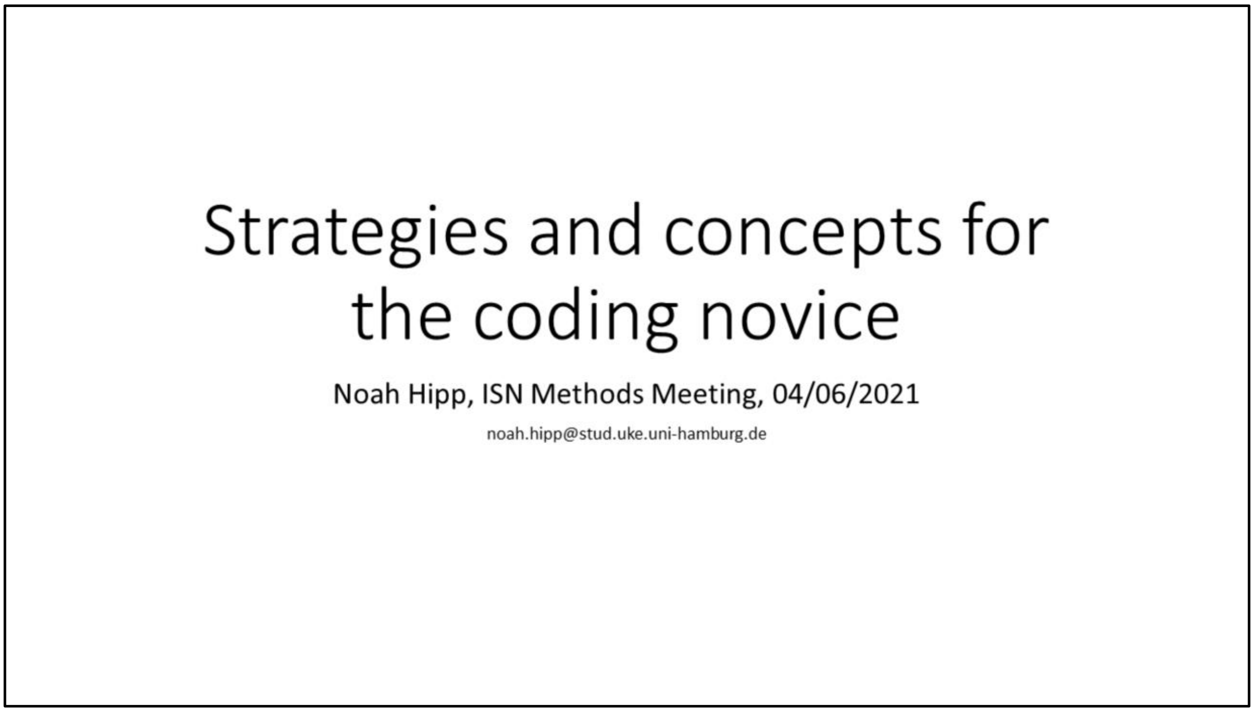
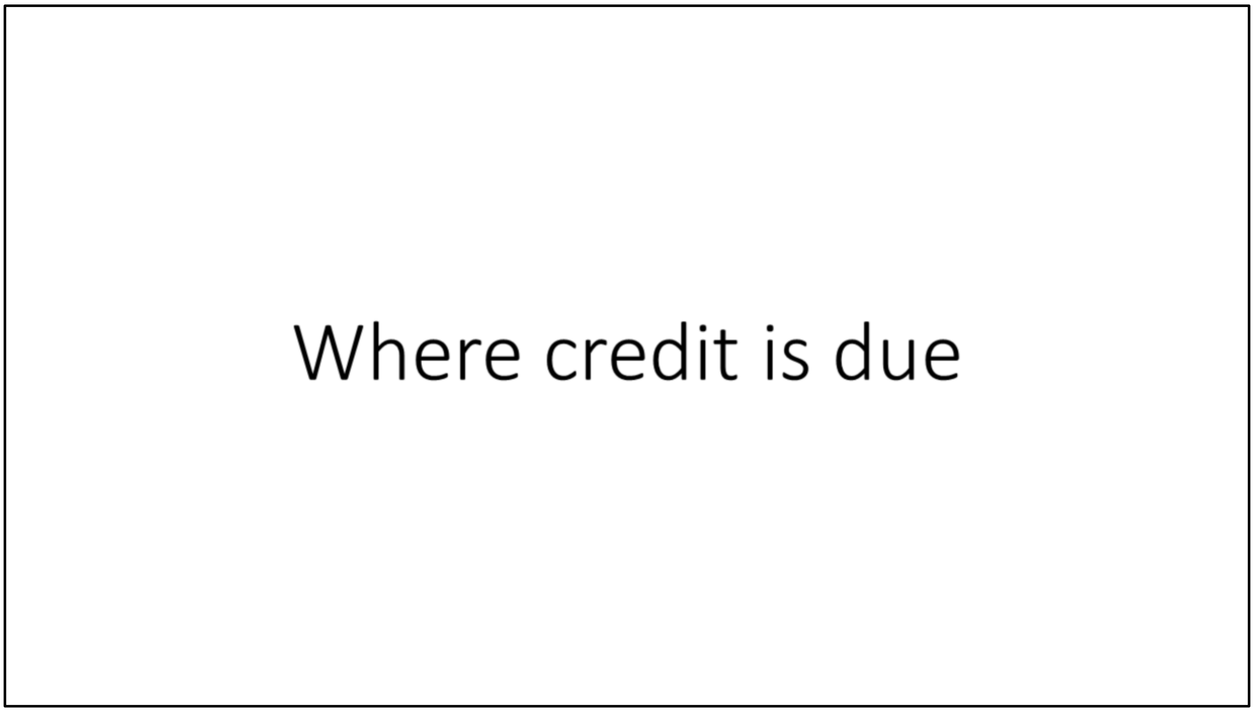


Strategies and concepts for the coding novice

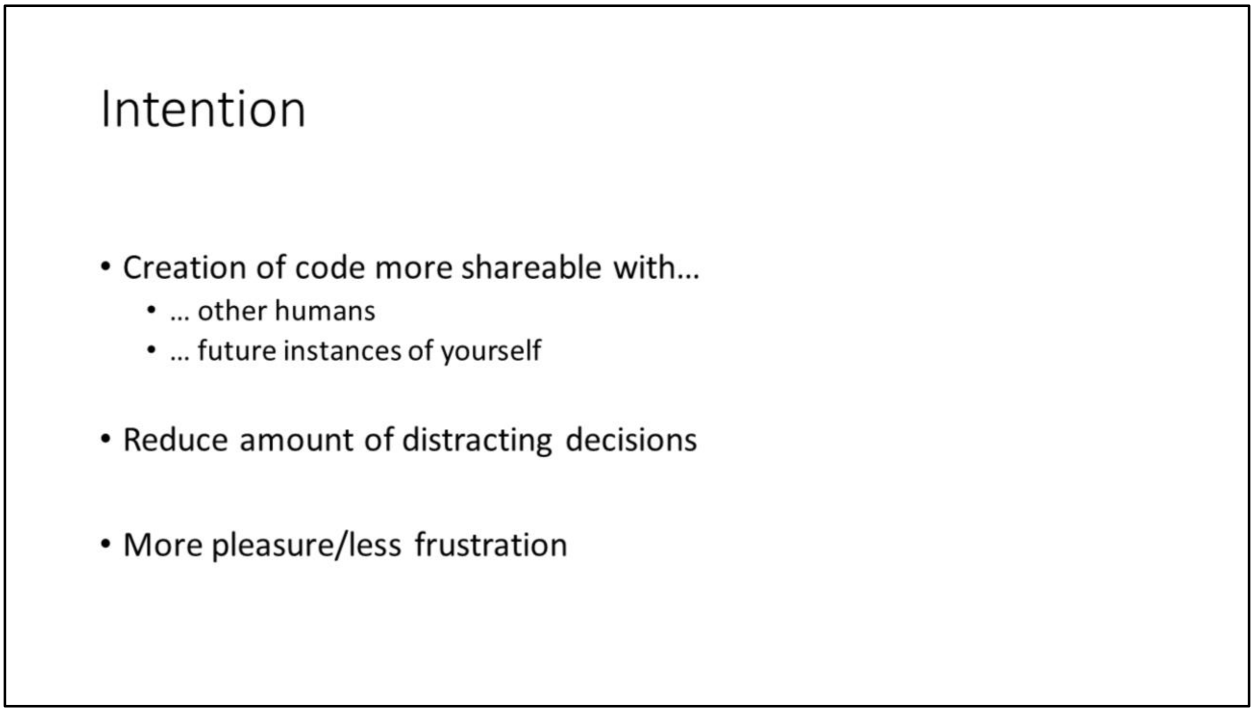
Noah Hipp, ISN Methods Meeting, 04/06/2021
noah.hipp@stud.uke.uni-hamburg.de

Hi everyone, welcome to strategies and concepts for the coding novice.



Where credit is due

When I started at the institute in the Fall of 2018 I hardly knew anything about programming. However, I was lucky enough to be surrounded by very helpful collegues, especially Björn and also Christian, Alexandra, Lea and Lukas who quickly helped me to get the hang of it.



Intention

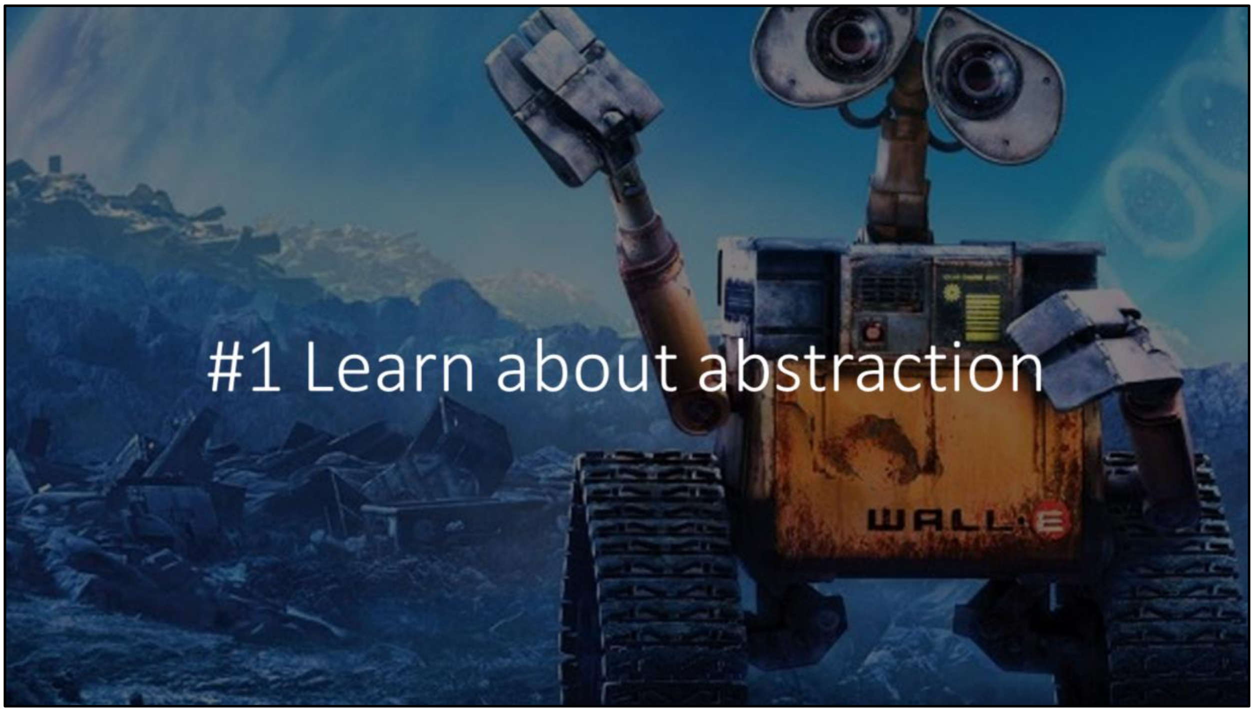
- Creation of code more shareable with…
  - … other humans
  - … future instances of yourself
- Reduce amount of distracting decisions
- More pleasure/less frustration

With this talk I want to facilitate the following:

1. The creation of code that is more shareable with other humans and future instances of yourself. By that I mean still being able to understand the code you write know in year or so.

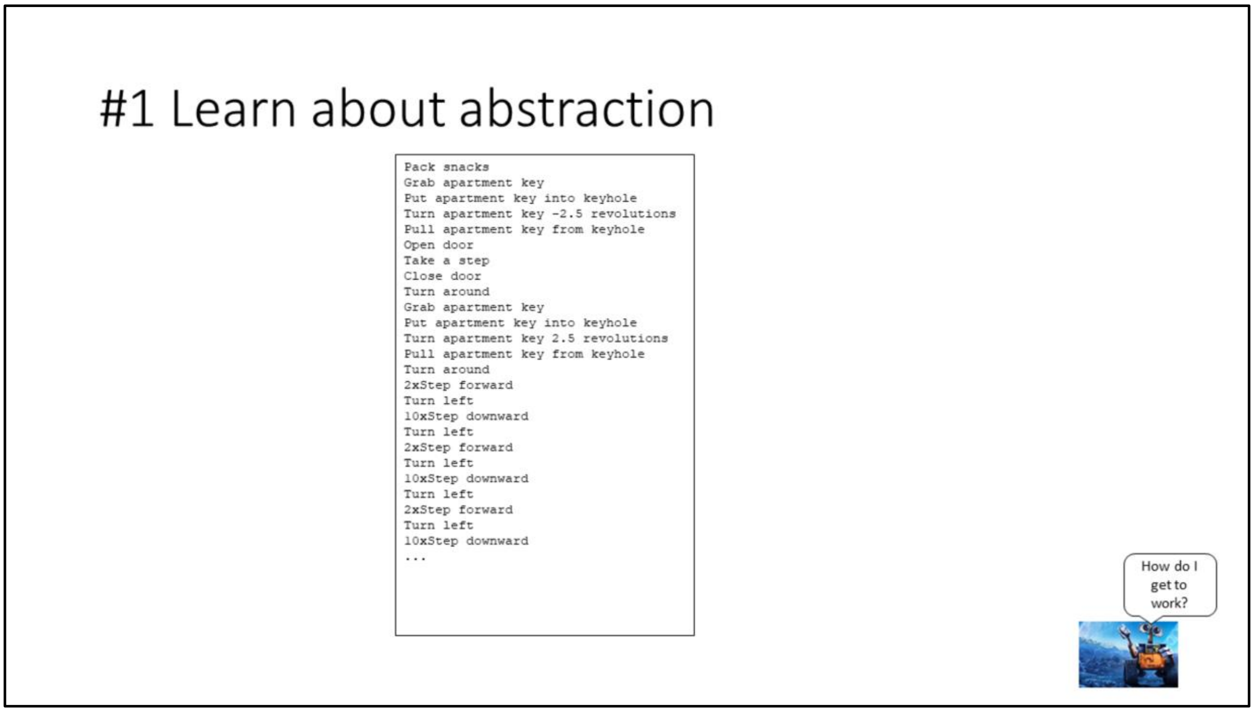
2. Provide some simple guideline suggestions to help you reduce the amount of distracting decisions

3. And to have more pleasure and less frustration in the process.



#1 Learn about abstraction

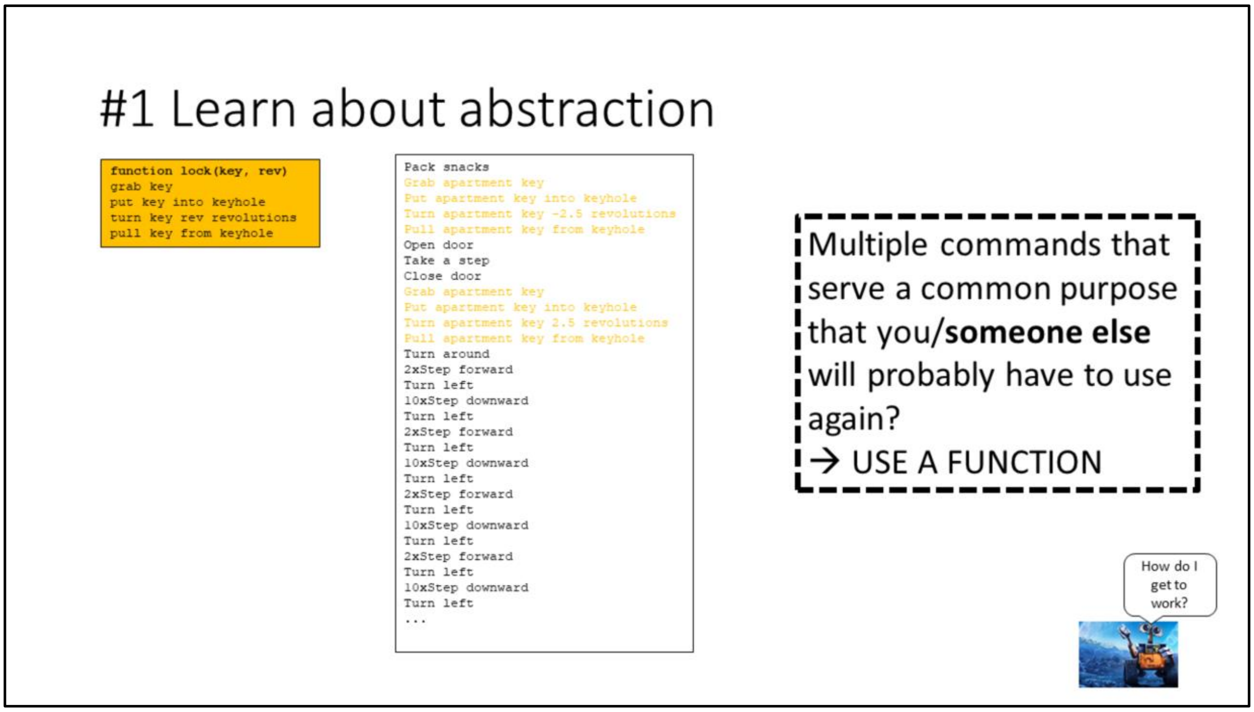
I decided to start with this one because its rather long compared to the other tips I'll present and I have to go a little far a field to make my point but bear with me here. I promise the following ones will be quicker.
Suppose one day this guy [wall-e] pays you a visit. And as wall e is a very diligent creature the first thing he wants to know is how you get to work. He also points out that he is only able to understand very basic instructions like grab something, take a step, take a step up, take a step down etc. You would rather talk about something else but you soon give in and start writing it out. The first few lines end up looking something like this:



#1 Learn about abstraction

```
Pack snacks
Grab apartment key
Put apartment key into keyhole
Turn apartment key -2.5 revolutions
Pull apartment key from keyhole
Open door
Take a step
Close door
Turn around
Grab apartment key
Put apartment key into keyhole
Turn apartment key 2.5 revolutions
Pull apartment key from keyhole
Turn around
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
...
```

We already made use of some abbreviations but it still's still very tedious. Does any of you have an idea how to write this more concise, maybe even by employing a popular strategy from coding? Yeah thats right! We could use functions. The instructions given for unlocking your aparment door are very similar to those for locking it. We could encapsulate them in a function.



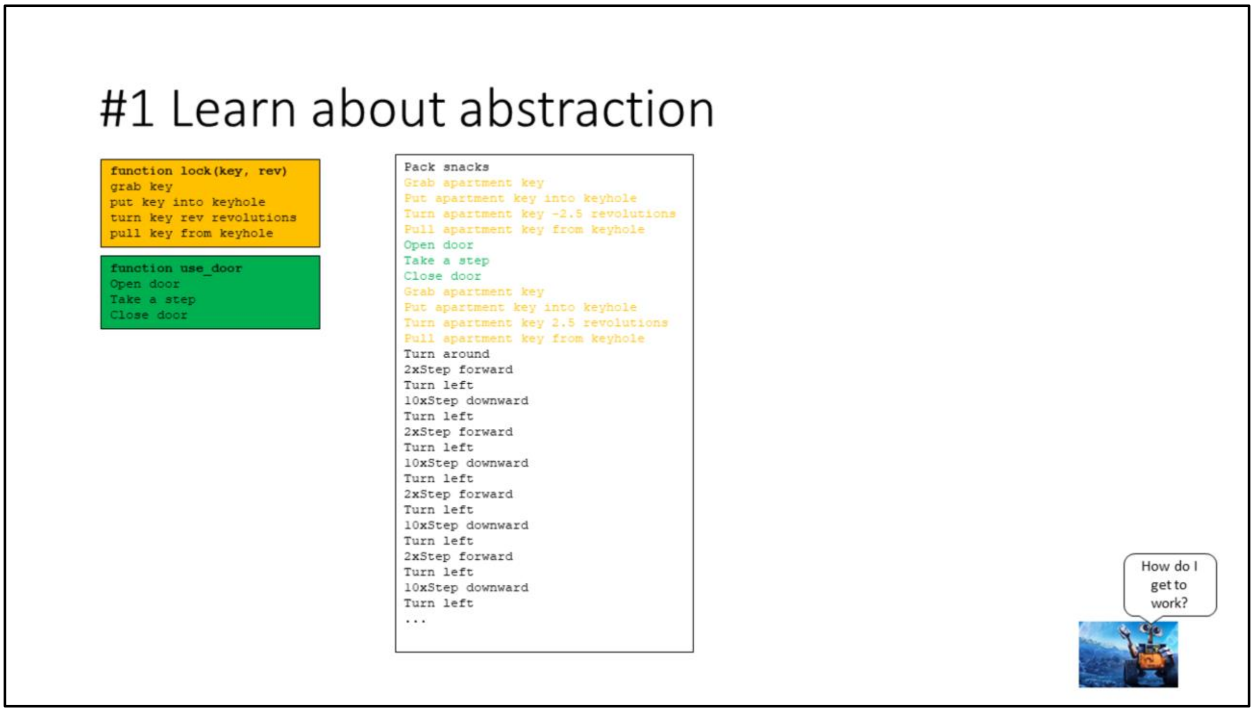
#1 Learn about abstraction

```
function lock(key, rev)
grab key
put key into keyhole
turn key rev revolutions
pull key from keyhole
```

```
Pack snacks
Grab apartment key
Put apartment key into keyhole
Turn apartment key -2.5 revolutions
Pull apartment key from keyhole
Open door
Take a step
Close door
Grab apartment key
Put apartment key into keyhole
Turn apartment key 2.5 revolutions
Pull apartment key from keyhole
Turn around
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
...
```

Multiple commands that serve a common purpose that you/**someone else** will probably have to use again?
→ USE A FUNCTION

This you could refer to as your locking function. You could even make it adaptive by specifing the correct key and amounts to turn. We just did something very important: We put together commands that serve a common purpose so we can easily reuse them in that constellation later on.
We could also apply this for opening the door…



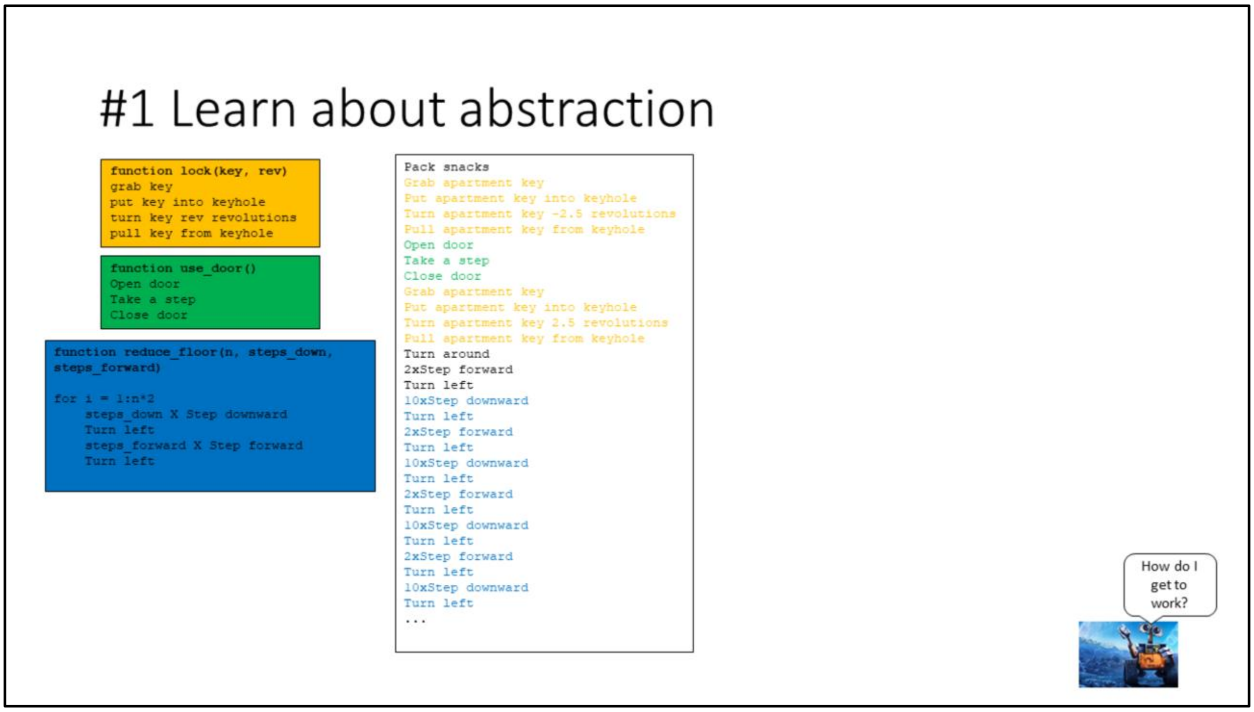
#1 Learn about abstraction

```
function lock(key, rev)
grab key
put key into keyhole
turn key rev revolutions
pull key from keyhole
```

```
function use_door
Open door
Take a step
Close door
```

```
Pack snacks
Grab apartment key
Put apartment key into keyhole
Turn apartment key -2.5 revolutions
Pull apartment key from keyhole
Open door
Take a step
Close door
Grab apartment key
Put apartment key into keyhole
Turn apartment key 2.5 revolutions
Pull apartment key from keyhole
Turn around
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
...
```

And for going down multiple floors…



#1 Learn about abstraction

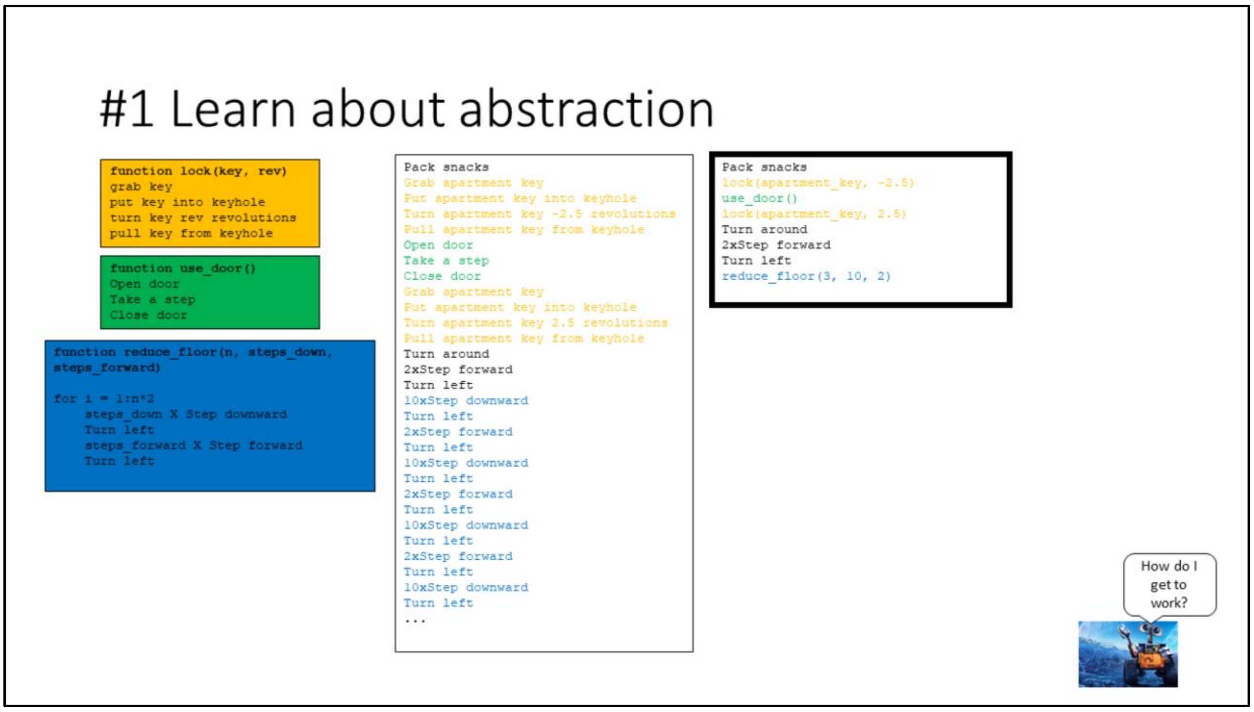
```
function lock(key, rev)
grab key
put key into keyhole
turn key rev revolutions
pull key from keyhole
```

```
function use_door()
Open door
Take a step
Close door
```

```
function reduce_floor(n, steps_down,
steps_forward)

for i = 1:n*2
    steps_down X Step downward
    Turn left
    steps_forward X Step forward
    Turn left
```

```
Pack snacks
Grab apartment key
Put apartment key into keyhole
Turn apartment key -2.5 revolutions
Pull apartment key from keyhole
Open door
Take a step
Close door
Grab apartment key
Put apartment key into keyhole
Turn apartment key 2.5 revolutions
Pull apartment key from keyhole
Turn around
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
...
```

With our new function stack we can rewrite this code



#1 Learn about abstraction

```
function lock(key, rev)
grab key
put key into keyhole
turn key rev revolutions
pull key from keyhole
```

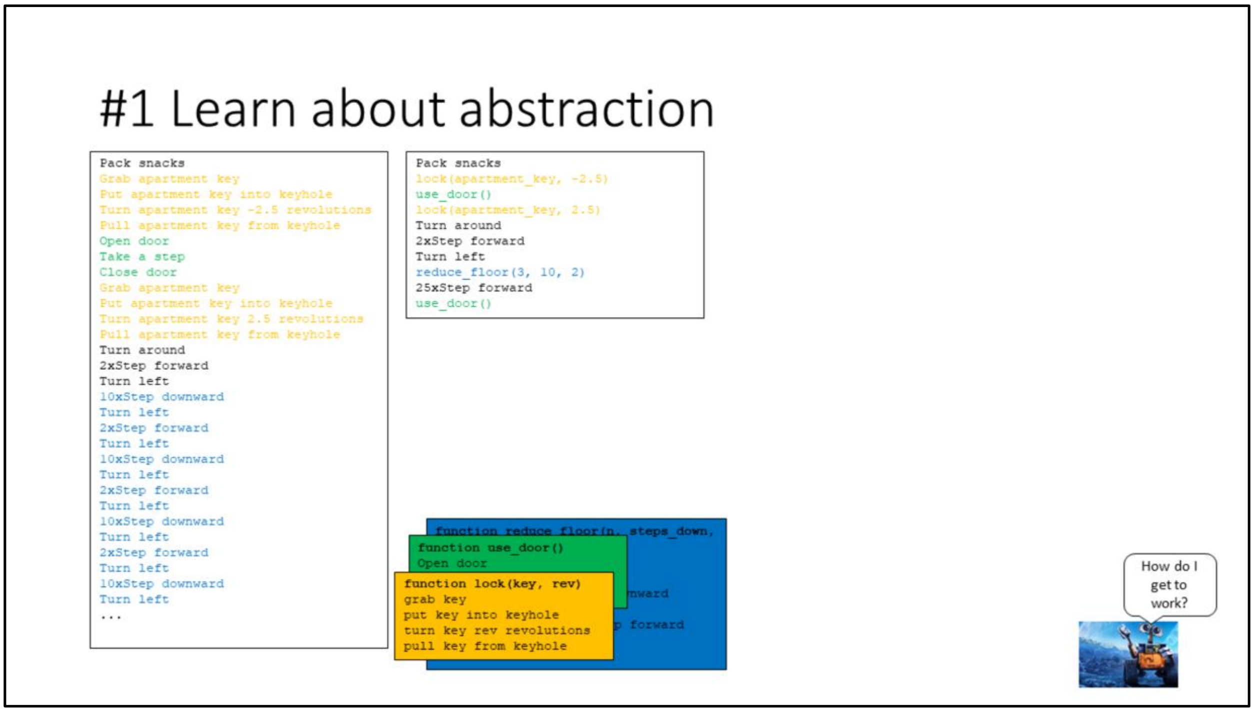
```
function use_door()
Open door
Take a step
Close door
```

```
function reduce_floor(n, steps_down,
steps_forward)

for i = 1:n*2
    steps_down X Step downward
    Turn left
    steps_forward X Step forward
    Turn left
```

```
Pack snacks
Grab apartment key
Put apartment key into keyhole
Turn apartment key -2.5 revolutions
Pull apartment key from keyhole
Open door
Take a step
Close door
Grab apartment key
Put apartment key into keyhole
Turn apartment key 2.5 revolutions
Pull apartment key from keyhole
Turn around
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
...
```

```
Pack snacks
lock(apartment_key, -2.5)
use_door()
lock(apartment_key, 2.5)
Turn around
2xStep forward
Turn left
reduce_floor(3, 10, 2)
```

Lets rearrange this a bit and add a little stuff



#1 Learn about abstraction

```
Pack snacks
Grab apartment key
Put apartment key into keyhole
Turn apartment key -2.5 revolutions
Pull apartment key from keyhole
Open door
Take a step
Close door
Grab apartment key
Put apartment key into keyhole
Turn apartment key 2.5 revolutions
Pull apartment key from keyhole
Turn around
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
...
```

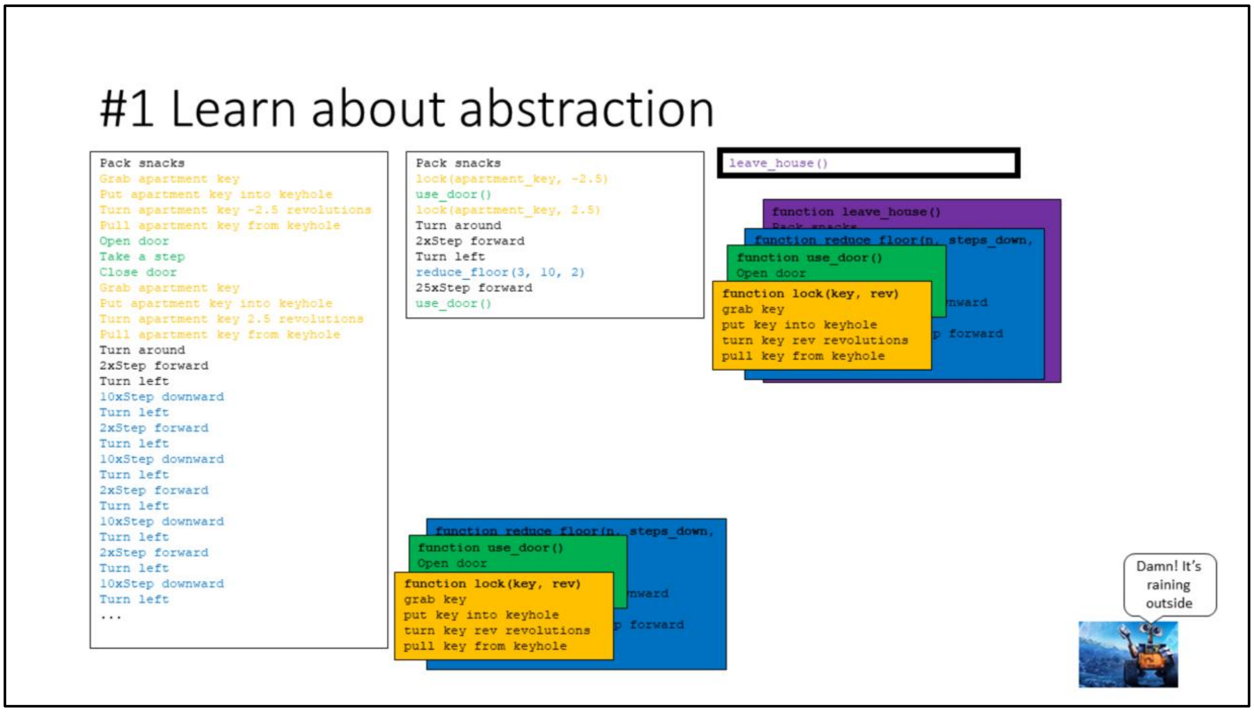
```
Pack snacks
lock(apartment_key, -2.5)
use_door()
lock(apartment_key, 2.5)
Turn around
2xStep forward
Turn left
reduce_floor(3, 10, 2)
25xStep forward
use_door()
```

```
function reduce_floor(n, steps_down,
                                    nward
                                  p forward
```

```
function use_door()
Open door
```

```
function lock(key, rev)
grab key
put key into keyhole
turn key rev revolutions
pull key from keyhole
```

And take this a step further…



#1 Learn about abstraction

```
Pack snacks
Grab apartment key
Put apartment key into keyhole
Turn apartment key -2.5 revolutions
Pull apartment key from keyhole
Open door
Take a step
Close door
Grab apartment key
Put apartment key into keyhole
Turn apartment key 2.5 revolutions
Pull apartment key from keyhole
Turn around
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
...
```

```
Pack snacks
lock(apartment_key, -2.5)
use_door()
lock(apartment_key, 2.5)
Turn around
2xStep forward
Turn left
reduce_floor(3, 10, 2)
25xStep forward
use_door()
```

```
leave_house()
```

```
function leave_house()
Pack snacks
```

```
function reduce_floor(n, steps_down,
nward
p forward
```

```
function use_door()
Open door
```

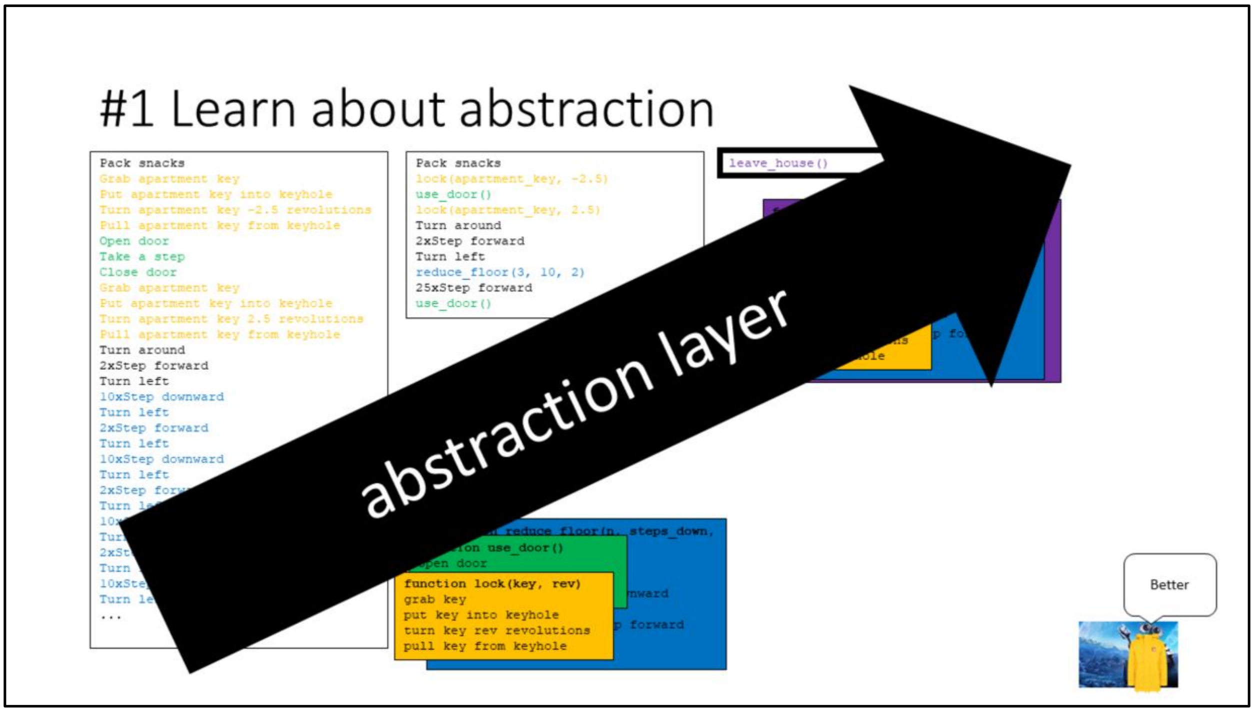
```
function lock(key, rev)
grab key
put key into keyhole
turn key rev revolutions
pull key from keyhole
```

```
function reduce_floor(n, steps_down,
nward
p forward
```

```
function use_door()
Open door
```

```
function lock(key, rev)
grab key
put key into keyhole
turn key rev revolutions
pull key from keyhole
```

We know encapsulated all the instructions for wall e to leave my apartment in one function. Don't get me wrong here: The goal is not to encapsulate all your code into 1 function.



#1 Learn about abstraction

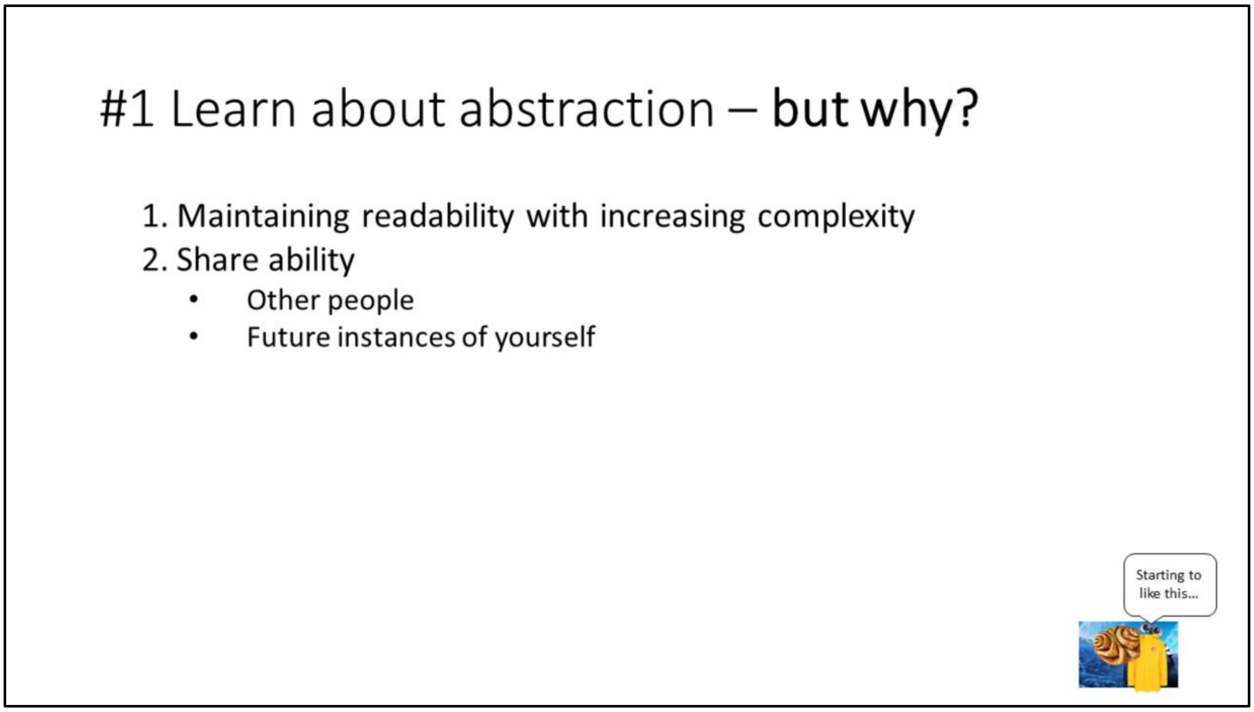
```
Pack snacks
Grab apartment key
Put apartment key into keyhole
Turn apartment key -2.5 revolutions
Pull apartment key from keyhole
Open door
Take a step
Close door
Grab apartment key
Put apartment key into keyhole
Turn apartment key 2.5 revolutions
Pull apartment key from keyhole
Turn around
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forward
Turn left
10xStep downward
Turn left
2xStep forw
Turn le
10x
Tur
2xSt
Turn
10xSte
Turn le
...
```

```
Pack snacks
lock(apartment_key, -2.5)
use_door()
lock(apartment_key, 2.5)
Turn around
2xStep forward
Turn left
reduce_floor(3, 10, 2)
25xStep forward
use_door()
```

```
leave_house()
```

```
n reduce_floor(n, steps_down,
ion use_door()
open door
function lock(key, rev)
grab key
put key into keyhole
turn key rev revolutions
pull key from keyhole
nward
p forward
```

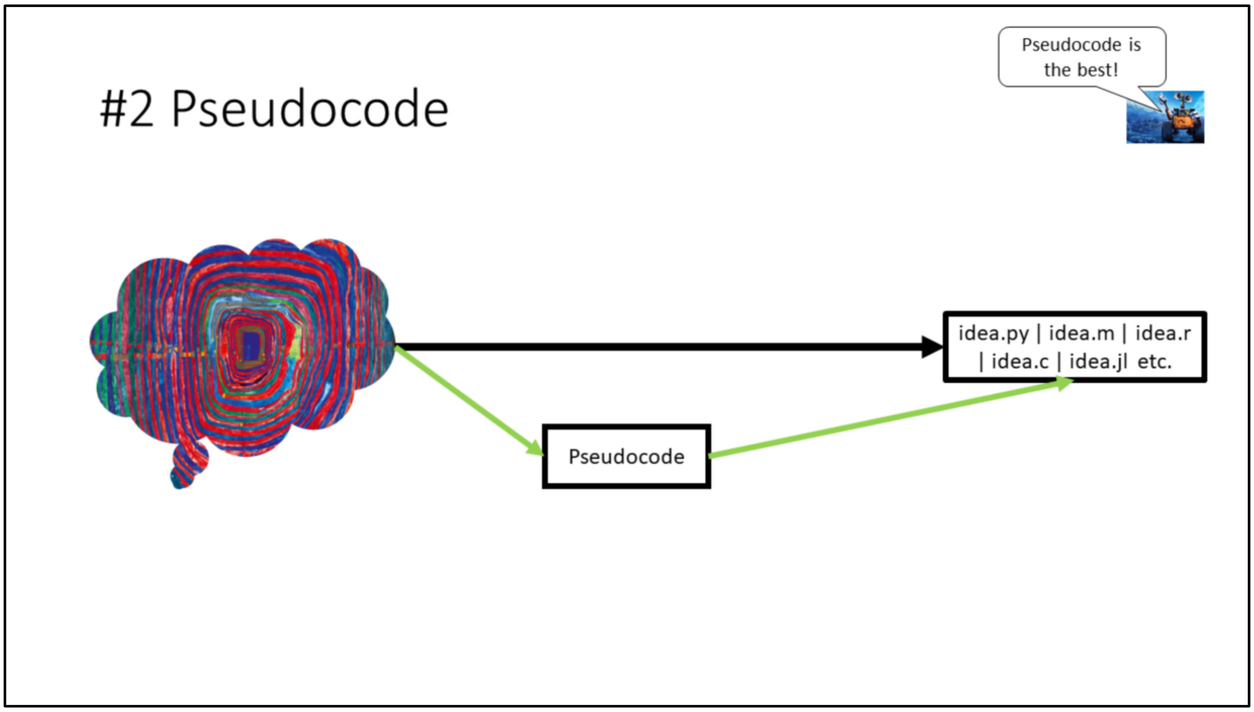
Instead I want to illustrate what it means to move between abstraction layers. For now our vehicle to move up abstraction layers is encapsulating commands that serve a common purpose into functions. If you are good with that you can think about encapsulating functions that serve a common purpose into classes. You trade in some information you openly show for readability/ease of use. It is your job to figure out what abstraction layer is best suited in a given context.



#1 Learn about abstraction – **but why?**

1. Maintaining readability with increasing complexity
2. Share ability
   - Other people
   - Future instances of yourself

But why even bother? Why not just write down what wall e our your computer for that matter needs to do command by command? Who cares about repeating commands, after all there is copy and paste nowadays. The first one is pretty obvious: You can build very complex systems and maintain readability (at least to some degree) by using abstraction techniques. But the second one is even more important in my opinion: Yes, identifying and encapsulating commands that could make up a function takes cognitive effort. However, if you do it well enough you maybe spare other people, and definitely future instances of yourself, this effort as they can just reuse your functions. Think about all the toolboxes, programming languages and operating systems there are nowadays. All due to some really good abstractions.

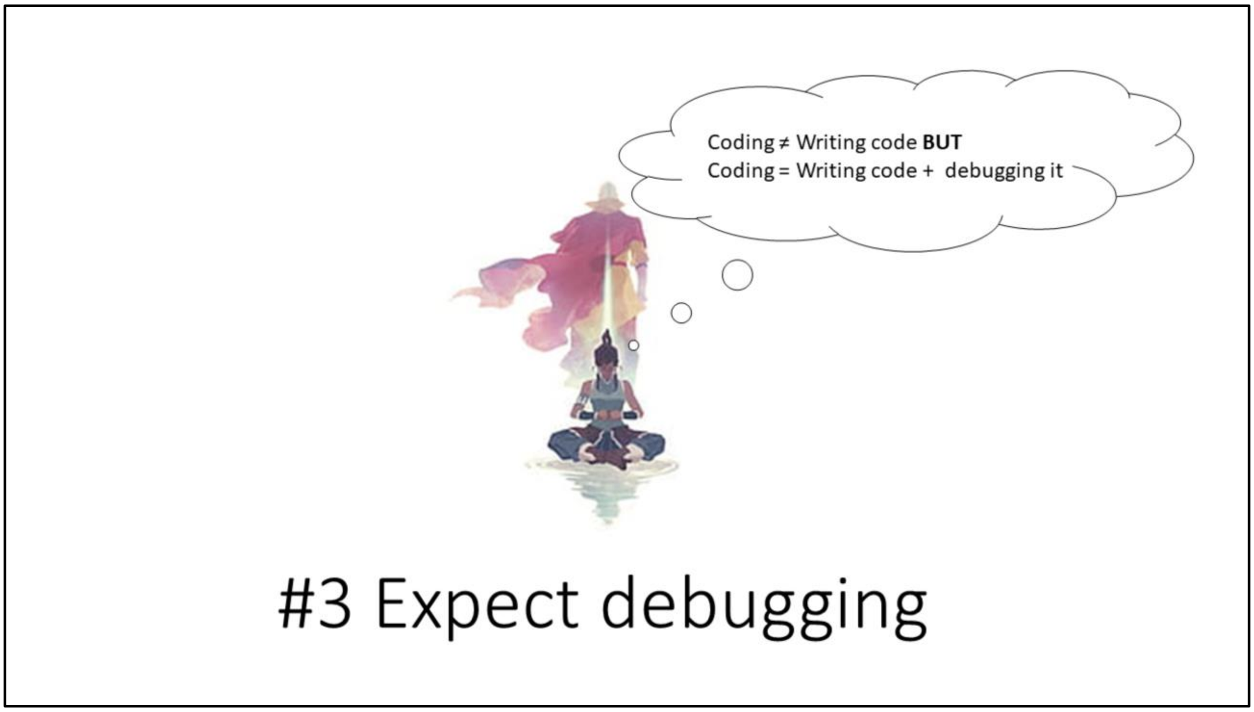


#2 Pseudocode

We didn‘t use a particular programming language to instruct wall-e. Instead we used something commonly referred to as pseudo-code. It cannot be interpreted by a machine, however it gives the reader a quick, language agnostic grasp of what the program does. It illustrates the codes purpose. Why is it useful?

Suppose you have some idea and would like to implement it into code. This process can be intimidating as the target language might still be largely unknown territory for you. One strategy to make your journey a little easier around this is to take a detour via pseudocode. You then try to translate this pseudocode into the actual language. This „chunk by chunk“ translation is often times easier and less intimidating as opposed to trying to translate the whole raw idea all at once. There are some best practices for pseudocode. I would not recommend to worry about them. In my opinion the main perk of pseudocode is that it can be written intuitively and effortless. While your pseudocode might start out looking like what we gave wall-e, over time your pseudocode will get closer and closer to actual code up to a point where you don‘t need to take that detour anymore.

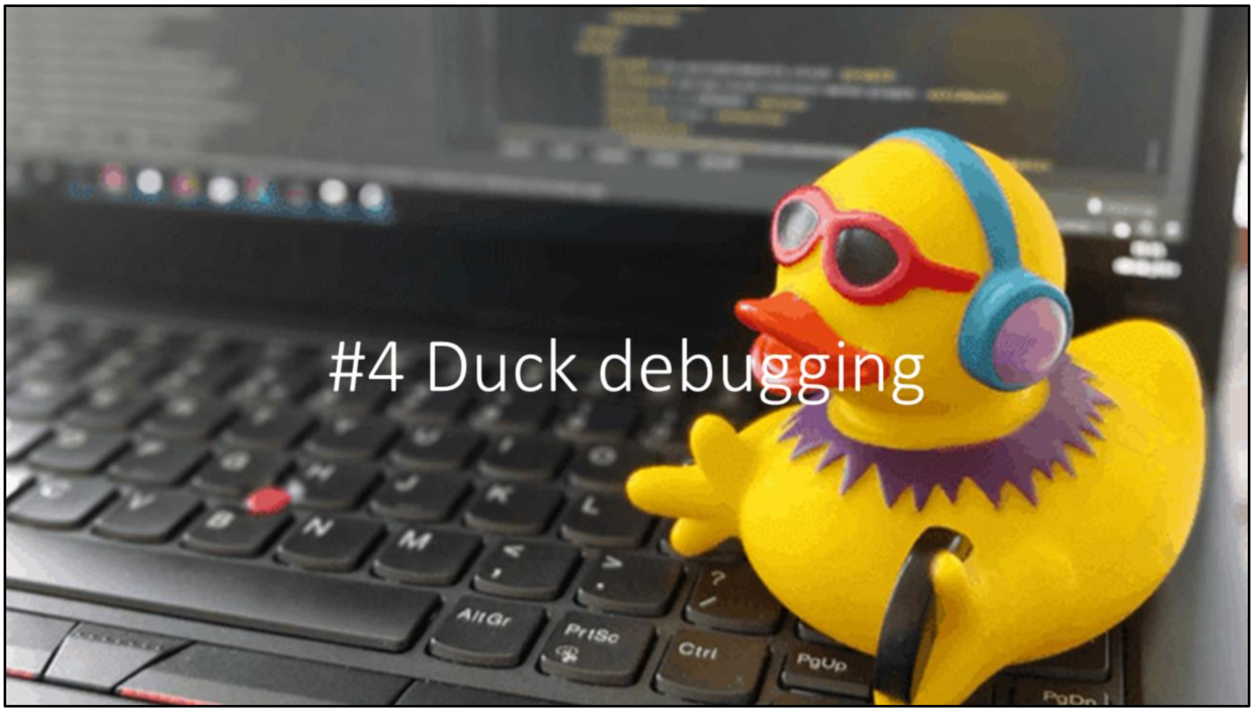
For the next one I wrote a little demo program



Coding ≠ Writing code **BUT**
Coding = Writing code + debugging it

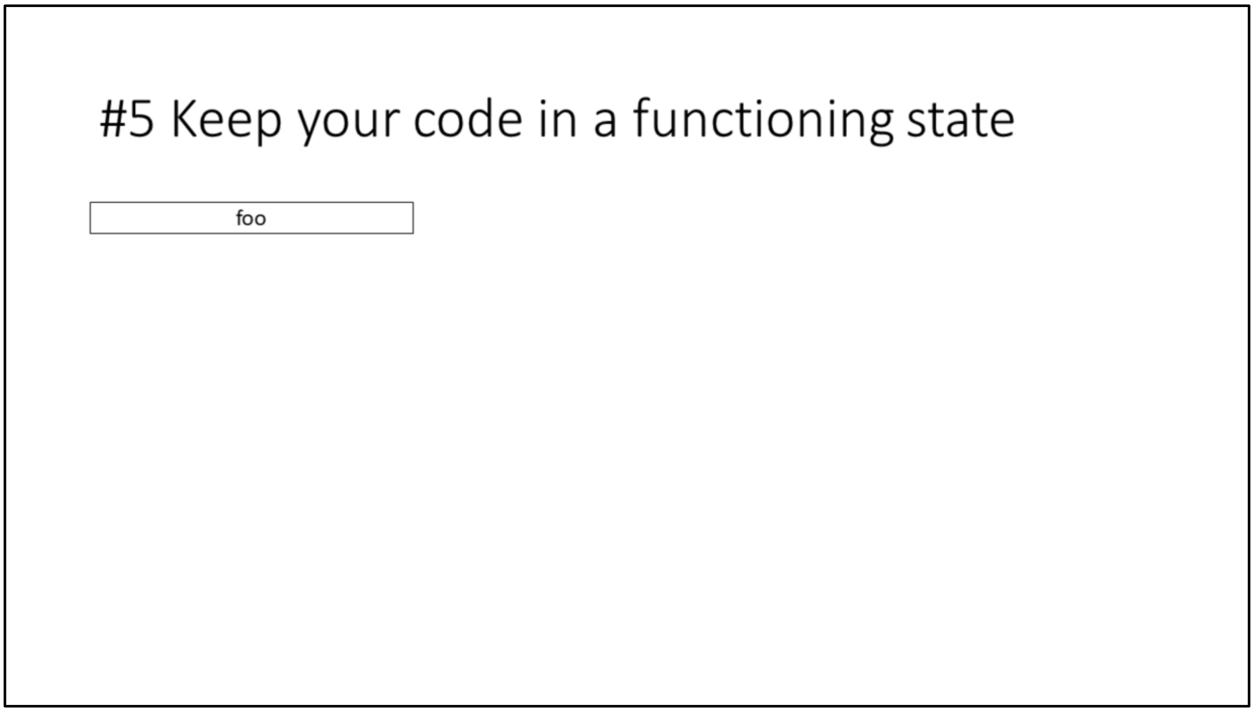
#3 Expect debugging

[shows code that casts error] Which is to expect debugging. This might be one of the biggest sources of frustration when coding. However, you can take some of that aversiveness away if you update your concept of coding to entail debugging. Coding means writing code and debugging it. It’s just part of the experience. Another positive aspect of debugging is learning. Expect debugging to avoid painful prediction errors.



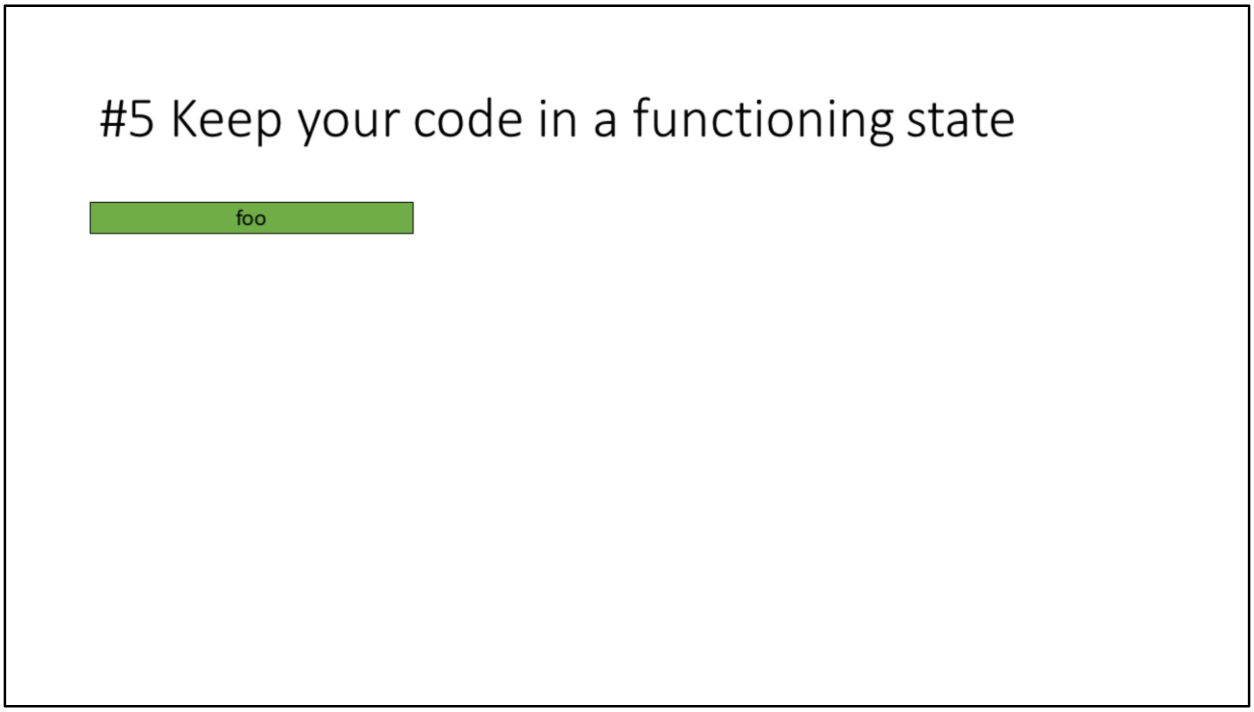
#4 Duck debugging

Though expecting it, you are still at the risk of getting stuck debugging. Chances are you are in home office with nobody around to help you. Perfect chance to give DUCK DEBUGGING a shot. This means explaining
your code to a rubber duck, which forces you to go through it one step at a time often leading to the discovery of the bug. If you do not have access to a rubber duck any object with limited coding knowledge will do until you get one.



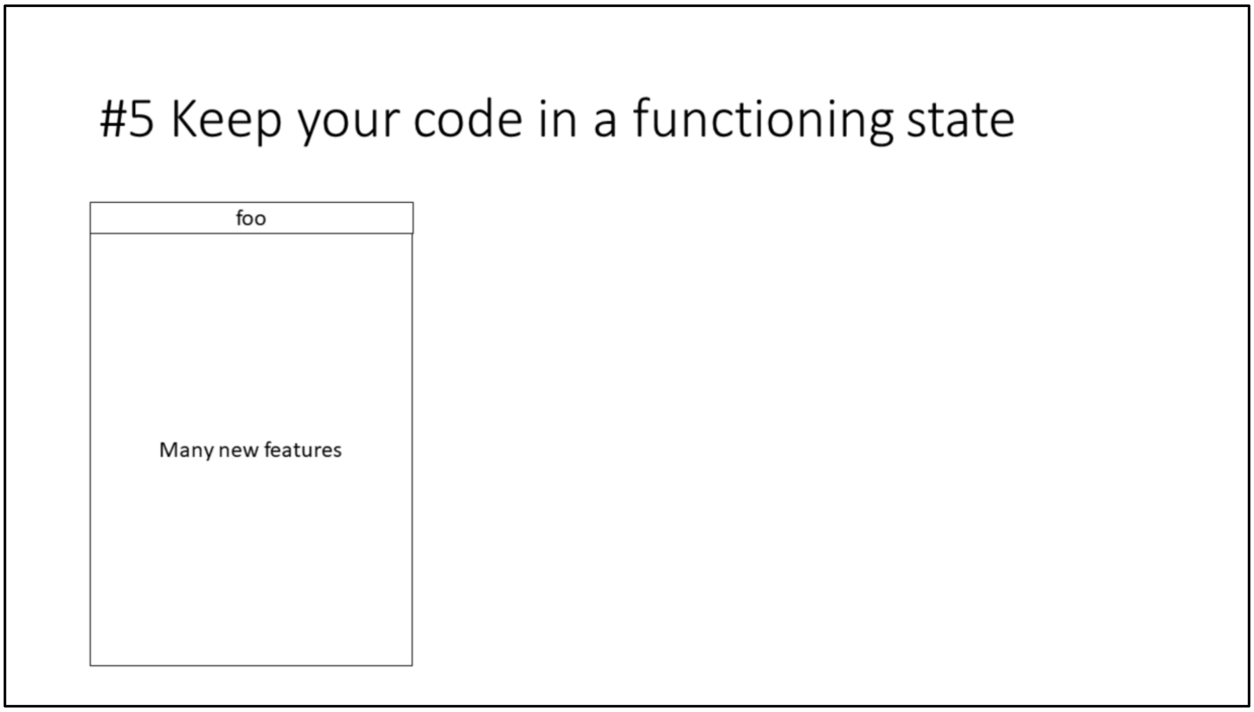
#5 Keep your code in a functioning state

Another technique to help you debugging is to keep your code in a functioning state. Suppose you have a small program called foo.



#5 Keep your code in a functioning state

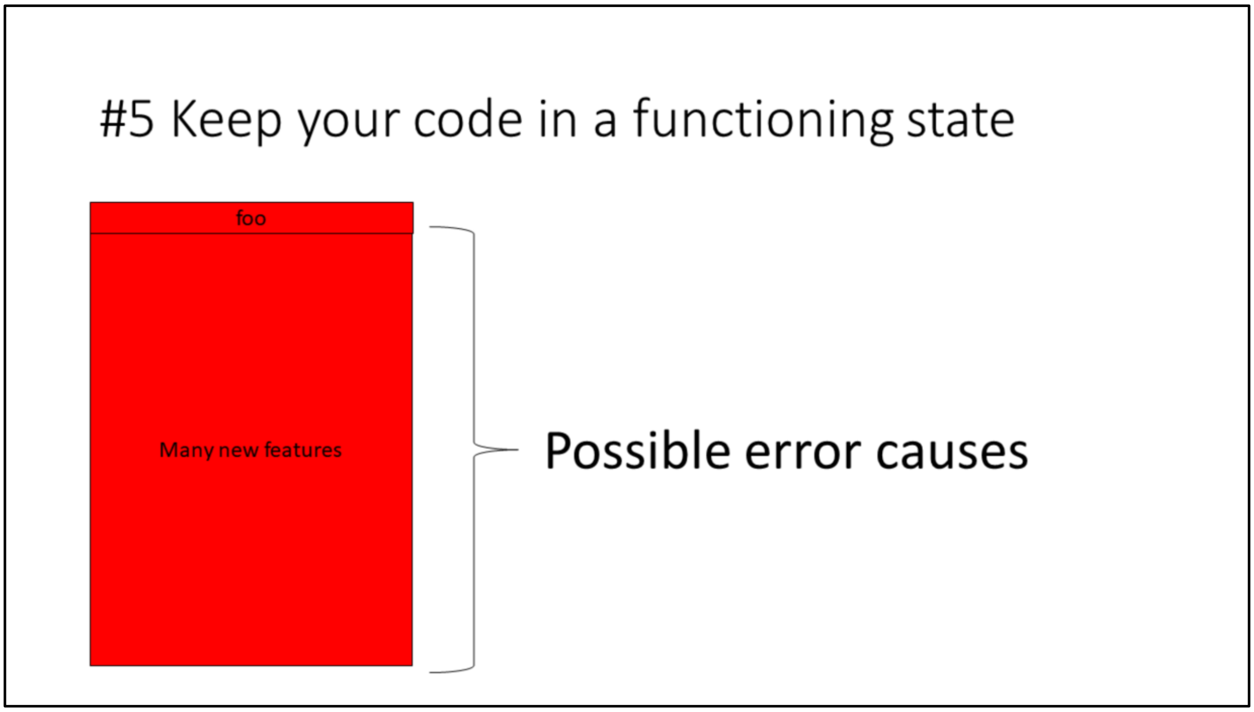
And its working. You want to add some features to it. You want to be fast so you add them all at once.



#5 Keep your code in a functioning state

foo
Many new features

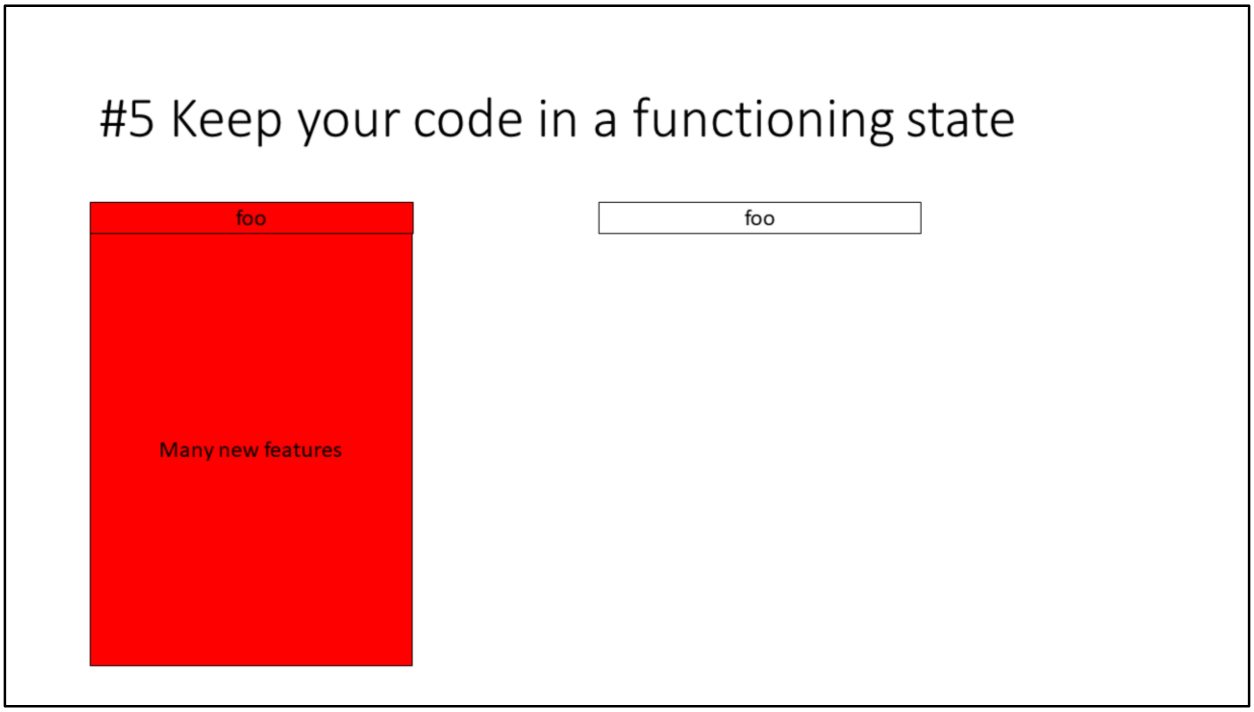
You are proud of your speed and try to run it



#5 Keep your code in a functioning state

foo
Many new features
Possible error causes

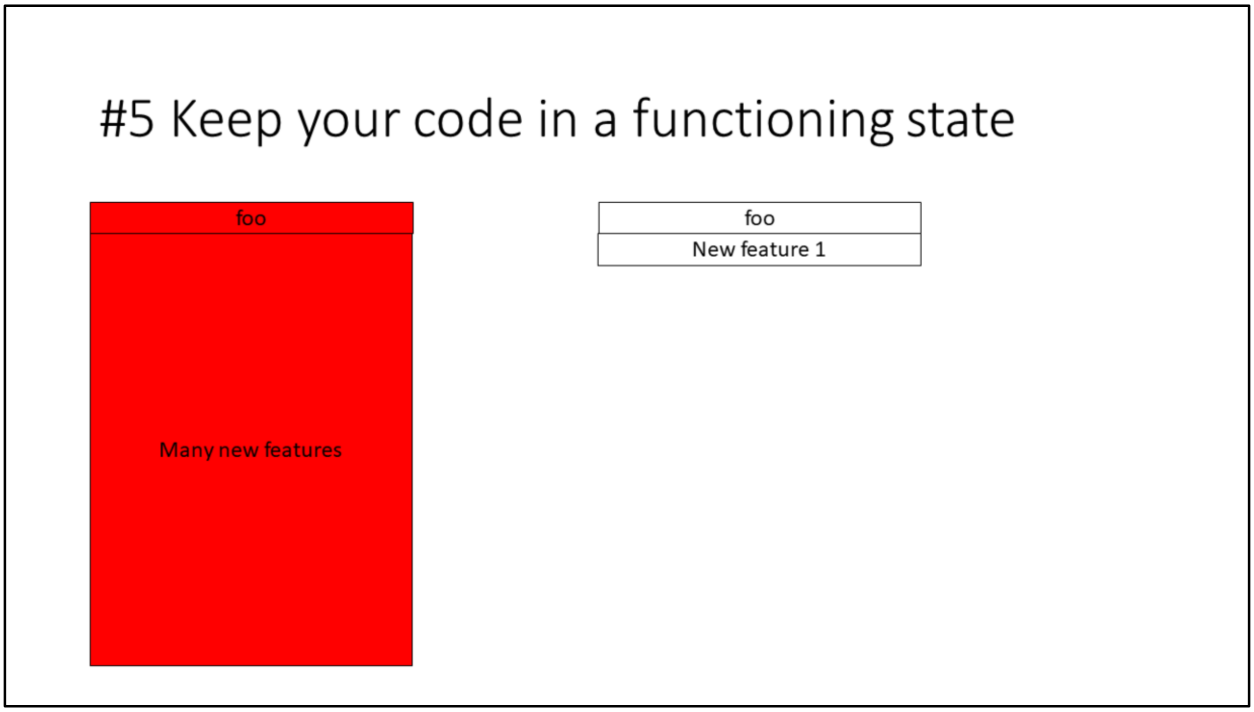
And it casts an error at you. What causes this error. Depending on the language used the error message might contain some helpful information but in theory it could be almost anywhere. Lets try again



#5 Keep your code in a functioning state

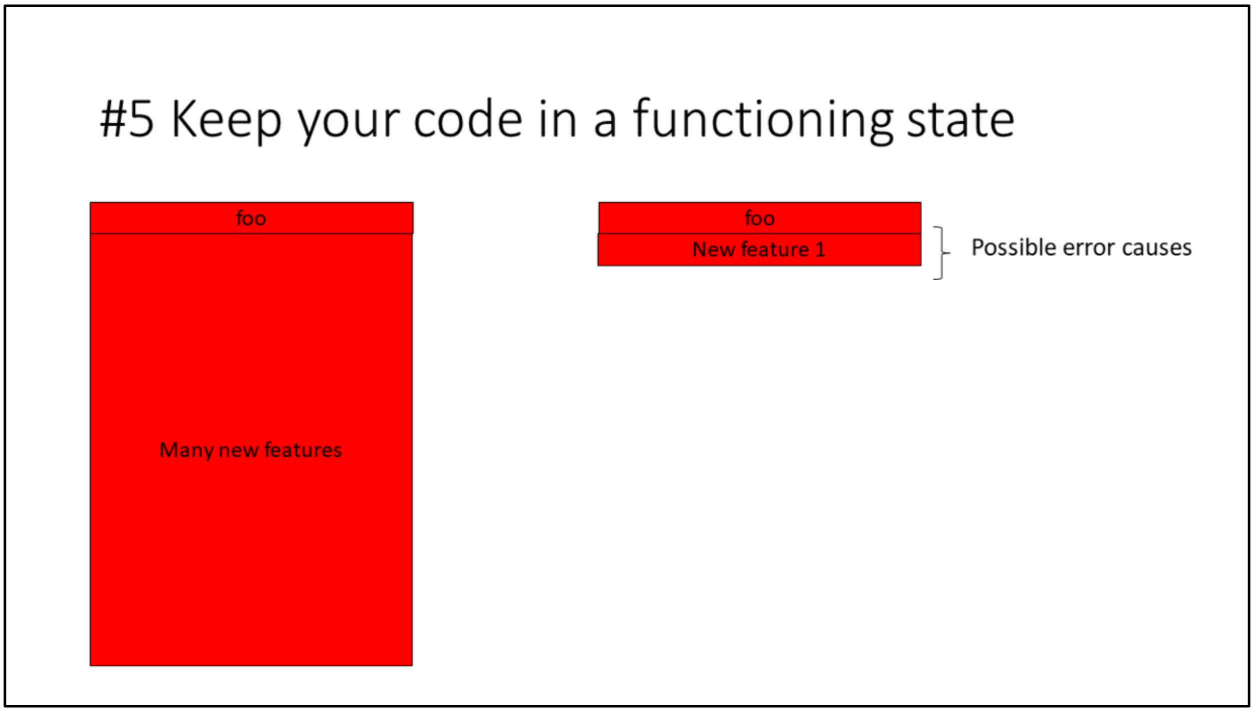
foo
Many new features
foo

This time you just add one feature



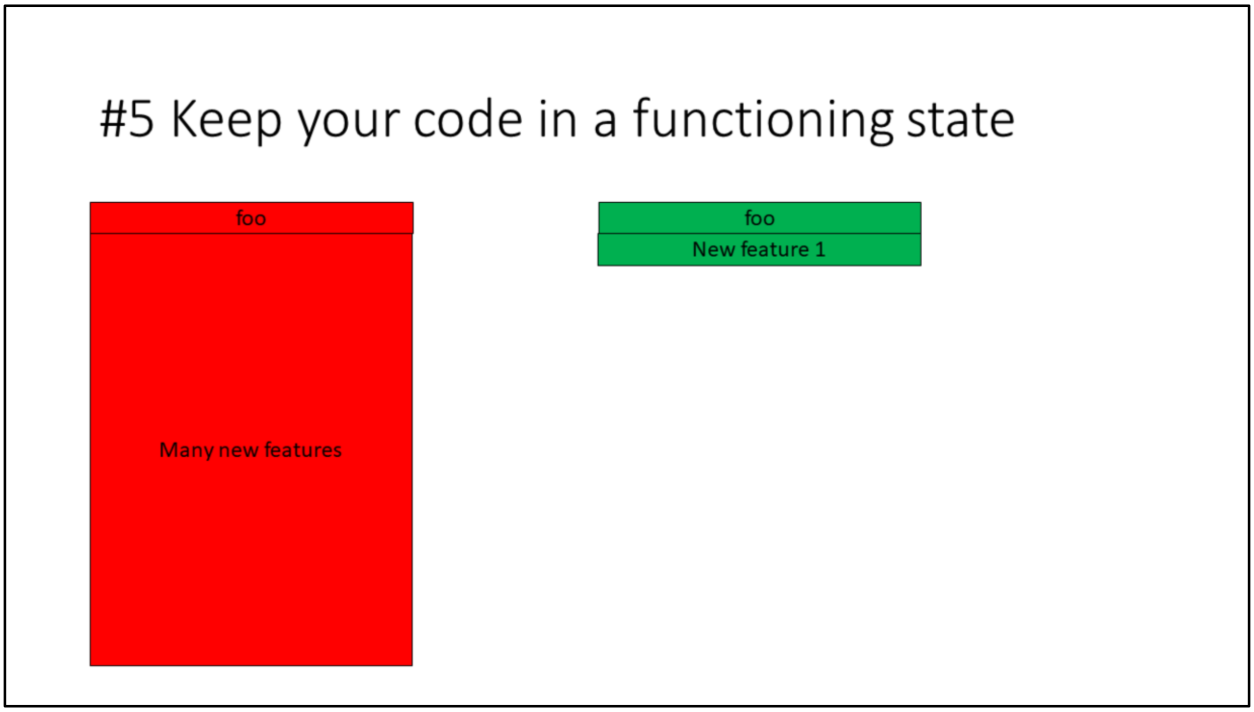
#5 Keep your code in a functioning state

Run it…



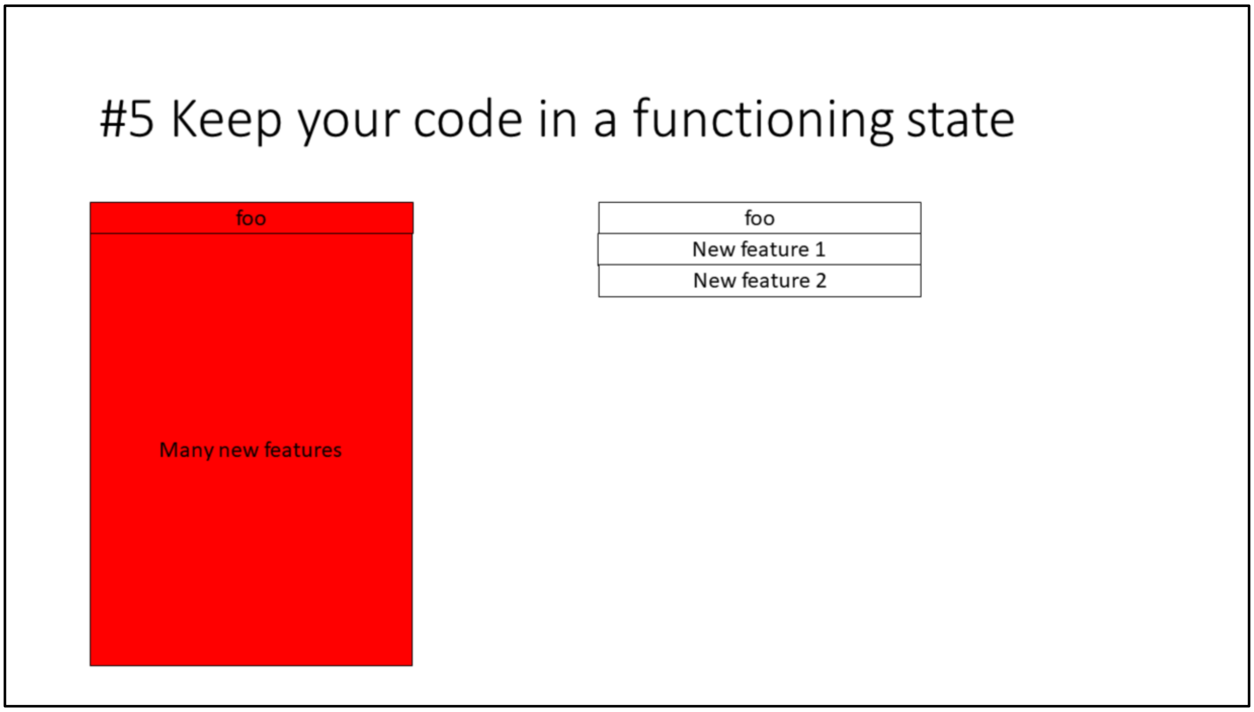
#5 Keep your code in a functioning state

And it casts an error again, but this time we have an easy time spotting the bug as we just added a few lines. And soon:



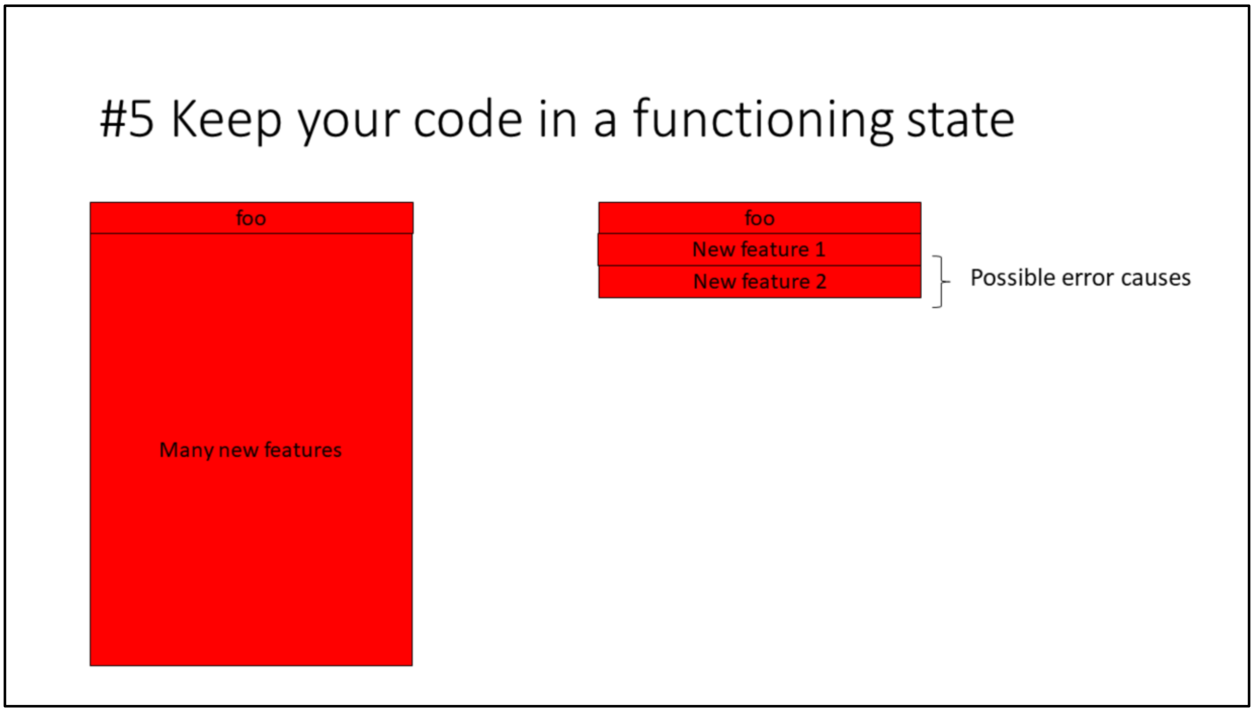
#5 Keep your code in a functioning state

We figure it out.



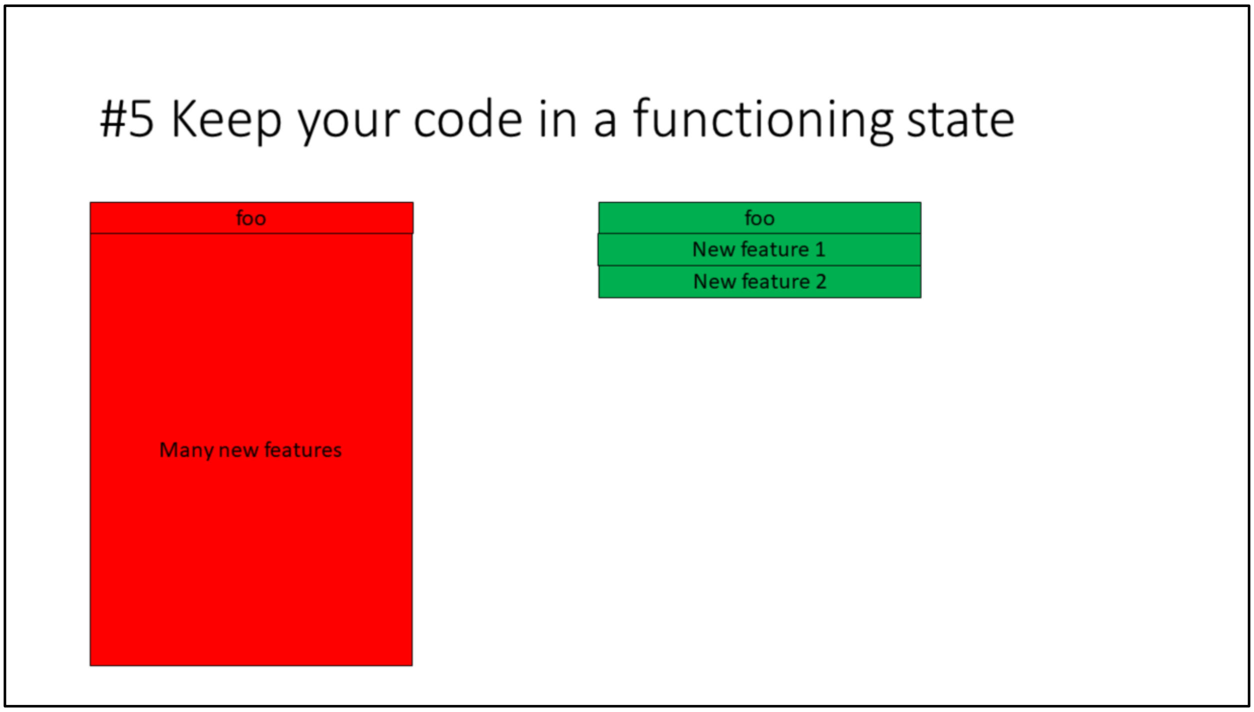
#5 Keep your code in a functioning state

And because we like this technique we just keep going like that

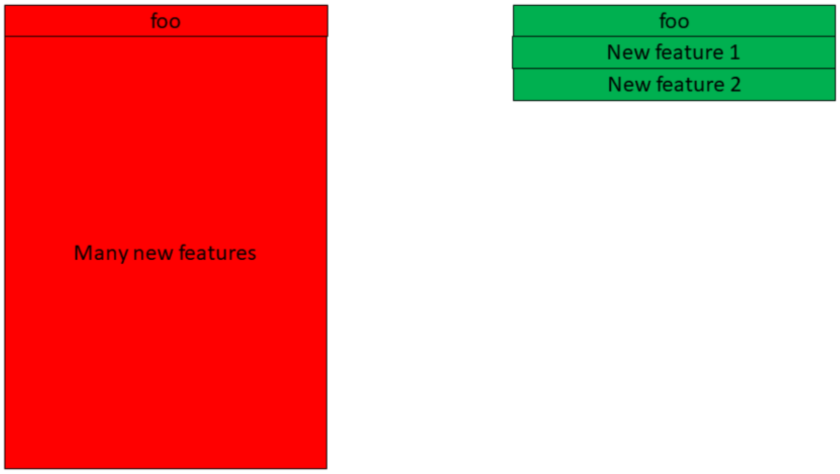


#5 Keep your code in a functioning state

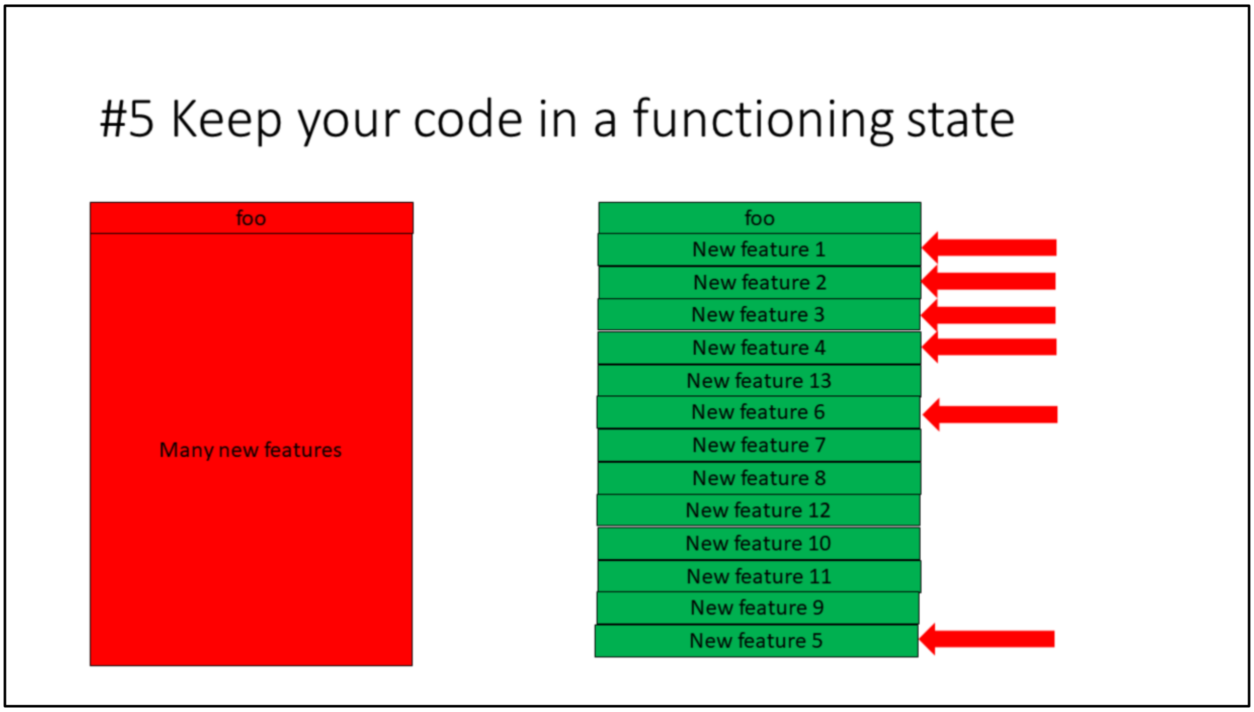
Another error. Nothing shocks us.



#5 Keep your code in a functioning state



We fix it.



#5 Keep your code in a functioning state

And soon we end up with this. Nice! By just adding one feature at a time we not only made debugging life easier, but we also kept our code very close to working condition most of the time. This view however is a little on sided. The approach on the left can be useful for didactic reasons as you will learn a lot from trying to untangle the mess you produced. The one on the right is safer. Lets suppose you want to send a copy of foo but

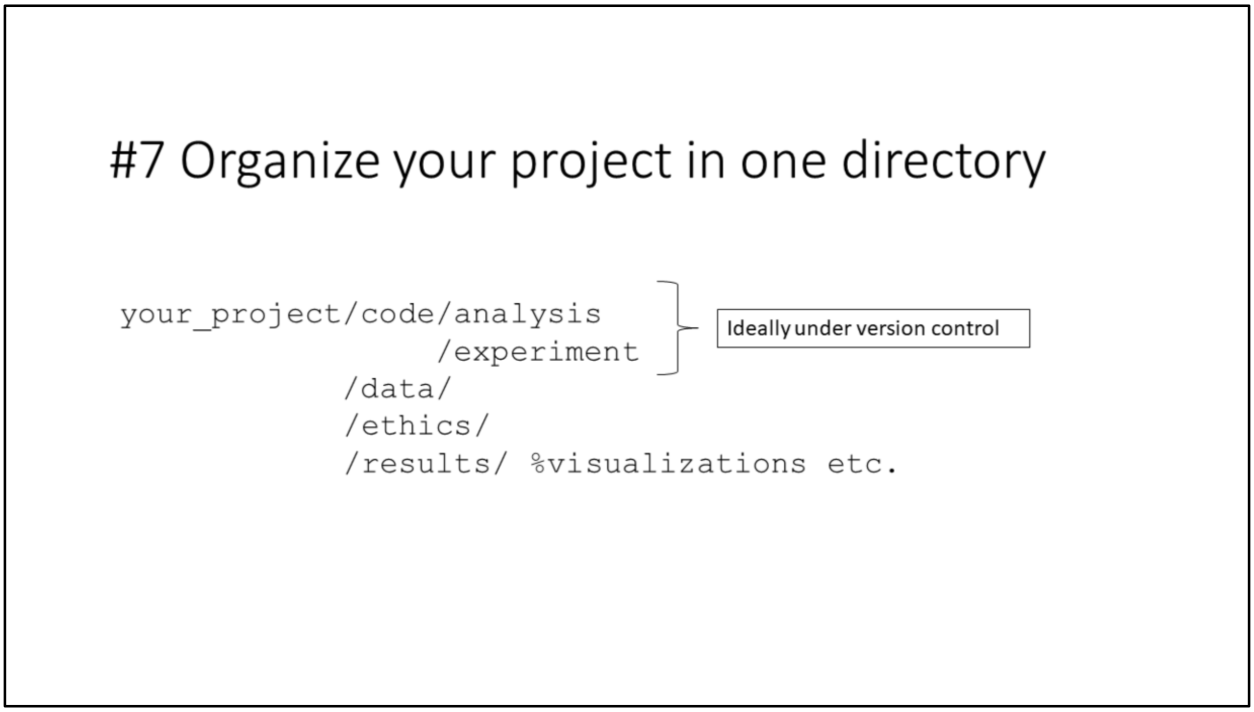
only with features 1-6 to a colleague. You could copy and paste it together but chances are you will mix something up as the features were moved out of their original order in the development process. If there only was a way to go back in time…

#6 git (and GitHub)

And luckily there is a way for code. It is called git. Git is version control software. It is used to track the evolution of your code. GitHub is a hosting service that provides a home on a server for your repository. GitHub is used to share your code with other people. Here‘s why you should use both:

#6 git (and GitHub) – why?

1. safety net
2. facilitates working from different machines
3. automatically chunk your work into neat little tasks
4. sharing becomes much easier

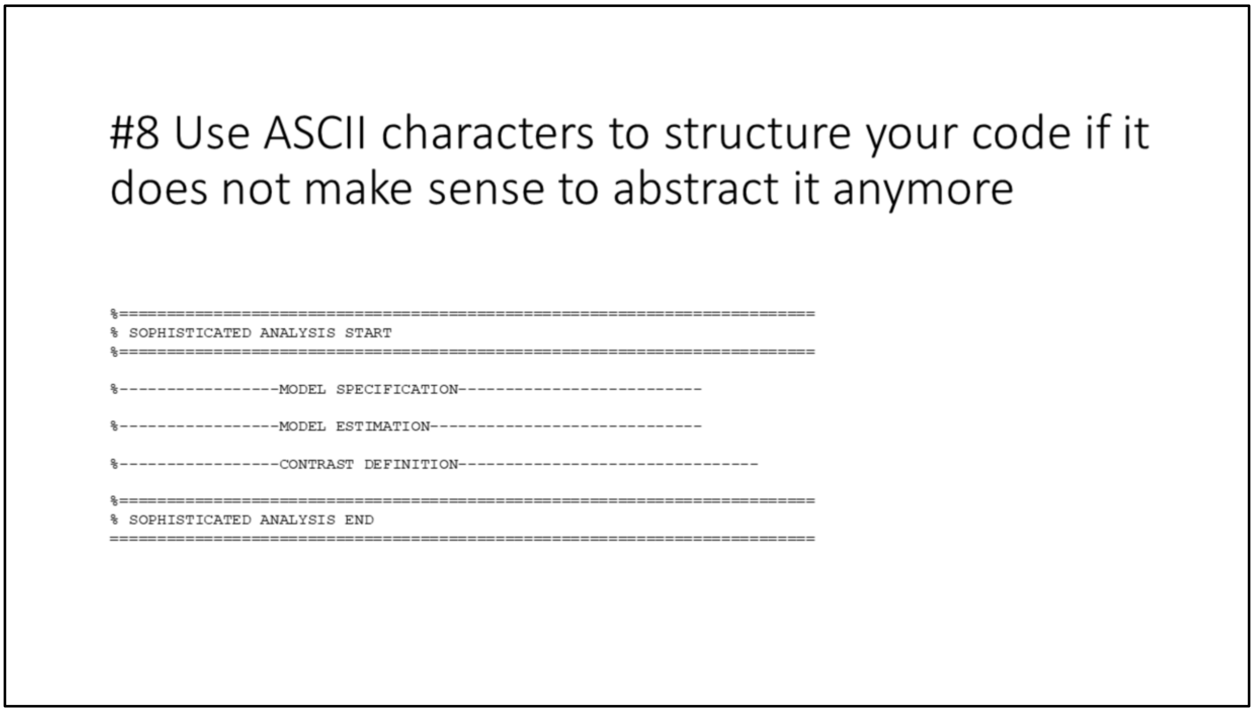


#7 Organize your project in one directory

```
your_project/code/analysis
                 /experiment
            /data/
            /ethics/
            /results/ %visualizations etc.
```

$\}$ Ideally under version control

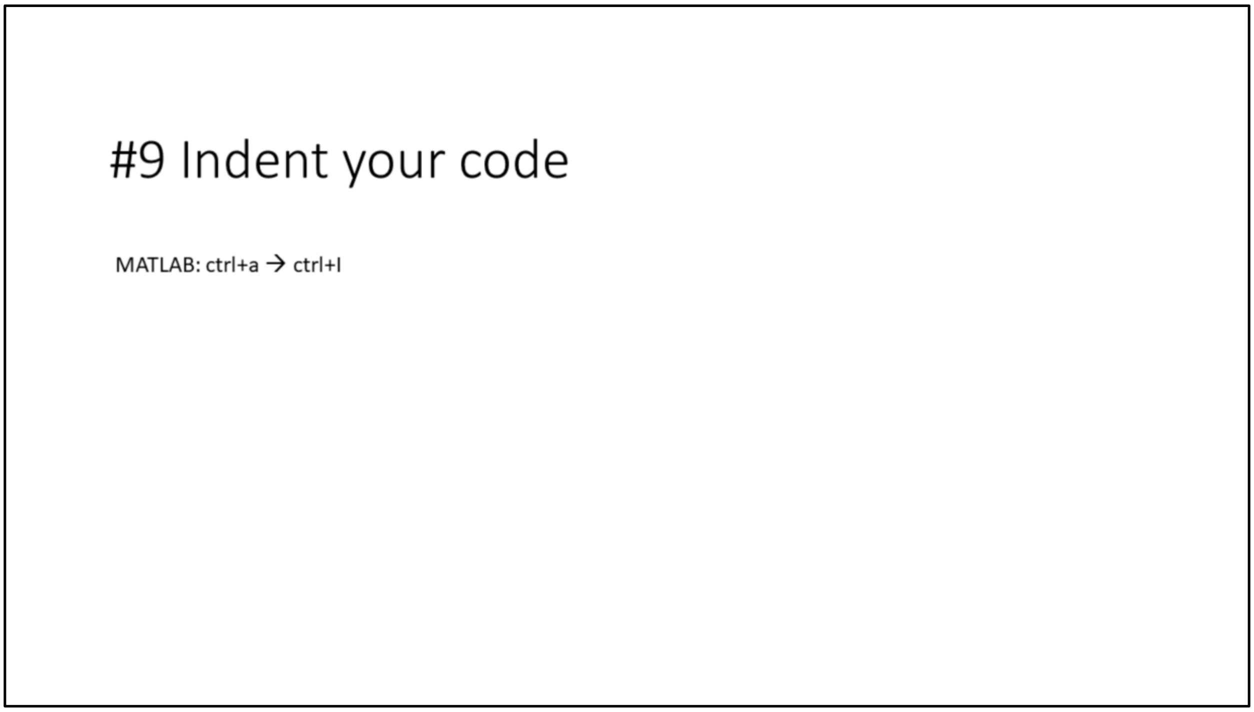
I would recommend to organize your project in one directory as opposed to having your code under documents and results under images for example.



#8 Use ASCII characters to structure your code if it does not make sense to abstract it anymore

```
%==========================================================================
% SOPHISTICATED ANALYSIS START
%==========================================================================

%-----------------MODEL SPECIFICATION--------------------------

%-----------------MODEL ESTIMATION-----------------------------

%-----------------CONTRAST DEFINITION--------------------------------

%==========================================================================
% SOPHISTICATED ANALYSIS END
===========================================================================
```

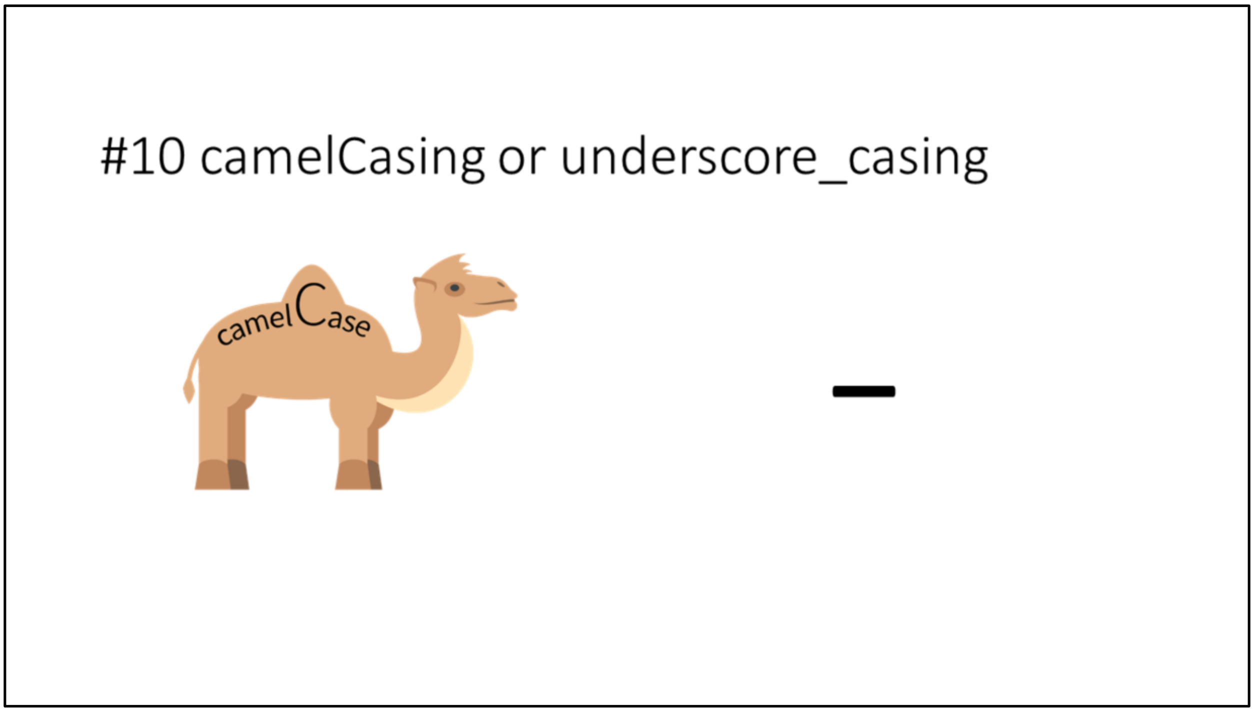
Sometimes functions can become long and it does not make sense to abstract them because many details need to be accessible from one file. For me this is the case for SPM scripts. Then you can use ASCII characters to help you get some structure.



#9 Indent your code

MATLAB: ctrl+a → ctrl+I

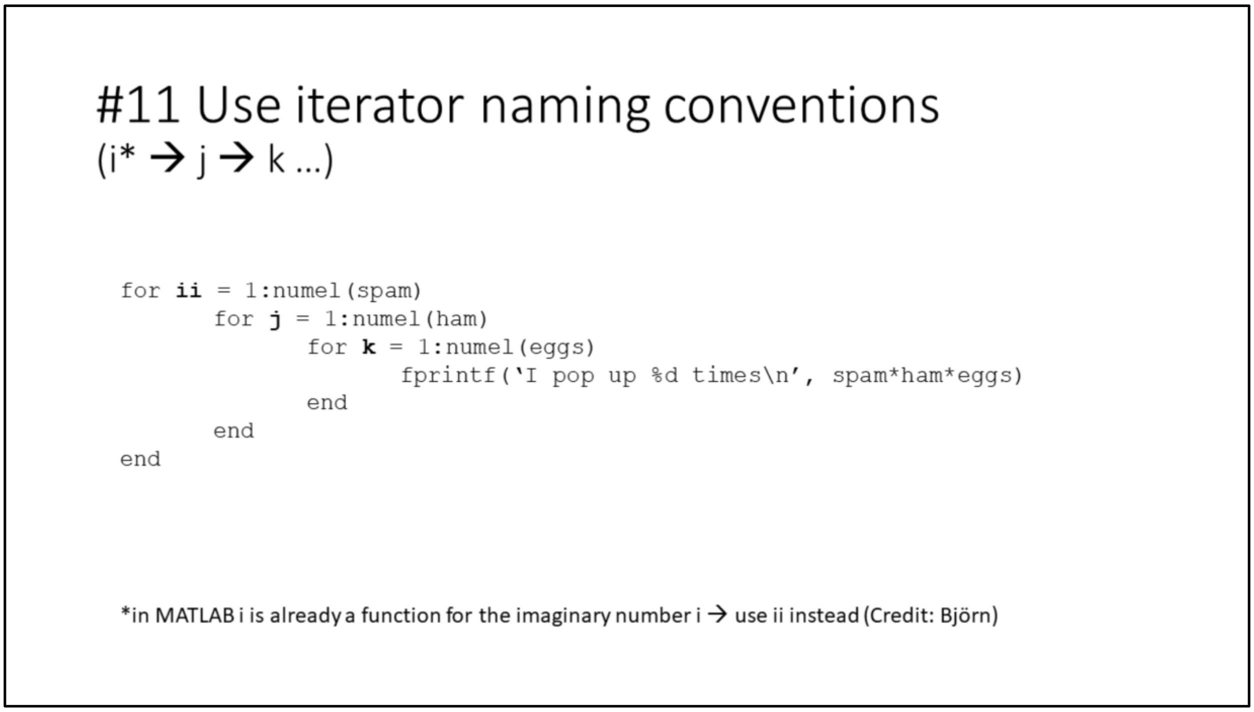
This is a no brainer. It improves readability and for Python this is even part of the syntax. The MATLAB editor even does this for you. If you hit ctrl a and then control I your code will be indented correctly. You can customize most other editors to do this.



#10 camelCasing or underscore_casing

_

There are mainly two very opposing conventions to name your variables and functions: camelCasing or underscore_casing Decide whether you want to use camelCasing or underscore_casing or
whatever other convention there is to name your functions and variables. You can also combine them and name your variables in underscore_case, your functions and methods in camelCase and global variables in all caps. But I would recommend to choose on for starters and then stick to it for a while. Sticking to it and being consistent is the important part.

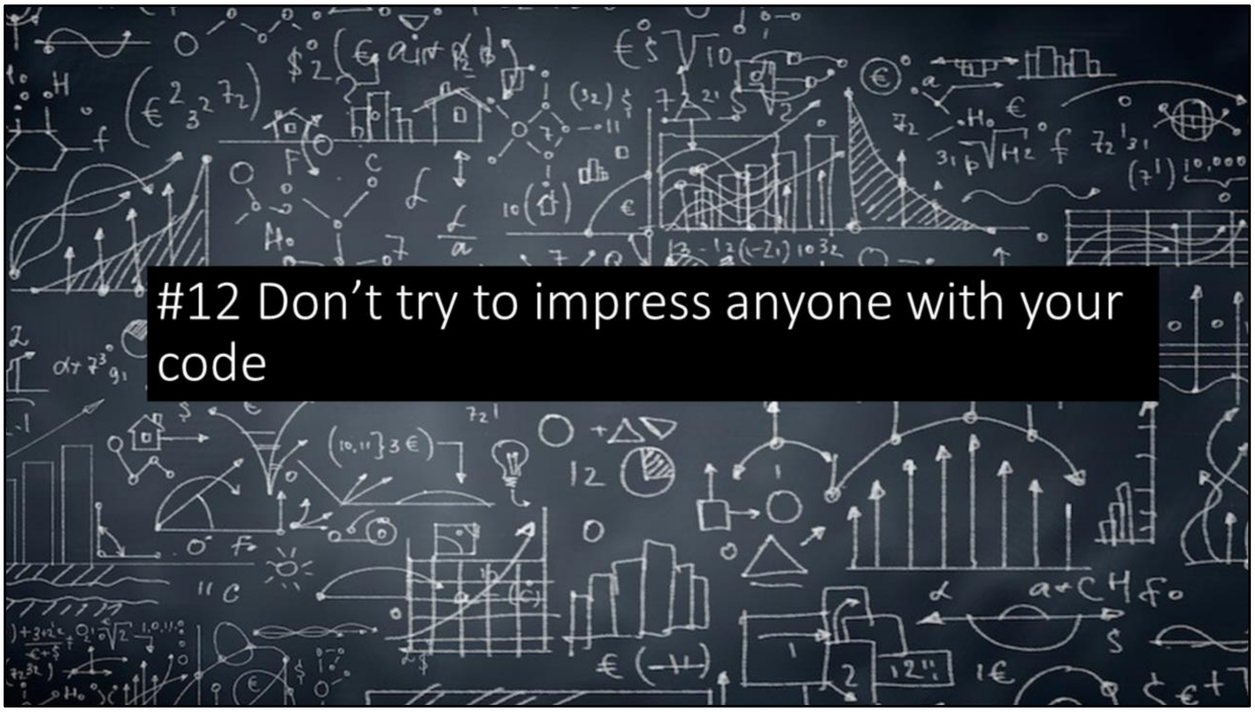


#11 Use iterator naming conventions
(i* → j → k …)

```
for ii = 1:numel(spam)
       for j = 1:numel(ham)
              for k = 1:numel(eggs)
                     fprintf('I pop up %d times\n', spam*ham*eggs)
              end
       end
end
```

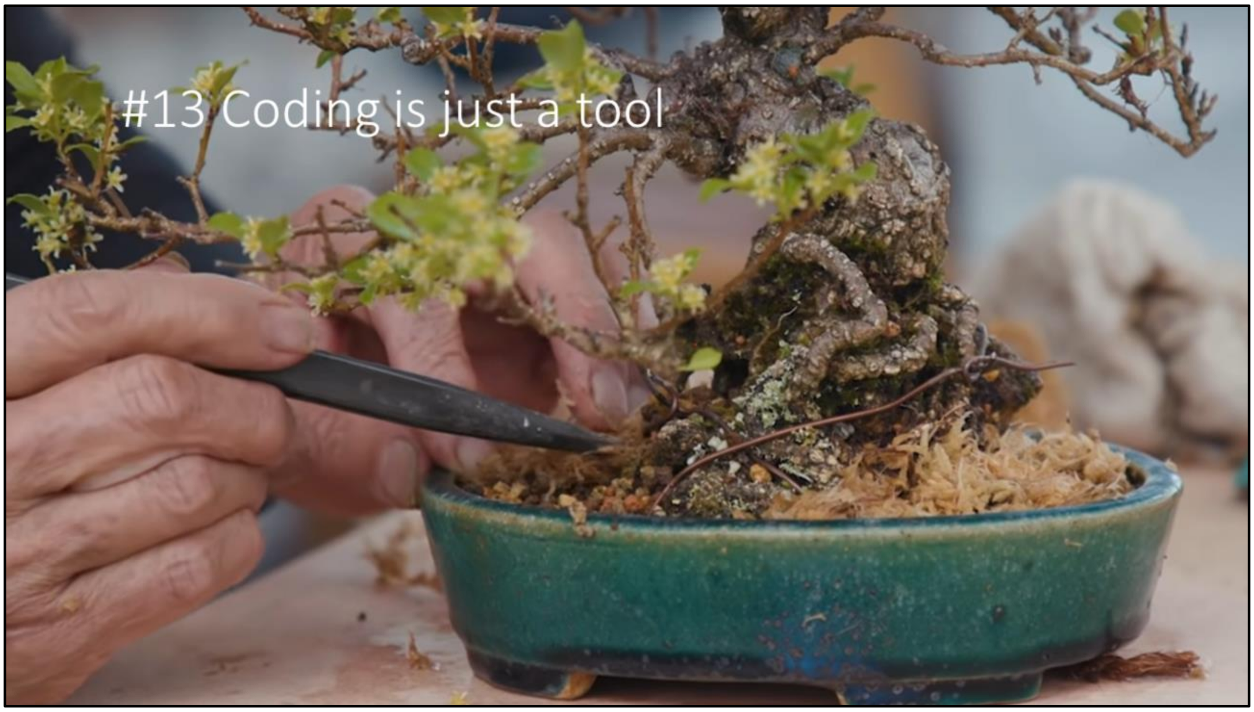
*in MATLAB i is already a function for the imaginary number i → use ii instead (Credit: Björn)

Same for iterators. Its just important to keep it consistent for yourself to reduce unnecessary decisions and for others to better understand your code. One way to do so is displayed here.



#12 Don’t try to impress anyone with your code

Okay, this is my second to last slide. Sometimes it can be tempting to show how good your coding has become. Try to resist the temptation. Instead of trying to show how many features of a programming language you know, try to keep your code as simple and basic as possible.



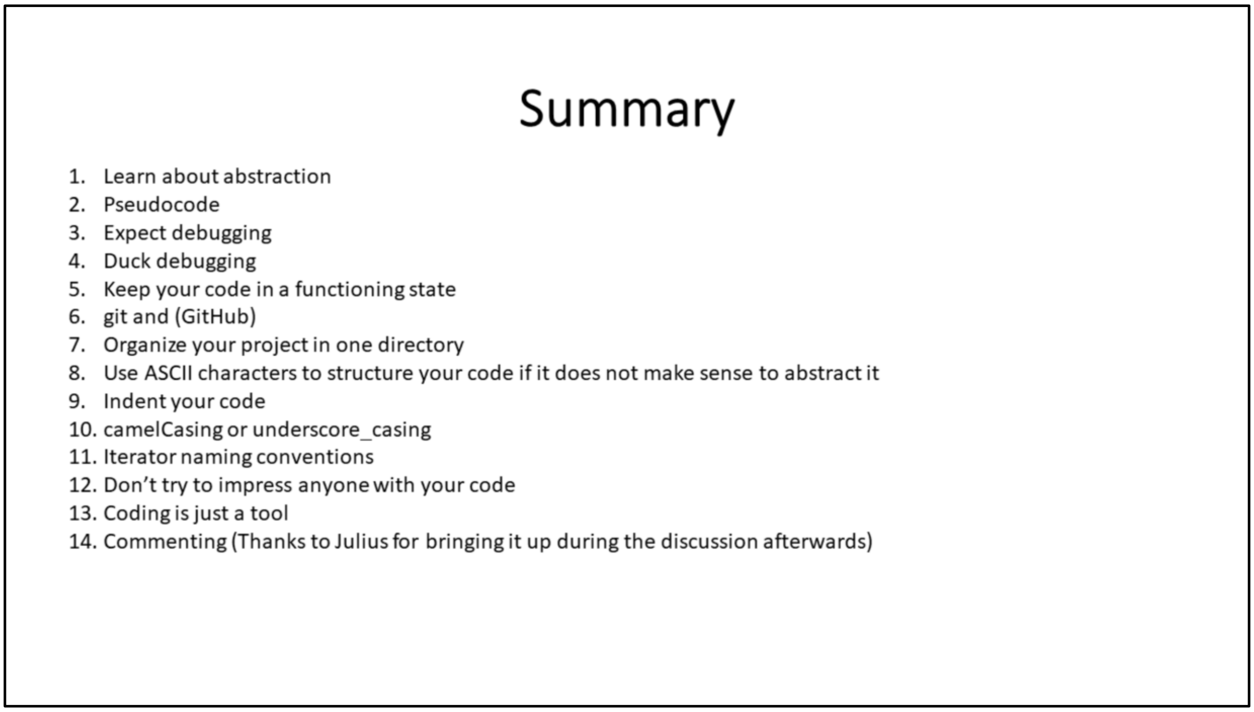
#13 Coding is just a tool

We have arrived at the final slide. In the end coding is a tool that helps us on our scientific endeavors. I think it is a good idea to acquire some theoretical knowledge about the tool, but in the end you improve by using it mindfully. Try your best to adhere to some strategies, or even better, create some of your own, but don't obsess about the tool itself. Just use it with purpose.

Alright that's it. Thank you very much for your attention.

#14 Commenting (Thanks Julius)

This is another important aspect of coding that Julius brought up during the discussion afterwards and that also nicely ties in with sharing code. The code we write in a given period of time is heavily influenced by the assumptions we employ during that period. However, those assumptions might change over time. Frankly speaking, this could result in something like: “I have no idea what I tried to accomplish with those lines of code (?!)”. To prevent this, try to account for those implicit assumptions with your comments so that other people and future instances of yourself, who might not have immediate access to those assumptions, are still able to understand your code.



Summary

1. Learn about abstraction
2. Pseudocode
3. Expect debugging
4. Duck debugging
5. Keep your code in a functioning state
6. git and (GitHub)
7. Organize your project in one directory
8. Use ASCII characters to structure your code if it does not make sense to abstract it
9. Indent your code
10. camelCasing or underscore_casing
11. Iterator naming conventions
12. Don’t try to impress anyone with your code
13. Coding is just a tool
14. Commenting (Thanks to Julius for bringing it up during the discussion afterwards)