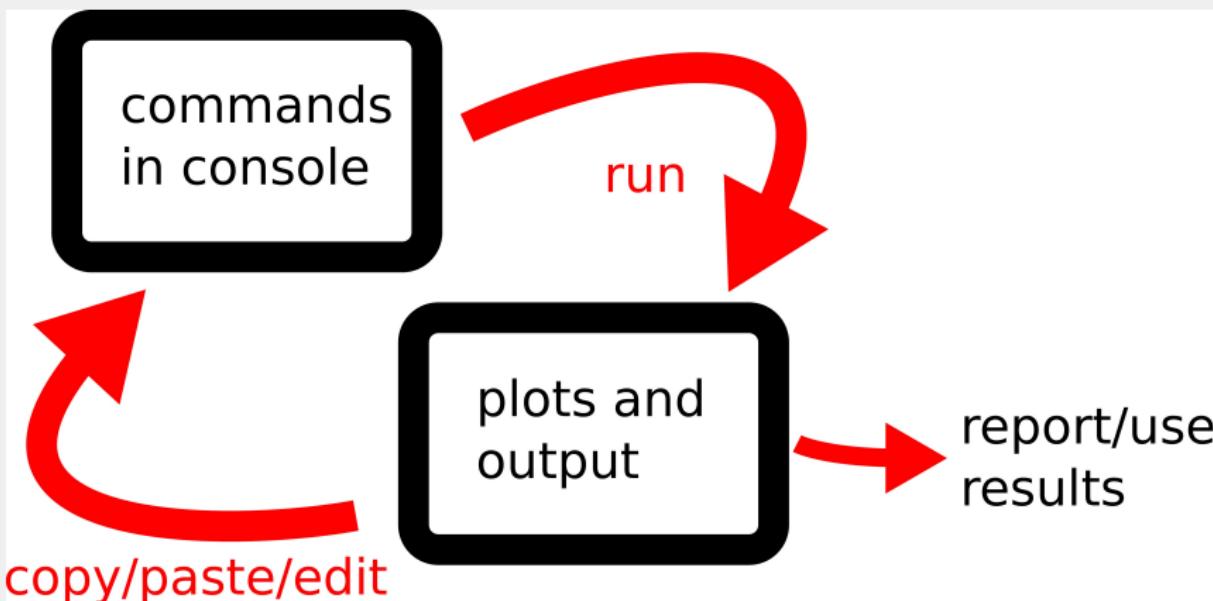
Run with the command make and pass in a named argument

```
#>make preproc
bash ./pre_proc.sh
Pre-processing raw data...
Pre-processing done: output in data.csv
#>
#>make analyze
Rscript analyze.R
Analyzing data...
Analysis done: results for plotting in plot_data.csv
#>
#>make plot
Rscript plot.R
Saving plots in plot.pdf
null device
      1
```

This helps self-document and collect all processing steps across languages/programs etc.

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

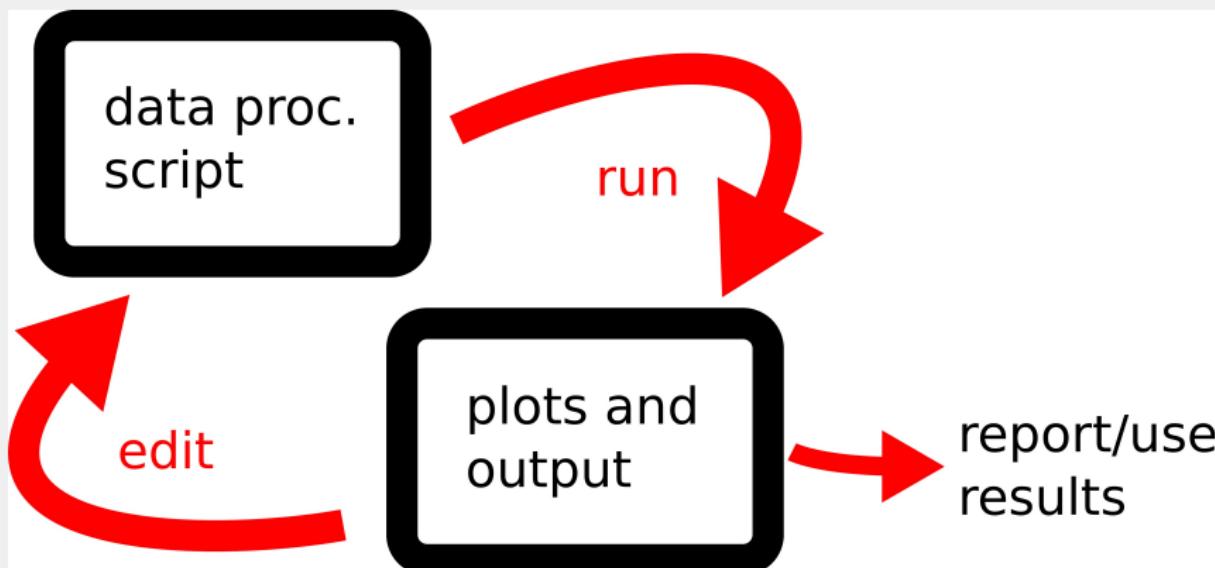
A **less** reproducible process:



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.

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

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

For example, 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.

**DRY** advantages:

1. makes your code easier to change / maintain (avoiding errors)

## 2. DON'T REPEAT YOURSELF, USE FUNCTIONS.

**DRY** advantages:

1. makes your code easier to change / maintain (avoiding errors)
2. makes your code easier to understand

## 2. DON'T REPEAT YOURSELF, USE FUNCTIONS.

**DRY** advantages:

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.

**DRY** advantages:

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

■ **be careful**, one can over-do it:

## 2. DON'T REPEAT YOURSELF, USE FUNCTIONS.

**DRY** advantages:

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

■ **be careful**, one can over-do it:

■ General rule:

## 2. DON'T REPEAT YOURSELF, USE FUNCTIONS.

**DRY** advantages:

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

■ **be careful**, one can over-do it:

■ General rule:

1. First time, write it.

## 2. DON'T REPEAT YOURSELF, USE FUNCTIONS.

**DRY** advantages:

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

■ **be careful**, one can over-do it:

■ General rule:

1. First time, write it.
2. Second time, copy it.

## 2. DON'T REPEAT YOURSELF, USE FUNCTIONS.

**DRY** advantages:

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

■ **be careful**, one can over-do it:

■ General rule:

1. First time, write it.
2. Second time, copy it.
3. Third\* time, refactor it.

## 2. DON'T REPEAT YOURSELF, USE FUNCTIONS.

**DRY** advantages:

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

■ **be careful**, one can over-do it:

■ General rule:

1. First time, write it.
2. Second time, copy it.
3. 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 = locPolSmotherC(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 = locPolSmoothenC(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 = locPolSmotherC(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

### 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

### 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,

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

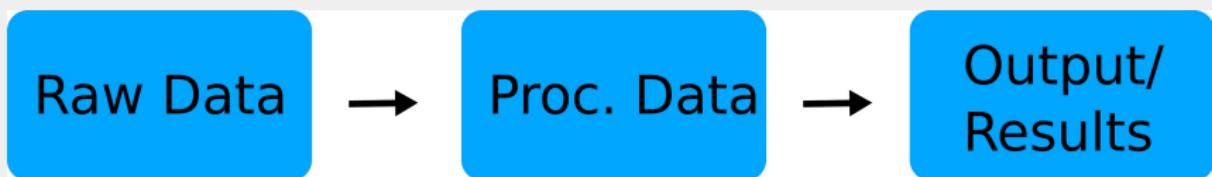
Naming **magic numbers** and other parameter choices

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.

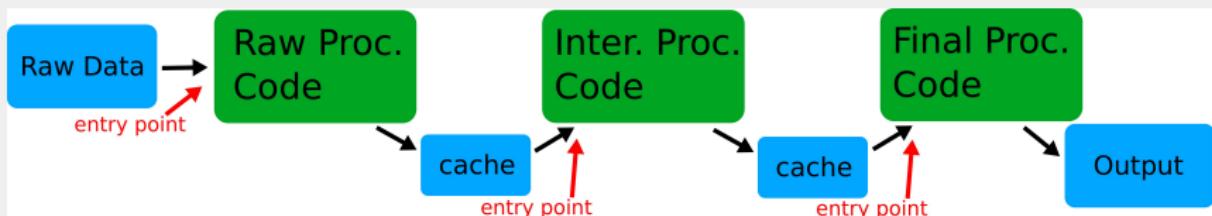
## 4. CACHE INTERMEDIATE RESULTS

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...

## 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
```

## EVERY-SO-OFTEN PRACTICES:

### **Every-so-often** practices:

1. cleaning up your pipeline. Go back periodically, and do things like:

## 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

## 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,

## 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,

## 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

## 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)

## 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

## 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

## 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

## 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)

# 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

# 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

# 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

## EXERCISES: MAKING CODE MORE REPRODUCIBLE.

1. Open up the messy code: either messy.R, messy.ipynb, or messy.qmd
2. Refactor this code so that it is more *proactively reproducible*. Remember the five idioms and try to incorporate them into your solution:
  - write it in code, not the console
  - don't repeat yourself, use functions
  - avoid magic numbers, expose them
  - cache intermediate results
  - seed random numbers

## 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?