You are a coder now

Tips to improve your code

Cyril Pernet, Phd

Intended Learning Outcomes

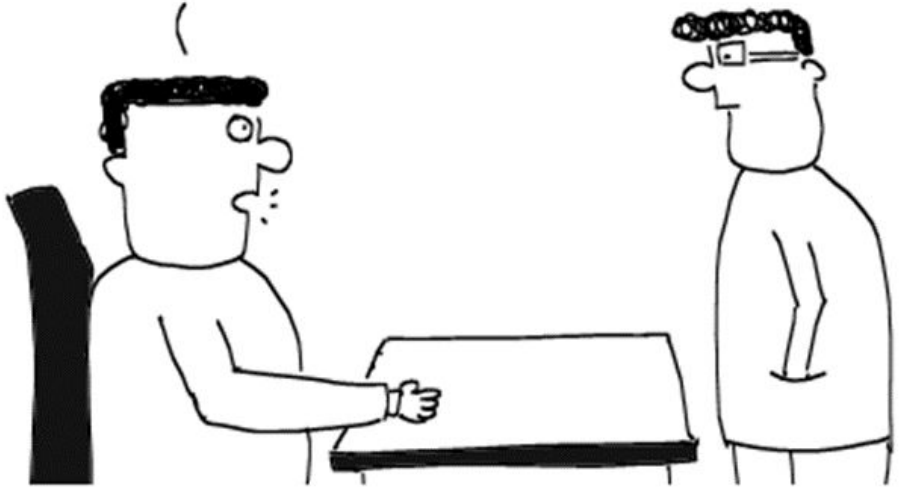
- Explain the difference between an editor and an IDE.
- List the 5 good practices related to coding
- Demonstrate practical knowledge on writing better code

Coding and Reproducibility

Not a lecture on good coding practice, but on good practices to code

- A useful code is going to be a code that allows to reproduce, at least the main results of a study.
- A useful code is a code that you can reuse yourself in 6 months, that somebody else can reuse

WHO CARES ABOUT THE USERS!!
IF THEY HAVE A PROBLEM THEY
SHOULD SOLVE IT ON THEIR OWN.



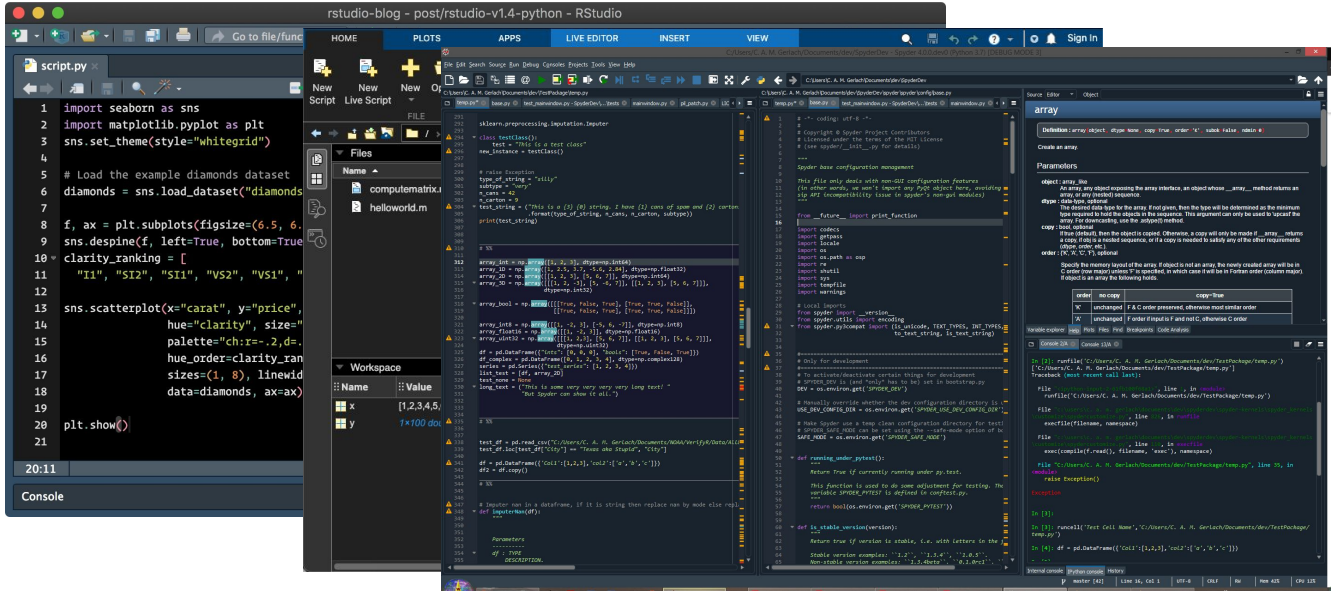
* That is how the "debug" feature came in to being...

Brainstuck.com

Coding Environment

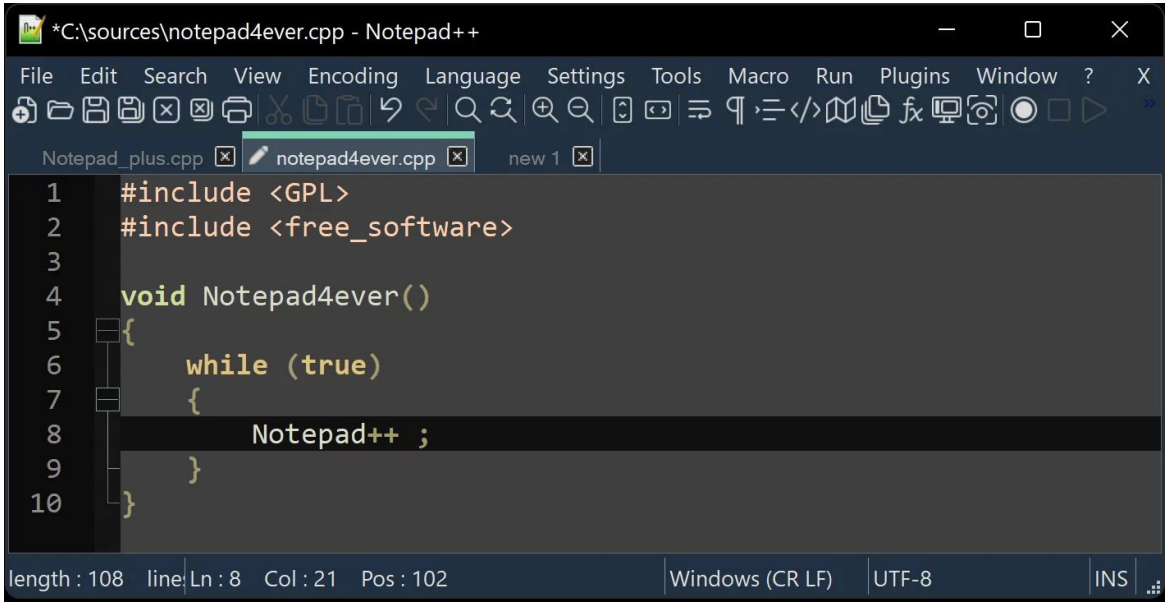
I love IDE

An **i**ntegrated **d**evelopment **e**nvironment is software that combines common developer tools into a single **g**raphical **u**ser **i**nterface (GUI)



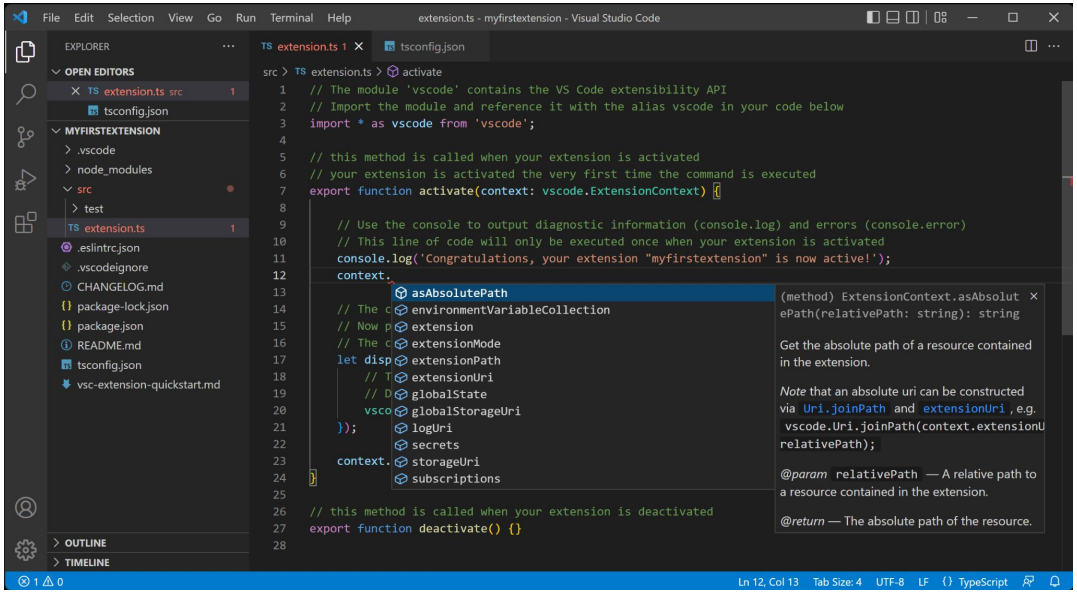
Still - you NEED a descent ‘generic’ editor (speed, flexibility and portability)

The basic: notepad++



Still - you NEED a descent ‘generic’ editor (speed, flexibility and portability)

Plenty of others exist … the current bad boy? VS Code



Have you noticed something in my pictures?

Yes … always in ‘dark mode’

Do what you want with your software(s), but you are going to spend many hours during your thesis using those tools - spend some time customizing to your need and taste (colours, layout, etc).

What software(s) are you using?-
Any customization tips to share?

The ultimate editing tools?

Linux/Mac your console has **Bash**

Windows - well since you ARE using Git, you have **Git Bash**

→ the *Bourne Again Shell* is small and allows to quickly do many things (see e.g. https://github.com/CPernet/Quicksheets/blob/main/bash/bash.mkd for pretty much the only commends you need)

→ it includes **Vim** (powerful but let's face it hard to learn) and **nano** (the simplest yet powerful way to edit quickly anything)

Good practices to code

Cool, we have seen (1) and (4) already

1. Documenting, commenting & literate programming
2. Naming and externalisation
3. Functionalizing (Many small functions is preferred over a huge one. This is much easier to understand what each chunk does and increases re-usability. It also makes it easier to test)

→ a header in each function that explains what is does ; see point 1

4. Version control
5. Testing

Naming variables

File naming - principles (reminder)

Principles for file naming
- human readable
- machine readable
- plays well with default sorting and ordering

Use “_” underscore and “-” hyphen
Use consistent/different capitalizations
Avoid spaces, punctuation, accented and special characters

Variable naming - principles

Principles for naming variables

- human readable, i.e. meaningful (‘s’ vs. ‘sub’ vs. ‘subject’)
- possibly use “_”` underscore (“-” is minus so not possible)
- avoid ~~spaces, punctuation~~, accented and special characters
- use consistent/different capitalizations

What makes a good name?

Long vs Short

Slower to type

Error prone

Easier to understand

Fewer collisions

Shorter lines

Easier to read

Good for

- frequently used
- “Local” within functions
- briefly used

Long vs Short

~~Slower to type~~ <TAB> it

~~Error prone~~

Easier to understand

(Self-documenting)

Fewer collisions

Shorter lines

Easier to read

Good for

- frequently used
- “Local” within functions
- briefly used

Capitalization as a convention

The idea is that capitalization helps you distinguish different variable classes in your code.

You can use your own convention or adopt ‘classic’ conventions - some companies have their own coding style (e.g. https://google.github.io/styleguide/).

functionnames,
variableNames and methodNames
ClassNames and EnumNames,
SYMBOLIC_CONSTANTS

Externalization

Externalisation

- Avoid “hard-coding”
  - If you use a number twice, make it a variable!
- Never change directory
  - And never hard-code paths

Externalisation

What's wrong with this code?

```
% for each subject
for i=1:10
  x = readData(i)
  % calculate subject mean
  meanSubj(i) = mean(x)
end
% calculate standard error
stdError = std(meanSubj)/sqrt(10)
```

```
# for each subject
for i in range(10)
  x = readData(i)
  # calculate subject mean
  meanSubj[i] = numpy.mean(x)
end
# calculate standard error
stdError = std(meanSubj)/sqrt(10)
```

Externalisation

You will have to re-code
if you get 11 subjects !

```
% for each subject
for i=1:10
  x = readData(i)
  % calculate subject mean
  meanSubj(i) = mean(x)
end
% calculate standard error
stdError = std(meanSubj)/sqrt(10)
```

Externalisation

```
NumSubj = 10; % number of subjects
```

```
% for each subject
for i=1:10
  x = readData(i)
  % calculate subject mean
  meanSubj(i) = mean(x)
end
% calculate standard error
stdError = std(meanSubj)/sqrt(10)
```

```
std(meanSubj)/size(meanSubj,1)
```

Externalisation

- Why?
  - Legible (what does that number *mean*?)
  - Modifiable
  - Saved along with data

- □ Functionalizing

Functionalizing

Functionalizing

- **Function vs Script**
- **Why use functions?**
- **When would you use them?**

Functionalizing

- Why functions?
  - ↓ repetition, ↑ reuse
  - alteration in single place
  - sandboxing (stack frames)
    - privacy!
  - semantic / ‘self-documenting’

When to Functionalize

- “Meaningful unit of computation”
- “self-contained”
- defined inputs and outputs

- **Refactoring**: to avoid copy/paste
- Sharing
  - Different experiments
  - Friends!

Functionalizing:
“Doctrine of Referential Transparency”

A function should produce ***identical results*** whenever it is called with the same parameters, unless documented

Calling a function should *never change anything* apart from the output

= A bit like a mathematical function

```
for subject=1:10
 cd([‘SUB’ subject]);
 load datafile
 cd ..
 sumCond1 = 0;
 sumCond2 = 0;
 numCond1 = 0;
 numCond2 = 0;
 for trial=1:100
  if COND(i) == 1
   sumCond1 = sumCond1+RT(i);
   numCond1 = numCond1+1;
  elseif COND(i) == 2
   sumCond2 = sumCond2+RT(i);
   numCond2 = numCond2+1;
  end
 end
 meanCond1(subject) = sumCond1/numCond1;
 meanCond2(subject) = sumCond2/numCond2;
end
```

What’s wrong is this code?

```
for subject=1:10
 cd(['SUB' subject]);
 load datafile
 cd ..
 sumCond1 = 0;
 sumCond2 = 0;
 numCond1 = 0;
 numCond2 = 0;
 for trial=1:100
  if COND(i) == 1
   sumCond1 = sumCond1+RT(i);
   numCond1 = numCond1+1;
  elseif COND(i) == 2
   sumCond2 = sumCond2+RT(i);
   numCond2 = numCond2+1;
  end
 end
 meanCond1(subject) = sumCond1/numCond1;
 meanCond2(subject) = sumCond2/numCond2;
end
```

- Changing folders
  - disrupts path for .m files
  - disorientating
  - if errors occur, you have to manually return
- Loading variables into the base stack frame is confusing (what is COND and RT? -- from datafile)
  - You might overwrite something
  - You can't tell what is current or old

BONUS:

(1) loop backward avoid declaring variables because an array is made in memory

(2) using logical vectors rather than loops when possible, e.g. sum(RT(COND==1))

Referential transparency

- Explicit message passing
  - No globals, save / load
- If not
  - all global requirements need commenting

Functionalizing AND versioning

- Maintain all previous functionality
- Conditional execution
- Parsing options

Function vs. Method

Function — a set of instructions that perform a task.

Functions are like recipes. They can execute a set of instructions on data or variables and return the result. The beauty of functions is that they are recyclable. That is, the function can be used repeatedly without having to write the same code again.

Method — a set of instructions that are associated with an object.

A method, like a function, is a set of instructions that perform a task. The difference is that a method is associated with an object, while a function is not.

(from https://www.codecademy.com/article/fwd-js-methods-functions)

Aligning and Linting

= make your code easy to read

lintr

R-CMD-check passing | codecov 100% | CRAN 3.1.0 | lifecycle stable

`{lintr}` provides static code analysis for R. It checks for adherence to a given style, identifying syntax errors and possible semantic issues, then reports them to you so you can take action. Watch lintr in action in the following animation:

checkcode

Check MATLAB code files for possible problems

Syntax

```
checkcode(filename)
checkcode(filename1,...,filenameN)

checkcode( __ ,option1,...,optionN)

info = checkcode( __ ,'-struct')
msg = checkcode( __ ,'-string')
[ __ , filepaths] = checkcode( __ )
```

pytype - 🦆✓

Pytype checks and infers types for your Python code - without requiring type annotations. Pytype can:

- Lint plain Python code, flagging common mistakes such as misspelle attribute names, incorrect function calls, and much more, even acros file boundaries.
- Enforce user-provided type annotations. While annotations are optional for pytype, it will check and apply them where present.

andalone files ("pyi files"), which can
source with a provided merge-pyi

RUFF

Lint at lightspeed

An extremely fast Python linter, written in Rust.

```
% read the data
GMd = readtable(['nrudataset' filesep 'GrayMatter_volumes.csv'],'ReadRowNames',false);
WMd = readtable(['nrudataset' filesep 'WhiteMatter_volumes.csv'],'ReadRowNames',false);
CSFd = readtable(['nrudataset' filesep 'CSF_volumes.csv'],'ReadRowNames',false);
GMt = readtable(['ds003653' filesep 'GrayMatter_volumes.csv'],'ReadRowNames',false);
WMt = readtable(['ds003653' filesep 'WhiteMatter_volumes.csv'],'ReadRowNames',false);
CSFt = readtable(['ds003653' filesep 'CSF_volumes.csv'],'ReadRowNames',false);

%% Volume analyses

TIVd = [GMd{:,1}+WMd{:,1}+CSFd{:,1} GMd{:,2}+WMd{:,2}+CSFd{:,2} GMd{:,3}+WMd{:,3}+CSFd{:,3} GMd{:,4}+WMd{:,4}+CSFd{:,4}].*1000;
TIVt = [GMt{:,1}+WMt{:,1}+CSFt{:,1} GMt{:,2}+WMt{:,2}+CSFt{:,2} GMt{:,3}+WMt{:,3}+CSFt{:,3} GMt{:,4}+WMt{:,4}+CSFt{:,4}].*1000;

[TIVd_est, TIVd_CI] = rst_data_plot(TIVd, 'estimator','trimmed mean','newfig','sub');
[TIVt_est, TIVt_CI] = rst_data_plot(TIVt, 'estimator','trimmed mean','newfig','sub');
[GMd_est, CId_GM,~,K1] = rst_data_plot(GMd{:,:}.*1000, 'estimator','trimmed mean','newfig','sub');
[WMd_est, CId_WM,~,K2] = rst_data_plot(WMd{:,:}.*1000, 'estimator','trimmed mean','newfig','sub');
[CSFd_est, CId_CSF,~,K3] = rst_data_plot(CSFd{:,:}.*1000, 'estimator','trimmed mean','newfig','sub');
[GMt_est, CIt_GM,~,K4] = rst_data_plot(GMt{:,:}.*1000, 'estimator','trimmed mean','newfig','sub');
[WMt_est, CIt_WM,~,K5] = rst_data_plot(WMt{:,:}.*1000, 'estimator','trimmed mean','newfig','sub');
[CSFt_est, CIt_CSF,~,K6] = rst_data_plot(CSFt{:,:}.*1000, 'estimator','trimmed mean','newfig','sub');
```

```
% read the data
GMd  = readtable(['nrudataset' filesep 'GrayMatter_volumes.csv'],'ReadRowNames',false);
WMd  = readtable(['nrudataset' filesep 'WhiteMatter_volumes.csv'],'ReadRowNames',false);
CSFd = readtable(['nrudataset' filesep 'CSF_volumes.csv'],'ReadRowNames',false);
GMt  = readtable(['ds003653' filesep 'GrayMatter_volumes.csv'],'ReadRowNames',false);
WMt  = readtable(['ds003653' filesep 'WhiteMatter_volumes.csv'],'ReadRowNames',false);
CSFt = readtable(['ds003653' filesep 'CSF_volumes.csv'],'ReadRowNames',false);

%% Volume analyses

TIVd = [GMd{:,1}+WMd{:,1}+CSFd{:,1} ...
    GMd{:,2}+WMd{:,2}+CSFd{:,2} ...
    GMd{:,3}+WMd{:,3}+CSFd{:,3} ...
    GMd{:,4}+WMd{:,4}+CSFd{:,4}].*1000;

TIVt = [GMt{:,1}+WMt{:,1}+CSFt{:,1} ...
    GMt{:,2}+WMt{:,2}+CSFt{:,2} ...
    GMt{:,3}+WMt{:,3}+CSFt{:,3} ...
    GMt{:,4}+WMt{:,4}+CSFt{:,4}].*1000;

[TIVd_est, TIVd_CI]      = rst_data_plot(TIVd, 'estimator','trimmed mean','newfig','sub');
[TIVt_est, TIVt_CI]      = rst_data_plot(TIVt, 'estimator','trimmed mean','newfig','sub');
[GMd_est, CId_GM,~,K1]   = rst_data_plot(GMd{:,:}.*1000, 'estimator','trimmed mean','newfig','sub');
[WMd_est, CId_WM,~,K2]   = rst_data_plot(WMd{:,:}.*1000, 'estimator','trimmed mean','newfig','sub');
[CSFd_est, CId_CSF,~,K3] = rst_data_plot(CSFd{:,:}.*1000, 'estimator','trimmed mean','newfig','sub');
[GMt_est, CIt_GM,~,K4]   = rst_data_plot(GMt{:,:}.*1000, 'estimator','trimmed mean','newfig','sub');
[WMt_est, CIt_WM,~,K5]   = rst_data_plot(WMt{:,:}.*1000, 'estimator','trimmed mean','newfig','sub');
[CSFt_est, CIt_CSF,~,K6] = rst_data_plot(CSFt{:,:}.*1000, 'estimator','trimmed mean','newfig','sub');
```

Testing

Testing

Do you know what I mean by code/software testing?

Test to be sure that your code does what it is supposed to be doing (outputting the ‘right’ result from your data is just confirmation bias)

Can you think of what needs to be tested and how?

Testing

- **validate changes: provide unit tests** (ensure that the correct results are returned from a function)

= executable requirement (the function does X, if you change and/or fix something it must still do X as designed, and you can test that))

- *More validation*:

 integration test (software modules are combined and tested as a group)

 continuous integration (runs all tests after each new piece of code/change)

https://www.software.ac.uk/testing-your-software

Conclusions

Reproducibility and good practices

- Be organized! This makes sharing easy (folders, names), and sharing allows reproduction
- Find your style (customization, variable names, etc)
- Code with documentation and unit testing allows for quick and easy reuse for reproductions and replications

Intended Learning Outcomes

- Explain the difference between an editor and an IDE.
- List the 5 good practices related to coding
- Demonstrate practical knowledge on writing better code