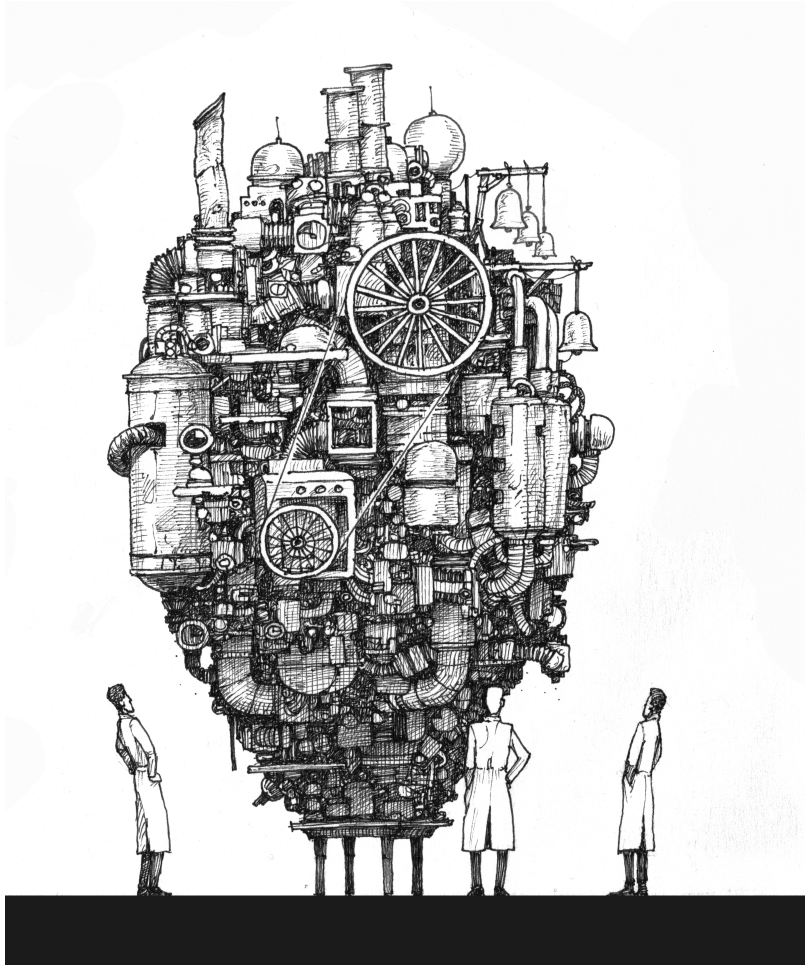
II. Code reusability

Some questions

- What does "modular code development" mean for you?
- What best practices can you recommend to arrive at well structured, reusable code in your favourite programming language?
- What do you know now about programming that you wish somebody told you earlier?



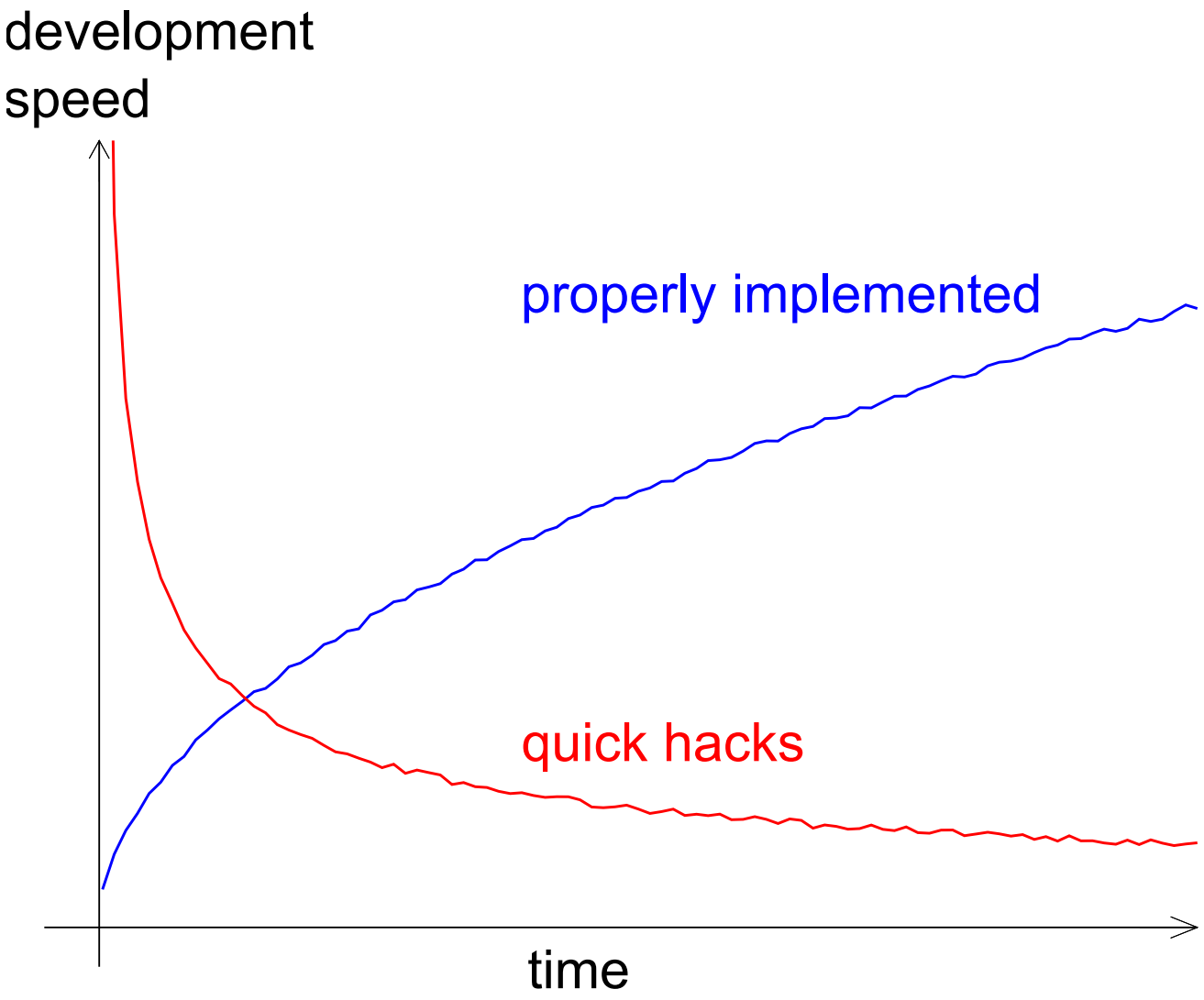
Additional questions

- Do you design a new code project on paper before coding? Discuss pros and cons.
- Do you build your code top-down or bottom-up? Discuss pros and cons.
- Would you prefer your code to be 2x slower if it was easier to read it?

The tar pit

- Over time, software tends to become harder and harder to reason about
- Small changes become harder to implement
- Bugs start appearing in unexpected places
- More time is spent debugging than developing
- Complexity strangles development because it does not scale well

Do you recognize this? Can you give an example?



Do you agree and what do you think "properly" means?

Modularity and composition

- Build complex behaviour from simple components
- We can reason about the components and the composite
- Composition is key to managing complexity
- Modularity does not imply simplicity, but is enabled by it

Code reusability: some guidelines

- Separate code and data: data is specific, code need not be
  - consider using a config file for project-specific (meta)data in a interoperable datatype (`.json` , `yaml`)
  - use relative paths
  - but DO hard-code unchanging variables, e.g. `gravity = 9.80665` , **once**.

Code reusability: some guidelines

- Do One Thing (and do it well)
  - One function for one purpose
  - One class for one purpose
  - One script for one purpose (no copy-pasting to recycle it!)

Code reusability: some guidelines

- Don't Repeat Yourself (DRY): use functions
  - Write routines in functions, i.e., code you reuse often
  - Identify potential functions by action: functions perform tasks (e.g. sorting, plotting, saving a file, transform data...)
  - Single source of truth reduces maintenance and makes debugging easier

Note, in MATLAB a function can only be called from a script if it bears it's name

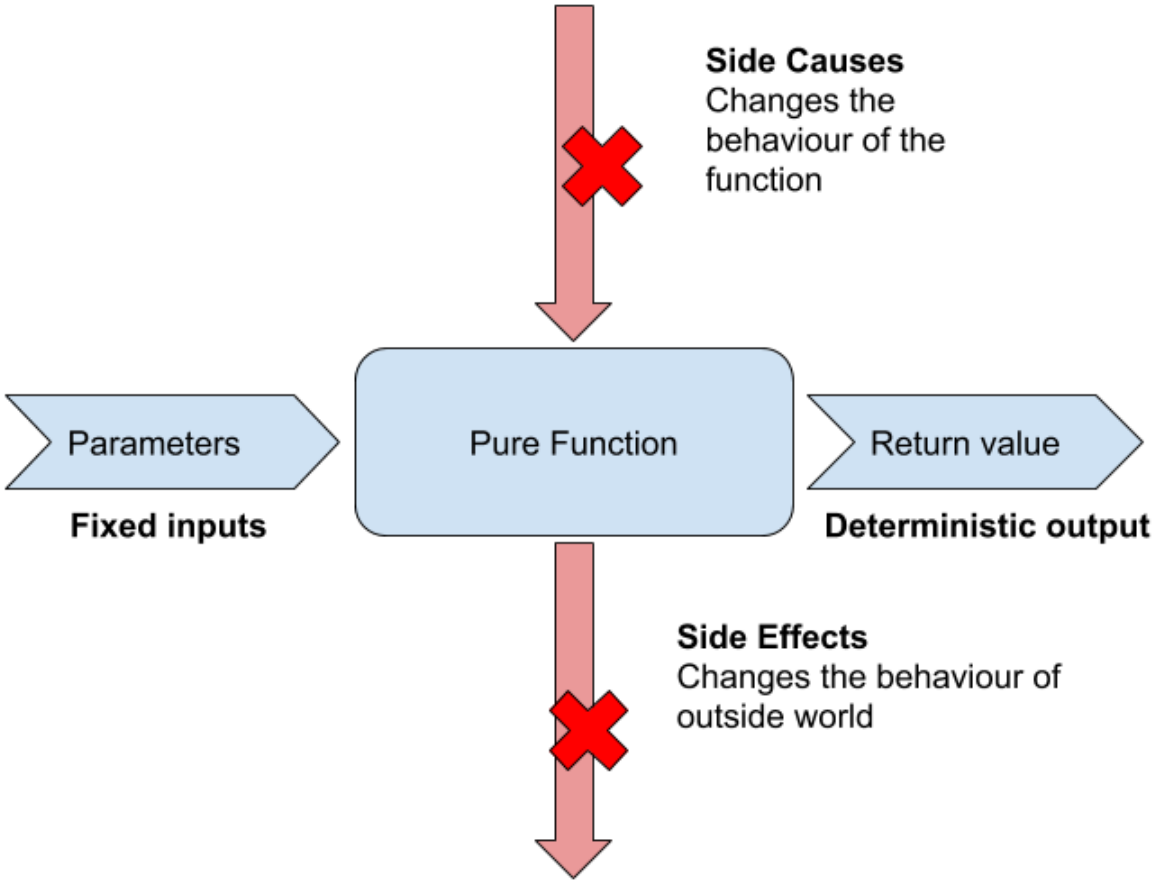
Purity

- Pure functions have no notion of state, i.e. they take input values and return values.
- Given the same input, a pure function *always* returns the same value

Examples

- Mathematical functions
  $f(x, y) = x - x^2 + x^3 + y^2 + xy$
- Unix shell

```
cat somefile | grep somestring | sort | uniq | ..
```



Pure functions are easy to

- Test (*we will address testing later*)
- Understand
- Reuse
- Simplify
- Parallelize
- Optimize
- Compose

Example: pure vs. stateful

Stateful code

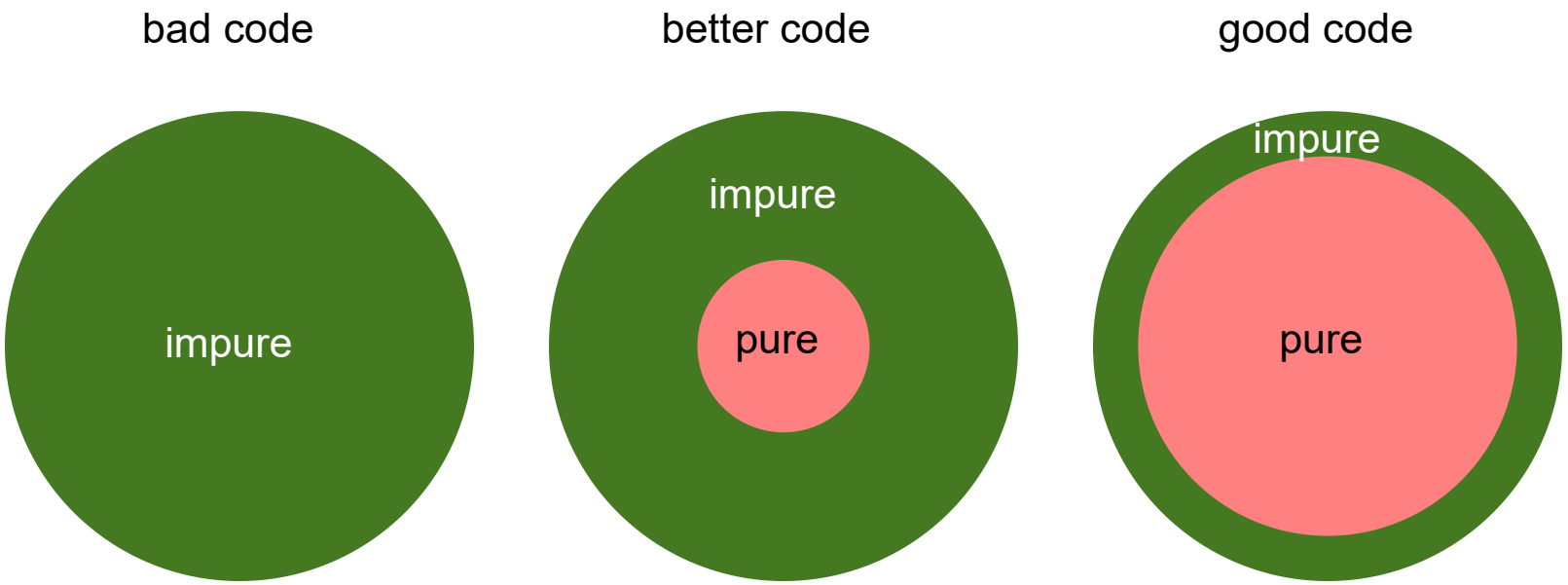
```
f_to_c_offset = 32.0
f_to_c_factor = 0.555555555
temp_c = 0.0

def fahrenheit_to_celsius_bad(temp_f):
    global temp_c
    temp_c = (temp_f - f_to_c_offset) * f_to_c_factor

fahrenheit_to_celsius_bad(temp_f=100.0)
print(temp_c)
```

Pure - no side effects

```
def fahrenheit_to_celsius(temp_f):
    temp_c = (temp_f - 32.0) * (5.0/9.0)
    return temp_c
temp_c = fahrenheit_to_celsius(temp_f=100.0)
print(temp_c)
```

Recommendations

- I/O is impure (reading/writing)
- Keep I/O on the outside and connected
- Keep the inside of your code pure/stateless



Functions do not necessarily make code shorter (at first)! Compare:

```
indexATG = [n for n,i in enumerate(myList) if i == 'ATG']
indexAAG = [n for n,i in enumerate(myList) if i == 'AAG']
```

with

```
def indexString(inputList: List, z: str) -> List:
  """Find index of string in list
  """
        zIndex = [n for n,i in enumerate(li) if i == z]
        return zIndex

indexATG = indexString(myList,'ATG')
indexAAG = indexString(myList,'AAG')
```

But, you can now reuse your function elsewhere, made it general (**one instance!**), and have added additional documentation (function name, docstring, typehinting)

Divide and conquer

How to approach your code

- Split up the code into functional parts
- Construct your program from these parts
  - functions
  - modules
  - packages (Python) or libraries (C or C++ or Fortran)

Your turn: visualize your code!

Choose:

- Make a screenshot, process it in paint, powerpoint, or your favorite editor;
- Copy paste your code to a text editor, and use markers.

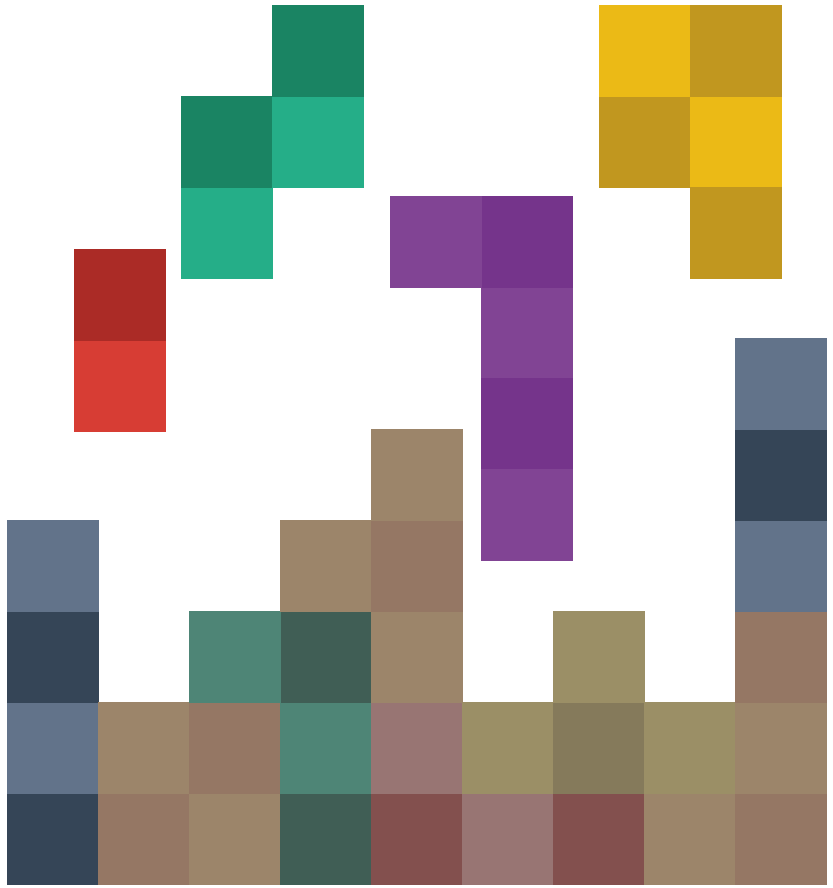
The objective is for you to 'see' your code!

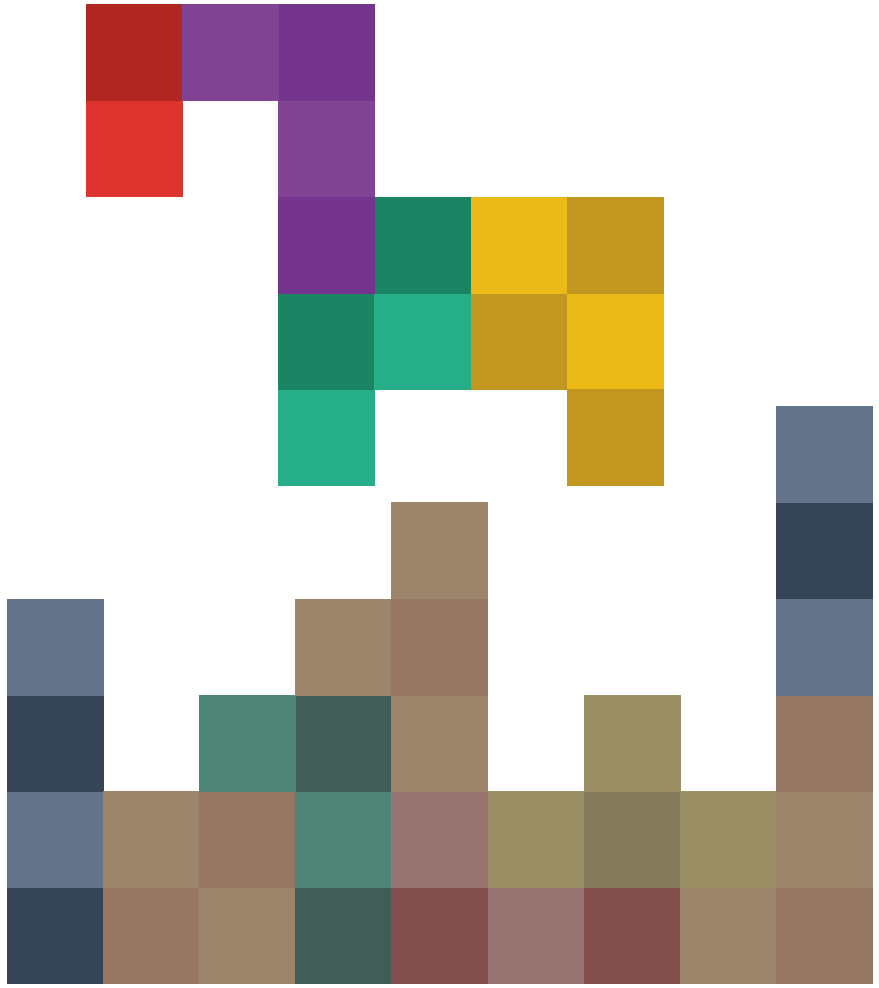
- Yellow denotes scripted, unstructured code *(basic, sequential lines of instructions)*
- Purple denotes functions or other structured code *(e.g. for-loops, conditionals, etc.)*
- Green denotes comments (or comment blocks)

Again, make notes in your code (`#TODO` !) if you see:

- **Scripted code**: this can be a function
- **Structured code**: this should be re-structured

How does your code look?





Functions, functions, functions

- Build your code from functions
- Break your code down to more functions
  - if you have too many levels of indentation
  - if a function gets too long
  - if a function does more than one thing
  - if you find it hard to name a function
  - if you find it hard to write tests for a function

Simplicity and clarity before elegance before efficiency

Avoid premature optimization

- Do not optimize
- If you have to optimize, do not optimize
- If you have to optimize, measure, do not guess

Simple is better than complex

> Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it. (Brian W. Kernighan)

```
// Dear programmer:
// When I wrote this code, only god and
// I knew how it worked.
// Now, only god knows it!
//
// Therefore, if you are trying to optimize
// this routine and it fails (most surely),
// please increase this counter as a
// warning for the next person:
//
// total_hours_wasted_here = 254
//

```

Your turn: make a function

You have visualized your code. Use your findings to improve it!

- **Preferably**: take scripted code and turn it into a function, *or* split an existing function into two or more functions, *or* generalize multiple similar functions (DRY!)
- If there is no function to work on: try and address the readability of your code.

However: for future exercises you will need at least one function, preferably with parameters, in your code!

Recommendations

- Divide and isolate
- Compose your code out of pure functions
- Keep it simple
- Prefer immutable data structures (*immutable* means that the data value cannot be modified)
- Think about the people reading your code, do not "Write Unmaintainable Code"