Programming for psychologists

Practical 1.1: Mindset

Matthias Nau

Programming mindset 1
Peanut butter jam sandwich

Peanut butter jam sandwich

=

+

+

Is that it?

Peanut butter jam sandwich

Initial ingredients

Now you write the full recipe!

Imagine instructing someone who has never made a PB&J sandwich.

Your goal is to write clear instructions that they can follow to make a sandwich without any confusion.

Around 6 groups. Pen and paper, or laptop. 15 Minutes.

Peanut butter jam sandwich

Full video: https://www.youtube.com/watch?v=cDA3_5982h8

Imperative instructions

Example recipe

Pros: Good control & clarity, efficient for simple tasks, intuitive.
Cons: Repetetive, error-prone, limited re-usability, difficult to scale.

- Put the two slices of bread into the toaster and toast them for 1 minute.
- Place the two toasted slices of bread flat on a plate, side by side.
- Unscrew the lid of the peanut butter jar and set it aside.
- Use a butter knife to scoop a generous amount of peanut butter from the opened jar.
- Spread the peanut butter evenly on one slice of bread, covering its surface.
- Unscrew the lid of the jam jar and set it aside.
- Use the butter knife to scoop a generous amount of jam from the opened jar.
- Spread the jam evenly on the other slice of bread, covering its surface
- Place the slice of bread with peanut butter on top of the slice with jam, aligning the edges.

Imperative instructions

A list of **steps and procedures** to be **followed in order**.

Imperative instructions “**mutate state**”, meaning that they change the state of things often irreversibly.

If the steps are inaccurate, or followed in the wrong order, you **burn down your kitchen** instead of making a sandwich.

In programming, imperative instructions are **extremely useful** as long as you **keep their risks in mind**.

Declarative instructions

An alternative receipe

Goal: **peanut_jam_sandwich**

- **peanut_jam_sandwich** is combine_slices(**peanut_slice**, **jam_slice**).
- **peanut_slice** is **toasted_slice** plus **peanut_butter**
- **peanut_butter** is scoop(**butter_knife**, **peanut_jar**)
- **jam_slice** is a **toasted_slice** plus **jam**
- **Jam** is scoop(**butter_knife**, **jam_jar**)
- **toasted_slice** is toast(**bread_slice**, **60**)

- combine_slices(input1, input2)
  # A function that concatenates two inputs and gently presses them together until delicious.
- toast(input1, input2)
  # Function that toasts input1 for time specified by input2 in sec.
- scoop(input1, input2)
  # A function that uses input1 to scoop out content of input2.

Declarative instructions

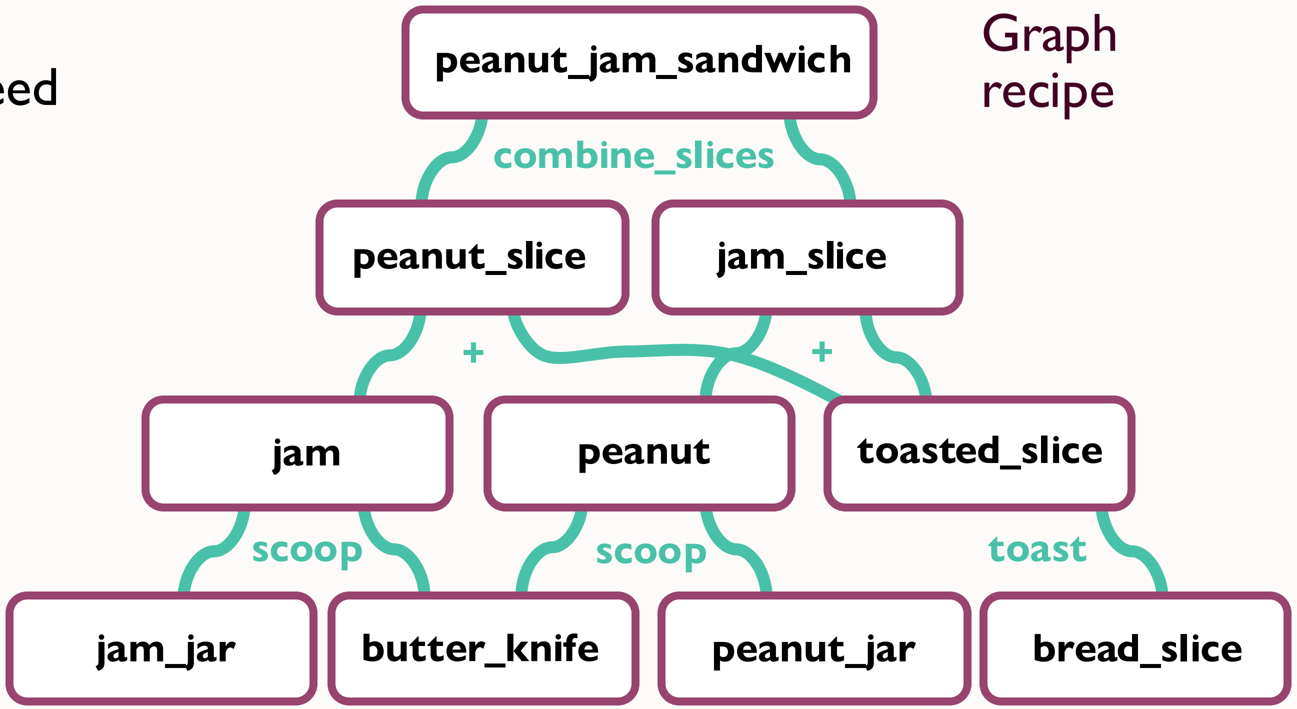
A list of **outcomes** to be achieved and **their relationship**. (e.g., bread_slice is needed for toasted_slice).

Declarative instructions **do not** necessarily need to be **followed in order**.

If something is inaccurate or not defined, your **code crashes**. You won't get a sandwich, but your **kitchen is safe**.

Declarative instructions can be represented as a graph, which helps to break down tasks into simpler steps.

Peanut butter jam sandwich



Top-down design

Breaking down **complex problems** into **simple parts**.

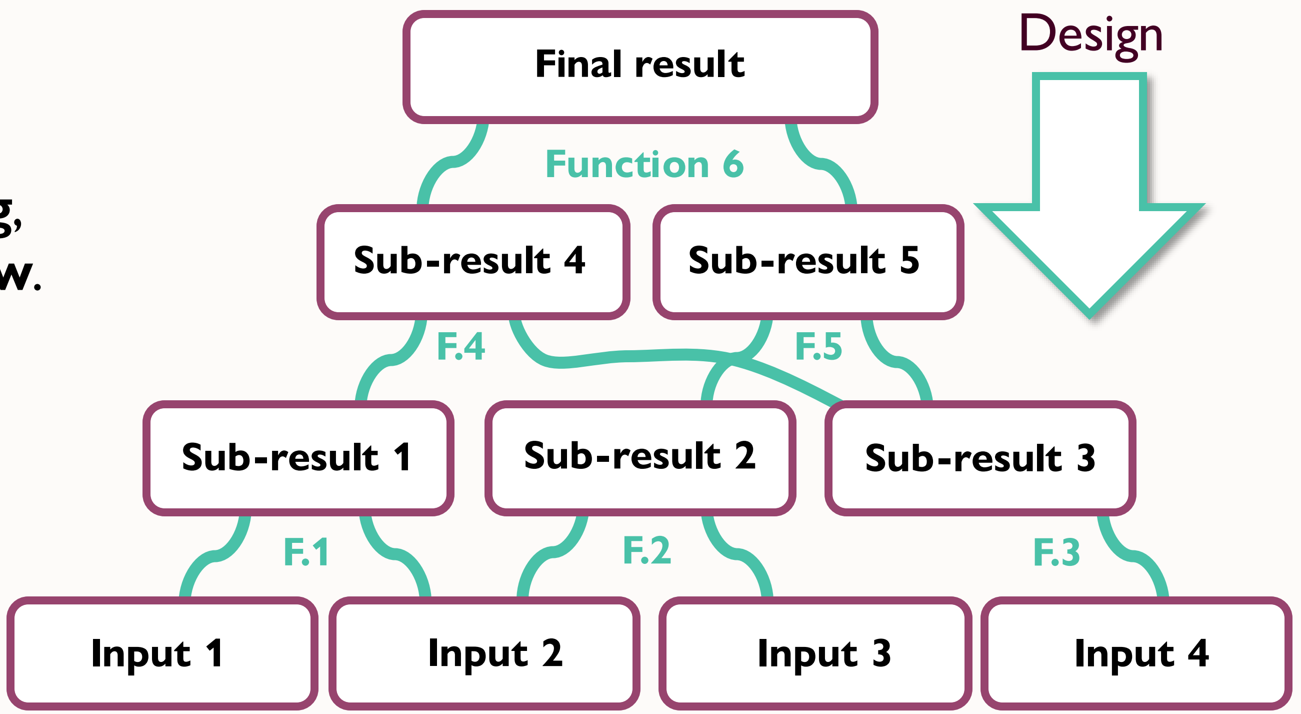
One of the **most powerful concepts** for **managing complexity**.

The **first step** of programming is **not coding**, but **thinking about what to code and how**.

First, think about the desired **sub-results** (outcomes) and their **relationships**!

You are not expected to know how to do this, but just keep this idea in mind as you learn how to program.

A problem you want to solve



Bottom-up implementation

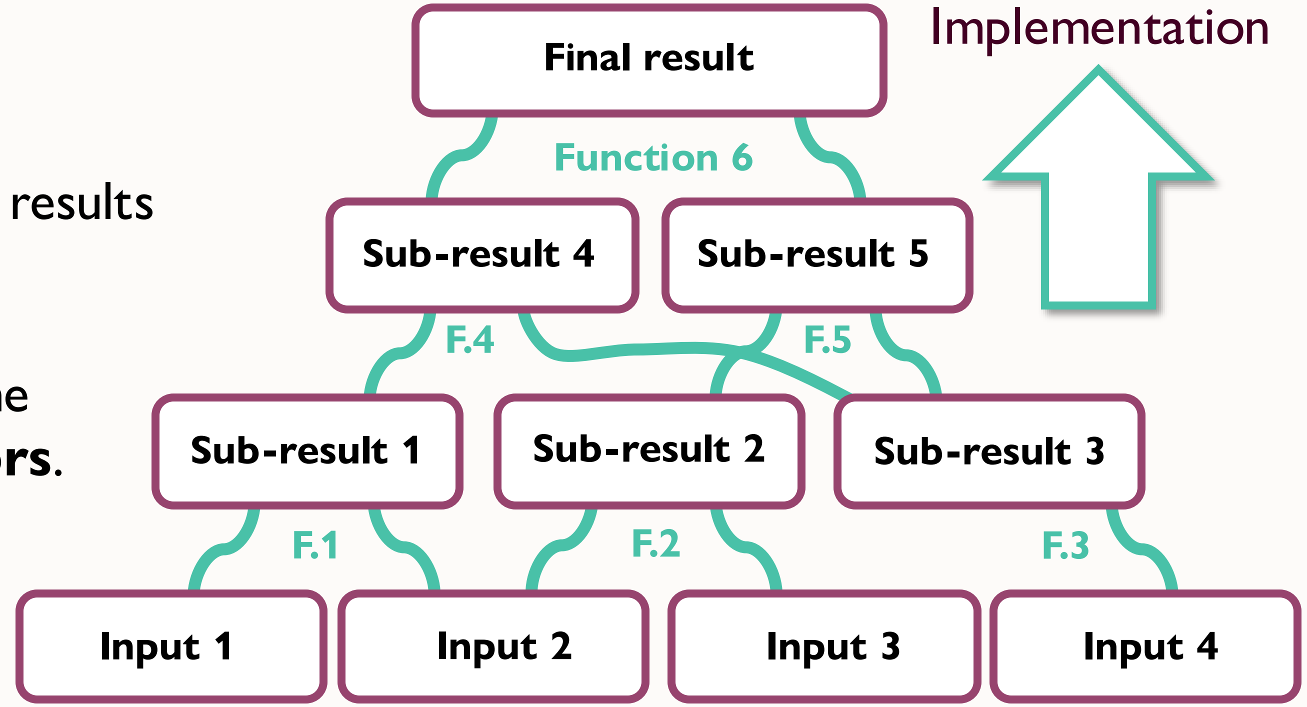
Now that the **problem is broken down**, begin to **solve each sub-task** starting at the **lowest level** (e.g., getting data into the right format).

Test each step extensively!
For example, create ”fake data” for which the results are known, and check if your result matches.

Only once you’ve past all tests, move on to the next level. This approach will **minimize errors**.

You are not expected to know how to do this, but just keep this idea in mind as you learn how to program.

A problem you want to solve



Take home messages - Part I

- There are almost always **multiple ways** to reach a goal (i.e. multiple recipes), and imperative and declarative **strategies often mix**.

- Solving **complex problems** starts with breaking them down into **simpler parts,** (**top-down**) and then implementing solutions **bottom-up**.

- Code design takes **practice**, but it's a **key skill** that (I believe) you want to work towards.

- What makes a **good programmer** is knowing **what to code** in addition to **how**.

⇧

Some more on this after the break!

Programming mindset 2
What makes a good programmer?

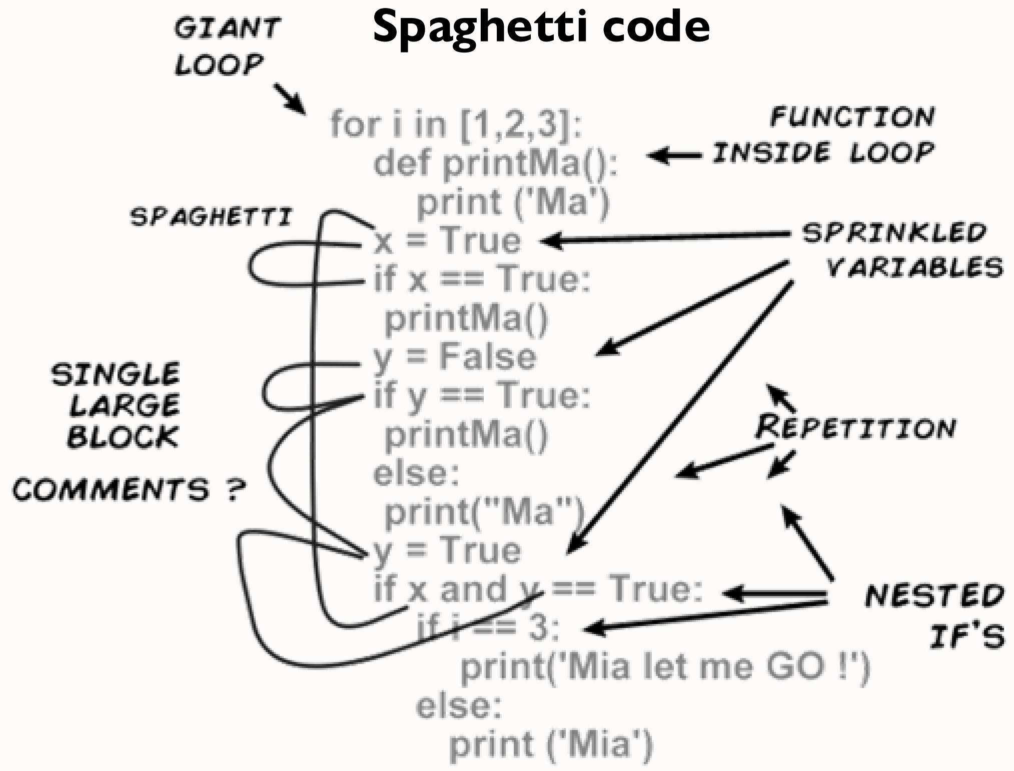
Keep the goal in mind

Define your goal before you code and **remind yourself** of it often.

Ideally, think through your **top-down design**. If that's too much, at least define final result.

If the goal is not clear, you risk writing "**spaghetti code**" that keeps growing in size, complexity, and **error rate**.

Our goal was making a PB&J sandwich.


Spaghetti code

```
for i in [1,2,3]:
  def printMa():
    print ('Ma')
  x = True
  if x == True:
   printMa()
  y = False
  if y == True:
   printMa()
  else:
   print("Ma")
  y = True
  if x and y == True:
    if i == 3:
      print('Mia let me GO !')
    else:
     print ('Mia')
```

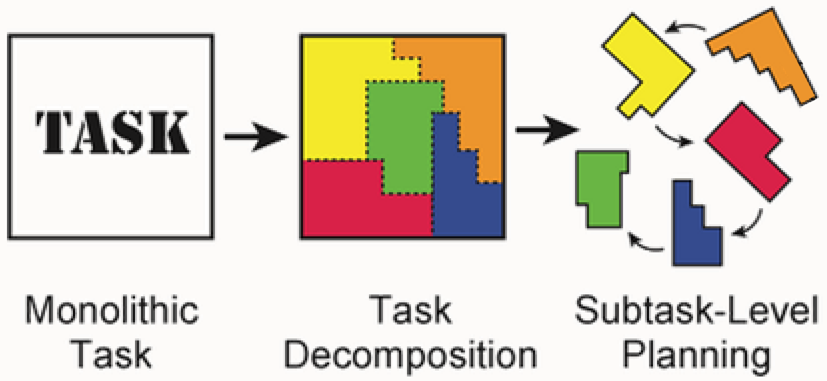
Think about task decomposition

Define sub-goals that your code should accomplish in pursuit of the final result.

Smaller tasks are more manageable, and errors are easier to fix in shorter blocks of code.

Each sub-task can then be **accomplished by a function** (we will talk about functions again later)

Each function should have one purpose (**single-responsibility principle**), but similar tasks can often be grouped together (**cohesion principle**), or even be solved by the same function depending on its **abstraction.**



Correa et al. 2023

We decomposed the goal of making a sandwich into many smaller steps.

Remember that there is no spoon

Think for yourself! What do you want to build, how will you build it, and why?

There is not one right way, there are many. You are the creator who defines what your code does and how it does it.

Be proud of your solution and own it. Owning it includes being able to explain what your code does and why, and to correct mistakes.

Don't follow recipes but write them.
Make the sandwich that you want to make.

"There is no spoon" (The Matrix)

There might be more elegant code than yours, but the only <u>wrong</u> code is the one that does not do what you think it does!

Avoiding stupidity is easier than seeking brilliance

If you can't see **the right way forward**, figure out how you would **mess things up** - then avoid those things!

For example, you may not know what a good variable name is, but naming two variables the same is intuitively a bad idea (right?!).

Be your own adversary when programming, and you will already write better code without being an expert.

This simple thinking pattern will make you a better programmer even without any expert knowledge on coding!

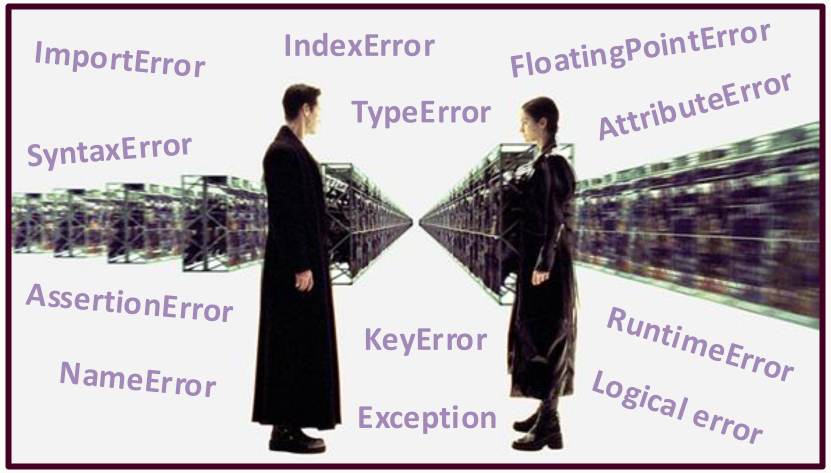
"All I want to know is where I'm going to die, so I'll never go there." - **Charlie Munger**

Expect problems and embrace them

Programming can be **frustratiiing!** 😤

You may get stuck on a problem you've solved before, or cryptic error messages mess up your data collection.

Errors do not mean you are doing it wrong!



Programming is about **overcoming problems**. Expect many of them and practice **radical acceptance**. Use Google. Ask colleagues. Go for a walk. **Helaas pindakaas!**

You learn more than you think, even if it does not feel like it. Learning reveals itself in hindsight, or did you think a PB&J sandwich had anything to teach you?

Focus on practice, not theory

Following tutorials is useful, but it can also become a way of avoiding actually doing something. **Get your hands dirty**!

This lecture is meant to set you off in the **right direction**, but ultimately, only **practice** will make you a good programmer.

Programming is like playing an instrument.
Theory helps but you learn a craft through practice.

Credit: Cultura RM Exclusive/Phil Fisk/Getty Images

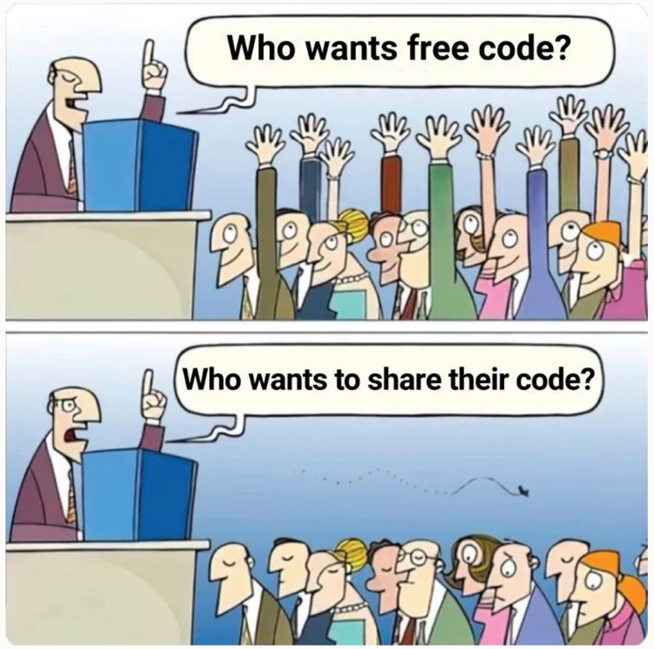
Sharing is caring

It can be tempting to solve problems through “**dirty hacks**” **instead of better code**. This approach carries **high risk!**
(e.g., you may forget that you hardcoded a number)

A **helpful thinking pattern** here is keeping in mind that **you will share your code later** (e.g., alongside a paper).

This **mindset** will make you more **aware**, **careful**, and **pro-active,** ultimately **helping others and your future self**.
(e.g., by writing better code documentation)

If customers wonder what on earth you put on their sandwich, it helps to be able to show the ingredient list and recipe!



Be the change! Just do it!

Don’t trust your memory

Good code speaks for itself, but most code is okayish.

Writing **documentation is key** (e.g., code comments), again not only **for others** but also **for your future self**.

Commenting code best practices:
https://stackoverflow.blog/2021/12/23/best-practices-for-writing-code-comments/

I have spent A LOT of time figuring out my own code.
Trust me, spending some time on better documentation will pay off later!

My early comments be like…

Take-home messages – Part II

- Programming is **more than coding**. It's a **way of thinking** about **solving problems**.
- **Good programming** is measured not by what the code does, but **how it is written**.
- Certain **thinking patterns** will help you **write better code** without expert knowledge.

1) Keep the goal in mind
2) Think about task decomposition
3) Remember that there is no spoon
4) Expect problems and embrace them
5) Avoid stupidity
6) Focus on practice
7) Don't trust your memory

What's next?
Follow practical 1.2 from home!

Thanks and have fun!

Thanks Dave Collien for helpful discussions on this lecture