

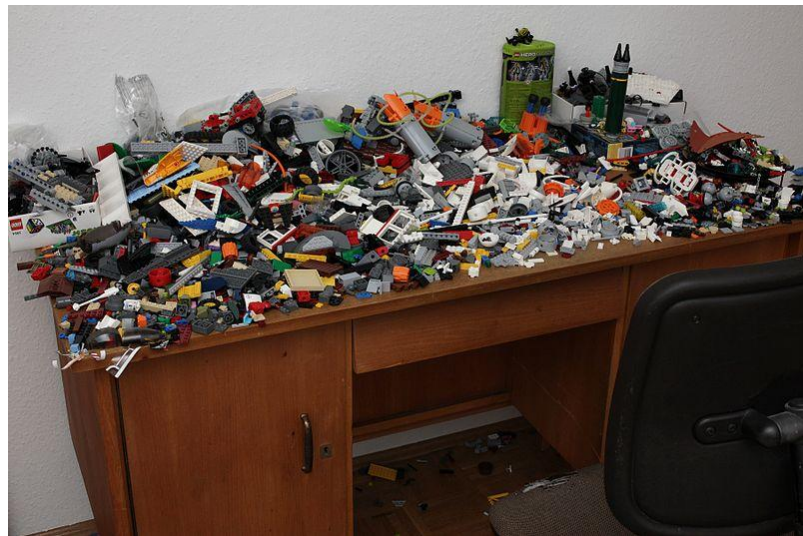


Organizing code in scripts and notebooks

Childhood Cancer Data Lab

Scripts and code get messy

- You don't want to see my desk
 - Luckily, only I have to use it
 - But also, I waste a lot of time looking for things on my desk
- I try to keep my code more organized
 - But sometimes it starts off looking like my desk
 - Patterns and rules help!



Pascal from Heidelberg, Germany, CC BY 2.0, via Wikimedia Commons



Read style guides and use them (with judgement)

- Which style doesn't really matter, but find some agreement with collaborators
 - R tidyverse: <https://style.tidyverse.org/>
 - Google style guides (R, Python, others): <https://google.github.io/styleguide/>
- Don't get too hung up
 - Some style guides are really pedantic. Find your level of comfort
 - Packages are a mix of styles... use what fits

Style points to think about

- Variable and function name styles
 - `my_variable` vs. `myVariable`
 - `do_task()` vs. `task_doer()`
- Indentation and spacing
 - `counter=1` vs. `counter = 1`
 - tabs vs. spaces for indentation (very controversial!)
 - line length (try not to let lines get too long)
- Commenting style and format

Have we mentioned comments?

- Comments are for collaborators, and *Future You*
 - You *will* forget what you were thinking when you wrote your code
 - Your collaborators never knew what you were thinking at all
- Use comments for organization and documentation
 - Try to explain **why** you are doing what you are doing
 - Some “what” is okay, but remember that you can usually look up the functions you used
 - On occasion, people will tell you that code is or should be self-documenting
 - They are wrong
-  Be aware of comments when updating code. You may need to update the comments too! 

Have we mentioned comments?

- Comments are for collaborators, and *Future You*
 - You *will* forget what you were thinking when you wrote your code
 - Your collaborators never knew what you were thinking at all
 - Use comments for organization and documentation
 - Try to explain *why* you are doing what you are doing
 - Some “what” info is helpful to remember that you can usually look up the functions you used
 - On occasion, you may tell you that code is or should be self-documenting
 - Comments are wrong
- Be aware of comments when updating code. You may need to update the comments too! ⚠

COMMENT YOUR CODE!!!

Have we mentioned comments?

- Comments are important for code documentation
 - You will forget why you did something when you write code
 - Your collaborators will need to know what you are doing
- Use comments for organization and documentation
 - Try to explain what you are doing in your code
 - Say "what" instead of "how" that you usually use to write the functions you used
 - Comments should tell you what code should be doing

Be aware of comments when updating code. You may need to update the comments too! ⚠

Let's look at some examples

These examples are in R, but the concepts we talk about will be common across languages

In your `rrp-workshop-examples` repository, open:

`analyses/mutation_counts/01_count-gene-mutations.R`



Header comments

- Start your script or notebook with a description of its purpose
- What kinds of data does it expect?
- What output does it produce?
- What options are available?
- Example(s) of how to run the script

```
#!/usr/bin/env Rscript
```

```
# Count the number of samples mutated for each gene in a MAF file.
```

```
# This script reads in a MAF file and writes out a table of the genes with  
# mutations. The table includes the number of samples that have at least one  
# mutation in the gene, as well as the total number of mutations detected across  
# all samples.
```

```
# Option descriptions:
```

```
#
```

```
# --maf : File path to MAF file to be analyzed. Can be .gz compressed.
```

```
# --outfile : The path of the output file to create
```

```
# --vaf: Minimum variant allele fraction of mutations to include.
```

```
# --min_depth: Minimum sequencing depth to call mutations.
```

```
# --include_syn: Flag to include synonymous mutations in counts.
```

```
# Example invocation:
```

```
#
```

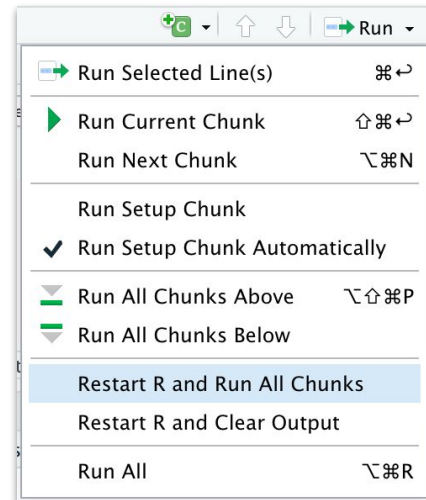
```
# Rscript 01_count-gene-mutations.R \
```

```
#   --maf mutations.maf.tsv.gz \
```

```
#   --outfile gene_counts.tsv
```

Even more comments! Computational Notebooks

- Rmarkdown, Jupyter
- Mix code, text and results in a single document
 - Record your thinking with styled text (markdown), headers for different sections, etc.
 - Plots and results appear right next to the code used to generate them
 - Publish results as html, pdf, or word documents
- Beware dragons:
 - Running code out of order is possible
 - Lingering environment variables can affect results
- Best practice: run everything from top to bottom in a clean environment



An example notebook

In your `rrp-workshop-examples` repository, open:

`analyses/mutation_counts/02_mutation-count-plots.Rmd`

To see the rendered version of this notebook, open:

`analyses/mutation_counts/02_mutation-count-plots.nb.html`

(in your browser)



Computational notebooks vs. scripts

- Notebooks are great for interactive work and reports
- Scripts can be better for repeated tasks with variable inputs and outputs
 - Scripts are often easier to integrate into a workflow
- A common pattern:
 - Start with a notebook
 - As the analysis matures or becomes repetitive, pull out code to a script
 - Generalize the script as needed
 - Keep a notebook with an analysis summary



A general plan for script and notebook content

(not exhaustive, not compulsory, somewhat controversial)

- A descriptive header comment first
 - describe the purpose of the script or notebook
- Setup code next
 - Packages and imported code
 - Inputs, outputs, and other “constant” parameters
 - Option parsing (scripts)
 - Custom function definitions
- Finally: the body of the code and content
 - Break up sections with comments and headings as appropriate


Packages and parameters

- Putting all `library()` or `import` statements near the beginning of the file makes it easier to see what packages are being used
 - All later code uses the same environment with access to the same functions
- Similarly, defining inputs and outputs early makes it clear what files are required, and what will be produced
- Parameters that won't be changed during the script: cutoffs, string names, etc.
 - You may want to modify these and rerun, so put them all together
 - Implicit in this: use variables for parameters that might get modified later

Functions before main script code

- Functions must be defined before they can be used
 - Generally this means appearing earlier in the script or notebook*
- Defining functions in multiple places within a script can make them hard to track down
- Having all the functions grouped together makes it easier to split them out for reuse

*It is sometimes possible to get a bit tricky: define a function called `main()` first with the core logic, then call it at the end of the script. This is common in some programming languages, but it doesn't work in notebooks.



Command line options

- You've seen how many UNIX commands take options (flags) at the command line
 - Your scripts can do this too!
- Writing code to process options can be tedious, but there are packages to help!
 - R: `optparse`
 - Python: `argparse`, `click`, others...
 - Things you get “for free”
 - parsing of options set at the command line & storing them in variables
 - default values for variables when no option is specified
 - basic error checking (missing values, types, etc.)
 - help documentation with `my_script --help`

optparse setup

```
34 option_list <- list(  
35   make_option(  
36     opt_str = c("--maf", "-m"),  
37     type = "character",  
38     default = NA,  
39     help = "File path of MAF file to be analyzed. Can be .gz compressed."  
40   ),  
41   make_option(  
42     opt_str = c("--outfile", "-o"),  
43     type = "character",  
44     default = "gene_counts.tsv",  
45     help = "File path where output table will be placed."  
46   ),
```

the command line option flags, long and short
(long version will be the variable name)

the required data type

default value

help text; explain what the option is for

Function comments

- Much the same as the main header comments
- What are the arguments for the function?
 - What types and formats are expected?
 - What are any default values?
 - Be kind to yourself and others: check the argument values early and print good error messages
- What does the function return?
- Use language conventions for documentation:
 - Python: “[docstring](#)”: a triple quoted comment right after the function definition
 - R: comments before the function
 - optional: use [roxygen2](#) format
(automates documentation for packages, but a nice standard format for other use)
- *Remember: Even though Present You is writing this function, Future You does not recall it at all.*

```
#' Plot the number of mutations for each gene from a data frame
#'
#' Filters to a cutoff and sorts genes from most to least mutated, then
#' creates a bar plot of the mutation counts, optionally highlighting
#' genes of interest.
#'
#' @param mutations_df A data frame with columns `Hugo_Symbol` and `mutated_samples`
#' @param min_mutated The minimum `mutated_samples` value to include in the plot
#' (default: 3)
#' @param highlight_genes A vector of genes to highlight in the plot (optional)
#' @param highlight_title The title for the highlighted genes legend
#' (default: "Gene of interest")
#'
#' @return A ggplot2 plot object
#'
plot_gene_mutations <- function(
  mutations_df,
  min_mutated = 3,
  highlight_genes = c(),
  highlight_title = "Gene of interest"
){
```

Explicit “namespaces” to avoid conflicts

- Sometimes multiple packages have functions with the same name
- R: Use `package::function()` syntax to avoid ambiguity
 - `dplyr::filter()` vs. `stats::filter()`
 - Also provides in-code documentation:
What package did this strange function come from?
 - Bonus: you don't need a `library()` statement
- Python: `package.function()` syntax is standard
 - avoid `from package import function`
 - use `import pandas as pd` and similar if there is a common standard

```
> library(dplyr)
```

```
Attaching package: 'dplyr'
```

```
The following objects are masked from 'package:stats':
```

```
filter, lag
```

```
The following objects are masked from 'package:base':
```

```
intersect, setdiff, setequal, union
```

Random number seeds

- Some code makes use of random numbers
 - simulations
 - fitting statistical models/machine learning
 - PCA, UMAP, tSNE
- But sometimes we want perfect reproducibility! (debugging, testing)
 - luckily, computers don't use real random numbers
 - random number generators are functions
 - given the same starting point (seed), they will give the same results
- Start your code by setting the seed for replicability
 - R: `set.seed(42)`
 - Python: `random.seed(42)`
 - other packages and tools: look at the docs for the correct option!
 - some packages don't use the language defaults