

Good Code Etiquette Or How to start on your optimisation journey

Dr. Rebecca Lange, Curtin Institute for Computation
Brisbane ResBaz, 10 July 2019

These slides are based on the presentation prepared by Dr. Paul Hancock for the 2019 HWSA. The accompanying github repo can be found at: <https://github.com/CurtinIC/good-code-etiquette>

HELLO

my name is

REBECCA

These slides were developed by Paul Hancock in collaboration with Rebecca Lange and Manodeep Sinha for the 2019 HWSA, and adapted by Rebecca for the 2019 Brisbane ResBaz.

Drag your dot to show if you're ready to move on:



Students, drag the icon!



What programming language do you use most often?

[A] Python 3.X

[B] Python 2.7

[C] R



Students choose an option

How would you feel showing your code to a peer?



 Pear Deck



Students, drag the icon!



Pear Deck Interactive Slide
Do not remove this bar

How would you feel showing your code to a supervisor?



 Pear Deck



Students, drag the icon!



Pear Deck Interactive Slide
Do not remove this bar

How would you feel showing your code to the world?



 Pear Deck



Students, drag the icon!



Pear Deck Interactive Slide
Do not remove this bar



Hadley Wickham ✓

@hadleywickham

Follow



The only way to write good code is to write tons of shitty code first. Feeling shame about bad code stops you from getting to good code

6:11 AM - 17 Apr 2015

P.S.. there is a **LOT** of shitty code on the internet.
Your +1 isn't going to end the world, and it'll get better over time.

Style Guides

“Programs must be written for people to read, and only incidentally for machines to execute.” – Harold Abelson, Structure and Interpretation of Computer Program

A style guide is about **consistency**. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important.

[PEP8 style guide]

Why care?

- provides consistency
- makes code easier to read
- makes code easier to write
- makes it easier to collaborate

[[Beautify your R code](#) by Saskia Freytag]

Python

- ❖ Python Enhancement Proposals
<https://www.python.org/dev/peps/#numerical-index>
- ❖ PEP 8 -- Style Guide for Python Code

R [[Beautify your R code](#) by Saskia Freytag]

- ❖ tidyverse style guide
 - most comprehensive, underscore for naming conventions
- ❖ Advanced R style guide
 - fairly comprehensive, underscore for naming conventions
- ❖ Google style guide
 - first of its kind, CamelCase for naming conventions

Create readable code



Python was designed to be readable

Code-blocks are defined by indentation

Line continuations are not required

Syntax is human readable

```
a="""Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
"""
```

```
lines = a.split('\n') # \n is the newline character  
num_lines = len(lines)
```

```
nwords = 0  
for line in lines:  
    words = line.split()  
    nwords += len(words)
```

```
def do_lots_of_things(option1, # required  
                     option2=0, # not required, has default value  
                     option3=1,  
                     labels=(),  
                     sqlconneccion=None,  
                     retries=3,  
                     verbose=False,  
                     do_print=True):  
  
    pass
```

Indenting issues



Python uses indenting to identify code blocks.

Advantage: No need for brackets, or semi-colons, easily readable

Disadvantage: Python allows **tabs** or **spaces** or some **mix** of the two [worst design choice ever made IMO]

Solution: Be **consistent** with indenting, ideally use 4 spaces (but 2 is ok)

```
def function(firstArgument,  
→ → → → .....secondOne,  
→ → → .....modified);
```

What you **see**
(In this editor anyway)

What you **have**

Many editors will bind the “tab” key to the “tab-ify” function, eg your Jupyter notebook

'One-liners' are write-only



What does this code do?

How does it do it?

```
print(reduce( (lambda r,x: (r.difference_update(range(x*x,N,2*x)) or r)
               if (x in r) else r),
             range(3, int((N+1)**0.5+1), 2),
             set([1,2] + range(3,N,2))))
```

'One-liners' are write-only



What does this code do?

How does it do it?

```
print(reduce( (lambda r,x: (r.difference_update(range(x*x,N,2*x)) or r)
               if (x in r) else r),
             range(3, int((N+1)**0.5+1), 2),
             set([1,2] + range(3,N,2))))
```

Readability counts



What does this code do, and how?

```
from math import sqrt
N = 100
sieve = list(range(N))
for number in range(2, int(sqrt(N))+1):
    if sieve[number]:
        for multiple in range(number**2, N, number):
            sieve[multiple] = False

for number, prime in enumerate(sieve):
    if prime:
        print(number)
```

Curly braces, {}, define the most important hierarchy of R code. To make this hierarchy easy to see:

- { should be the last character on the line. Related code (e.g., an if clause, a function declaration, a trailing comma, ...) must be on the same line as the opening brace.
- The contents should be indented by two spaces.

• *# Good*

Note `if (y < 0 && debug) {`
`message("y is negative")`
`}`

```
if (y == 0) {  
  if (x > 0) {  
    log(x)  
  } else {  
    message("x is negative or zero")  
  }  
} else {  
  y^x  
}
```

• *# Bad*

`if (y < 0 && debug) {`
`message("Y is negative")`
`}`

```
if (y == 0)  
{  
  if (x > 0) {  
    log(x)  
  } else {  
    message("x is negative or zero")  
  }  
} else { y ^ x }
```

and

Pipes in R



`%>%` should always have a space before it, and should usually be followed by a new line.

If the arguments to a function don't all fit on one line, put each argument on its own line and indent

Good

```
iris %>%  
  group_by(Species) %>%  
  summarize_if(is.numeric, mean) %>%  
  ungroup() %>%  
  gather(measure, value, -Species) %>%  
  arrange(value)
```

Bad

```
iris %>% group_by(Species) %>% summarize_all(mean) %>%  
ungroup %>% gather(measure, value, -Species) %>%  
arrange(value)
```

```
iris %>%  
  group_by(Species) %>%  
  summarise(  
    Sepal.Length = mean(Sepal.Length),  
    Sepal.Width = mean(Sepal.Width),  
    Species = n_distinct(Species)  
  )
```

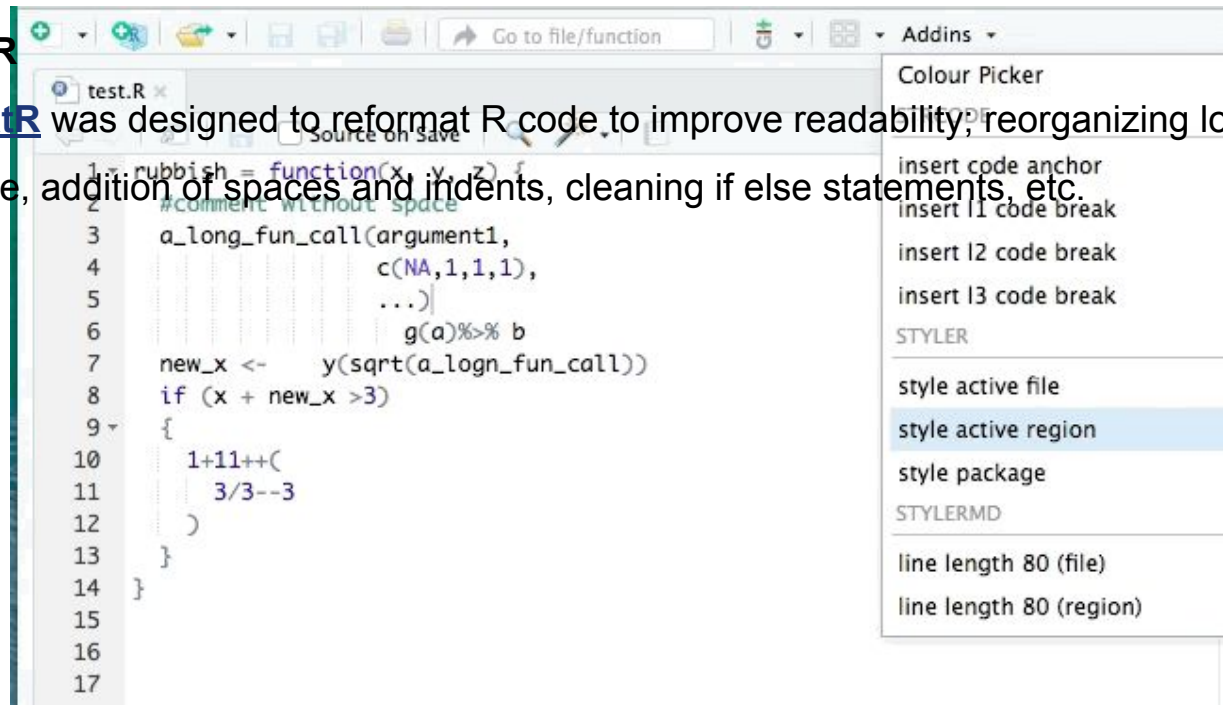

R's little stylists

tidyverse

- [lintr](#) performs automated checks to confirm that you conform to the style guide.
- [styler](#) allows you to interactively restyle selected text, files, or entire projects. It includes an RStudio add-in, the easiest way to re-style existing code.

Advanced R

- [formatR](#) was designed to reformat R code to improve readability, reorganizing long lines of code, addition of spaces and indents, cleaning if else statements, etc.



Interactive Development Environment

And IDE is like a text editor but with lots of extra fancy-ness added on.

In fact you can take your favorite text editor (emacs or vim) and give it an upgrade with plugins that will turn it into more of an IDE.

Syntax Highlighting and Checking

Auto Indentation

Spell Checking (language aware)

Get a 'real' IDE

Includes: debugging tools, integration with version control, refactoring tools, templates for new modules/files and docstrings.

- PyCharm (not just for python)
- RStudio

Others:

- Spyder (part of anaconda install)
- Emacs and Vim have lots of plug-ins and extensions for this
- Many others -> see what people around you are using

What kind of IDE do you usually use?



Full featured.
With Git, Templates, etc.



Some features.
Colours and highlighting.

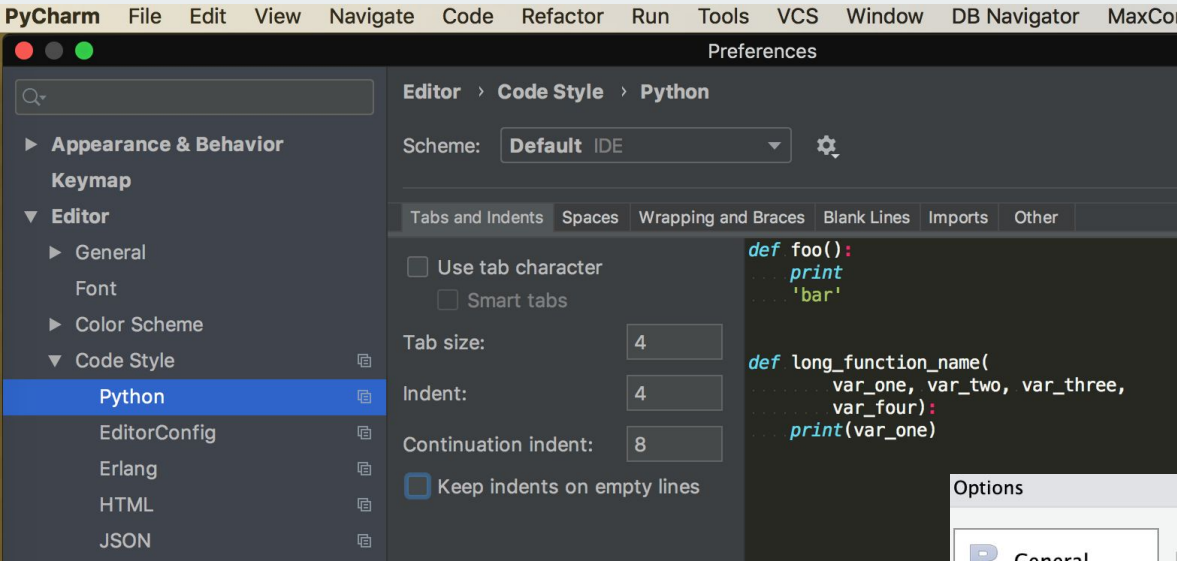


Basic text editor.
No added value.

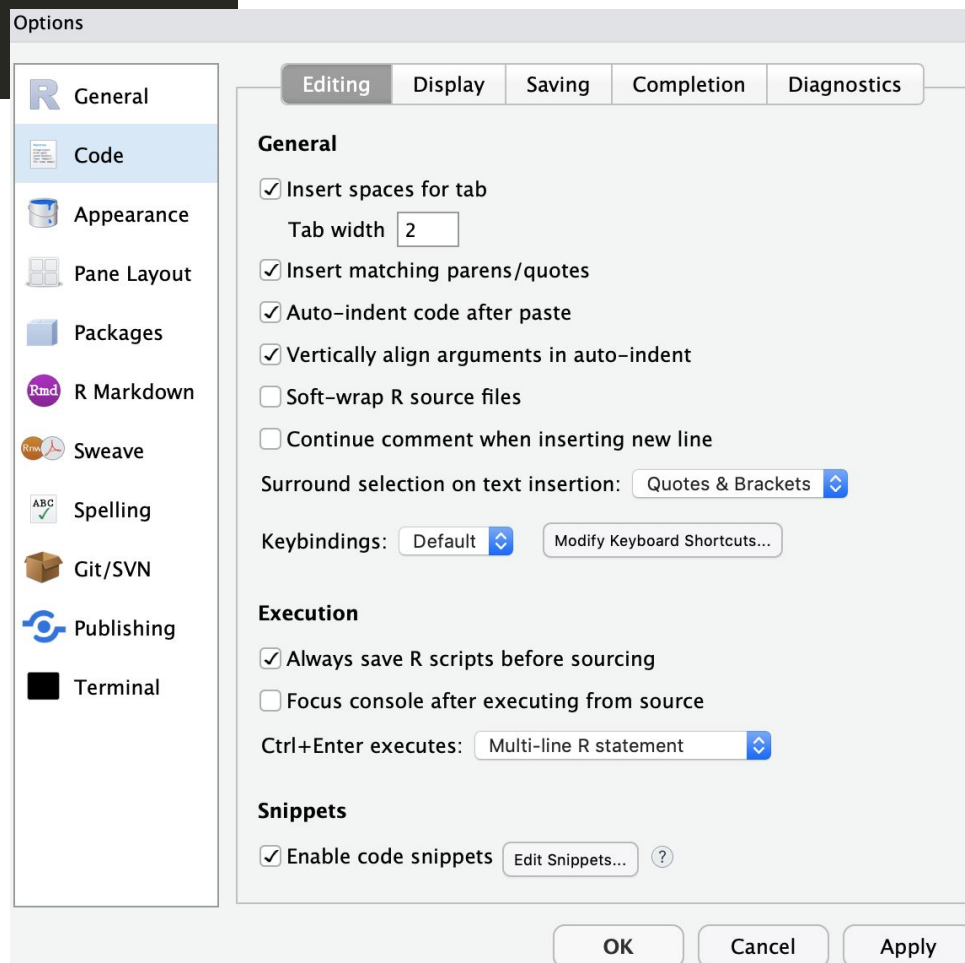


Students, drag the icon!





IDEs can help



Naming conventions



Use words!

Preferably: nouns for classes and variables, verbs for functions, (adjectives for decorators?)

Be verbose but not needlessly so.

`underscores_for_functions`

`CamelCaseForClasses`

`ALL_CAPS_FOR_STATIC_VARIABLES`

Naming conventions



Python doesn't enforce static/private variables or functions.

However, in practice:

`STATIC_VARIABLE`

`_private_function_or_variable` (not part of a public API)

Who likes making cakes?

Me



Not me



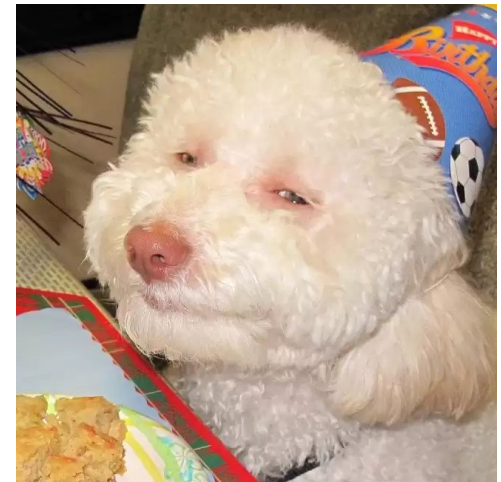
Students choose an option

Pear Deck Interactive Slide
Do not remove this bar

Writing code is not a story that unfolds and entertains people with twists and character developments.

It's a **recipe**. Like for yummy cakes.

1. Ingredients for the shopping list \Rightarrow modules to import
2. Description of techniques \Rightarrow functions
3. Directions \Rightarrow code in main scope



Template for code layout

Python

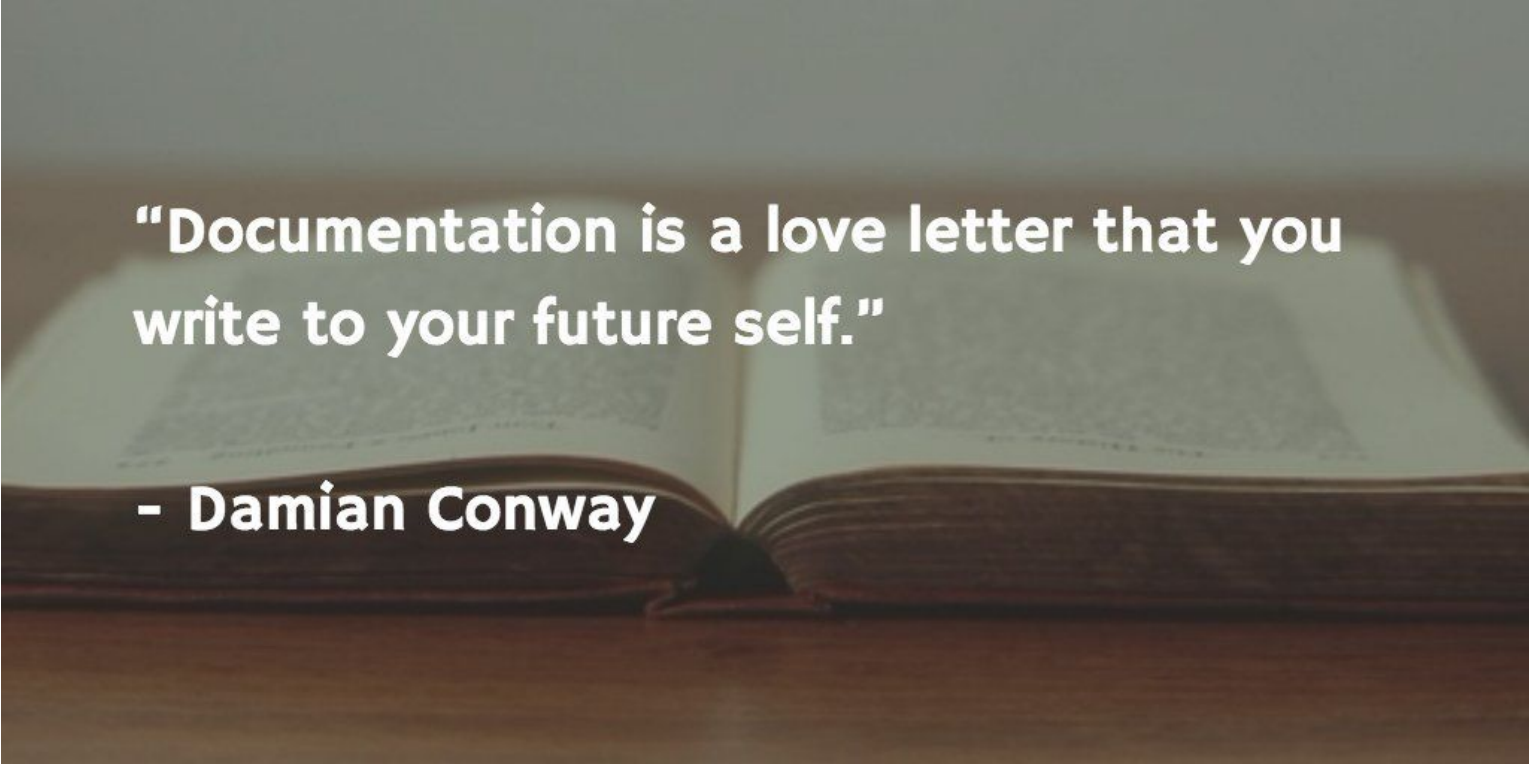
- Preamble
- Imports
- Static and global variables (be judicious)
- Classes
- Functions
- If `__main__`

R (Google style guide)

- Copyright statement comment
- Author comment
- File description comment, including purpose of program, inputs, and outputs
- `source()` and `library()` statements
- Function definitions
- Executed statements, if applicable (e.g., `print`, `plot`)

Structure.ipynb

Document your work

An open book is shown from a top-down perspective, lying flat on a dark wooden surface. The pages are slightly aged and feature faint, illegible text. Overlaid on the center of the open pages is a quote in white, bold, sans-serif font. The background is a soft, out-of-focus light gray.

“Documentation is a love letter that you write to your future self.”

- Damian Conway

Comments are not documentation

Documentation is for people **using** the code (regular folks)

Documentation describes the ingredients and what kind of cakes are made.

Comments are for people **reading** the code (ie developers and future you)

Comments are about the cake making process.

Triple quoted text immediately after a class or function definition is a docstring.

Any format is fine, though some conventions make it easier to work with.

Docstring: numpydoc

Much more flexible and has many more features than other docstring formats.

Used by numpy, scipy, astropy, etc.

```
def numpydoc_func(first, second):  
    """  
    My numpydoc description of a kind  
    of very exhaustive numpydoc format docstring.  
  
    Parameters  
    -----  
    first : array_like  
        the 1st param name `first`  
    second :  
        the 2nd param  
    third : {'value', 'other'}, optional  
        the 3rd param, by default 'value'  
  
    Returns  
    -----  
    string  
        a value in a string  
  
    Raises  
    -----  
    KeyError  
        when a key error  
    OtherError  
        when an other error  
    """  
    return
```

Google R style guide:

- Functions should contain a comments section immediately below the function definition line.
 - one-sentence description of the function;
 - a list of the function's arguments, with a description of each (including the data type); and
 - a description of the return value.

Tidyverse style guide:

- Use **roxygen2** with **markdown** support enabled to keep your documentation close to the code.
 - Use the `#'` for function documentation
- describe your functions in comments next to their definitions and **roxygen2** will process your source code and comments to **produce Rd** files in the `man/` directory

```
#' The length of a string (in characters).  
#'  
#' @param string input character vector  
#' @return numeric vector giving number of characters in each element of the  
#'   character vector. Missing strings have missing length.  
#' @seealso \code{\link{nchar}} which this function wraps  
#' @export  
#' @examples  
#' str_length(letters)  
#' str_length(c("i", "like", "programming", NA))  
str_length <- function(string) {  
  string <- check_string(string)  
  
  nc <- nchar(string, allowNA = TRUE)  
  is.na(nc) <- is.na(string)  
  nc  
}
```

DRY or DIE!

Don't Repeat Yourself (Duplication Is Evil)

Duplicated code means duplicated errors and bugs

Write a function, call it many times

Better still,

- write a **module** in Python and **import** this, or
- save your collection of **functions** in a separate .R script and **source** it

The DRY principle - II (or DRO maybe?)

Don't Repeat Others

- (re-) implementing code often means going through the same growth/development curve of bugs and corner cases
- Common problems have common solutions, use them!
- **'import' / 'library'** your way to success

[**DRY examples.ipynb**](#)

Name a module that you haven't used but would like to know more about.

- or -

Describe a module you wish existed.



Students, write your response!

Pear Deck Interactive Slide
Do not remove this bar

Namespaces

Namespaces help avoid name collisions, and keep track of where functions/variables come from.

Be explicit, especially if you **share** your work or **develop** code with others.

```
from math import *
from numpy import *

print(cos([0,0.5,1])) # prints [ 1.          0.87758256  0.54030231]
print(acos(cos([0,0.5,1]))) # Raises TypeError !?
```

```
import math
import numpy as np # common shorthand

print(np.cos([0, 0.5, 1])) #prints [ 1.          0.87758256  0.54030231]

try:
    # same error as before but now we have some hint why it occurs!
    print(math.acos(np.cos([0, 0.5, 1])))
except TypeError:
    print("It broke, lets try using map()")
    print(map(math.acos, np.cos([0, 0.5, 1])))
```

```
> library(tidyverse)
```

```
— Attaching packages —
```

```
tidyverse 1.2.1 —
```

```
✓ ggplot2 3.1.1    ✓ purrr  0.3.2
✓ tibble  2.1.1    ✓ dplyr  0.8.0.1
✓ tidyr   0.8.3    ✓ stringr 1.4.0
✓ readr   1.3.1    ✓ forcats 0.4.0
```

```
— Conflicts —
```

```
tidyverse_conflicts() —
```

```
✗ dplyr::filter() masks stats::filter()
✗ dplyr::lag()     masks stats::lag()
```


Separate code and data

Having a script that needs to be edited every time it runs is just asking for trouble.

Let's look at [KeepThemSeparated.ipynb](#)

Vectorizing code

Functions that work on single values can be made to work on lists, sets, arrays and just about any iterable object. This is called vectorizing code.

- Use **inbuilt vectorisation**, e.g. numpy functions work on non-scalar inputs
 - Vectorizing is super useful when paired with numpy arrays and related functions as they have been optimised for this purpose.
- The family of apply functions allows you to run your code over input vectors and lists
 - Especially **mapply** might be useful here: <https://rdr.io/r/base/mapply.html>
- We can vectorise our own functions by adding in some checks to see if the input is iterable and then iterating over the input value

Vectorize.ipynb

Test code

`"Finding your bug is a process of confirming the many things that you believe are true – until you find one which is not true."`

`–Norm Matloff`

The only thing that people write less than documentation is test code.

Pro-tip: Both documentation and test code is easier to write if you do it as part of the development process.

1. Write function definition and basic docstring
2. Write function contents
3. Write test to ensure that function does what the docstring claims.
4. Update code and/or docstring until (3) is true.

What to test?

Whatever you currently do to convince yourself that your code works is a test!

Everytime you find a bug or some corner case, write a test that will check it.

Making mistakes doesn't make you a bad person, making the **same mistake** over and over does.

Testing in R:

<http://r-pkgs.had.co.nz/tests.html>

Testing in Python:

<https://docs.python-guide.org/writing/tests/>

Testing.ipynb

Summary and cheat sheet

- Writing good code takes practice.
- Reuse things that work for you.
- Develop a support group you can call on for help.
 - We have weekly meet-up groups like hacky-hour
- Share your code on GitHub or similar, with documentation, so others can benefit from your work.
 - People can help you debug by reporting issues and submitting bug fixes via pull requests
 - Remember sharing your code on github does not mean you can be held accountable for its maintenance.

Oh, and publish your code and cite that of others!!!

[GoodCode_CheatSheet.pdf](#)

Practice what we just learned

PracticeEtiquette.ipynb

Optimisation

Premature optimisation should be avoided.

Knowing **when** to optimise is as important as knowing **how**

Good coding style™ can make optimisation easy

HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE
EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

		HOW OFTEN YOU DO THE TASK					
		50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
HOW MUCH TIME YOU SHAVE OFF	1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
	5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
	30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
	1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
	5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
	30 MINUTES		6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
	1 HOUR		10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
	6 HOURS				2 MONTHS	2 WEEKS	1 DAY
	1 DAY					8 WEEKS	5 DAYS

When and What to Optimise

Optimisation mantras:

1. Do not optimise unless you have to (your time is the most precious)
2. Optimise for readability (imagine very cranky future developers, and reduce their cognitive load)
3. Variety of optimisations
 - a. Readability
 - b. Usability
 - c. Performance
 - d. Memory
 - e. Lines of code (code-golf)
4. Remember, every line of code is a potential source of bugs, and future maintenance headaches. Choose wisely!

Benchmarking.ipynb