Lecture 3:

INTRO TO PYTHON & PAIR PROGRAMMING

CSC111: Introduction to CS through Programming

R. Jordan Crouser

Assistant Professor of Computer Science

Smith College

Announcements

- Note Taker: paid position available
  - **Requirements**: on work study, 1.5hr week
  - **What you'll do**: take clear, concise notes for this course and deliver them to the Office of Disability Services
  - **If interested**: email Lisa Roberge in ODS at lroberge@smith.edu (subject: *Note Taker for CSC111-01 / R. Jordan Crouser*)

Overview of the week

- ✓Crash course in computers
  - ✓A little history
  - ✓4 key components
  - ✓Quick hardware demo
  - ✓Boolean logic
- Introduction to Python
- Life skill #1: pair programming
- Lab: Getting Started with Python
- User Input

Recap

What do you remember

from Monday's class?

Recap: the good news

- “High level” programming languages like Python mean we don’t have to write in “low level” binary

- Instead, we write statements like:

```
print(“hello”)
```

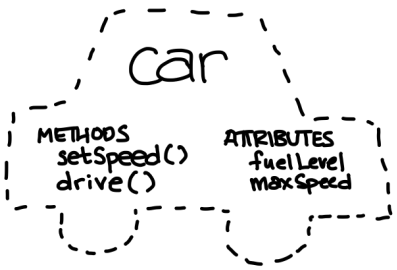

python
TM

What do you know?

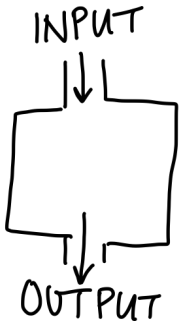
multi-paradigm
interpreted language
with dynamic typing
and automatic memory management

python™

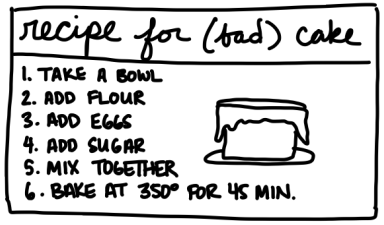
multi-paradigm
interpreted language
with dynamic typing
and automatic memory management



object-oriented



functional



imperative

declarative

multi-paradigm
interpreted language
with dynamic typing
and automatic memory management

Quick digression

	COMPILER	INTERPRETER
What it takes in:		
What it returns:		
Relative speed:		
Memory usage:		
Work is done:		
Reports errors:		
Example language:		

Quick digression

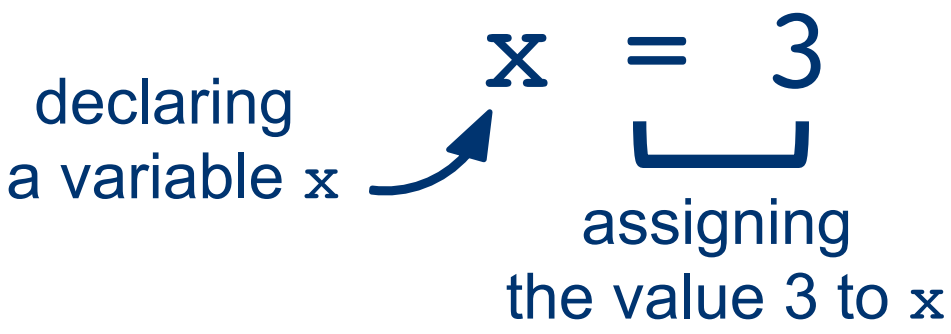
	COMPILER	INTERPRETER
What it takes in:	an entire program	a single line of code (or a single instruction)
What it returns:	intermediate object code (e.g. classes)	<nothing>
Relative speed:	faster	slower
Memory usage:	uses more memory (↑ objects take space ↑)	uses less memory (no intermediate objects)
Work is done:	just once	every time the code is executed
Reports errors:	after checking the entire program	after each instruction is run

multi-paradigm
interpreted language
with dynamic typing
and automatic memory management

more about this
a bit later

Core concept 1: `variables`

- In CS, a **`variable`** is a place to store a piece of data
- In Python, variables are:
  - **declared** by giving them a name
  - **assigned** using the equals sign
- Example:



Core concept 2: numeric values

- Two kinds of **numbers** in CS:
  - integers (“whole numbers”)
  - floats (“decimals” or “floating point numbers”)
- In Python, the kind of number is implied by whether or not the number contains a **decimal point**
- Example:

```
x = 3

x = 3.0
```

Basic operations on numbers

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Exponentiation: `**` (power)
- Modular arithmetic: `%` (modulo)

Core concept 3: strings

- In CS, a sequence of characters that isn't a number is called a **`string`**
- In Python, a string is declared using **quotation marks**
- Strings can contain letters, numbers, spaces, and special characters
- Example:

```
x = "Jordan"
x = "Stoddard G2"
```

Math on strings?!

- Let's say we have:

```
x = "Jordan"
```

- If we ask Python for:

```
2*x
```

what will happen?

`print()`: a very useful function

- A function is a procedure / routine that takes in some input and does something with it (just like in math)
- In Python, the **`print()`** function takes in a value and outputs the value to the console
- This seems silly now, but will come in handy in lab when you write/run your first program inside a **file**

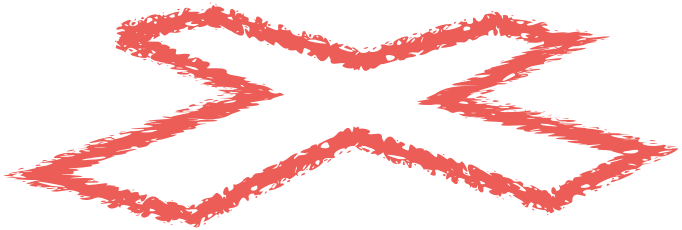
Quick check in

```
a = 1
b = 2.7
c = “Smithies”
```

Overview of the week

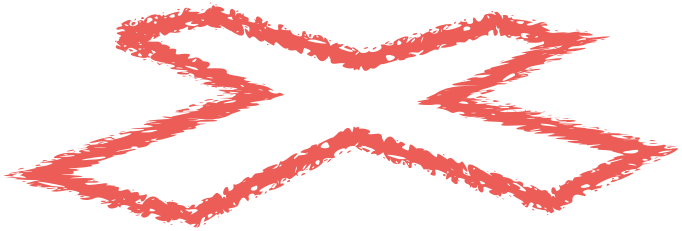
- ✓Crash course in computers
  - ✓A little history
  - ✓4 key components
  - ✓Quick hardware demo
  - ✓Boolean logic
- ✓Introduction to Python
- Life skill #1: pair programming
- Lab: Getting Started with Python
- User Input

The programming process



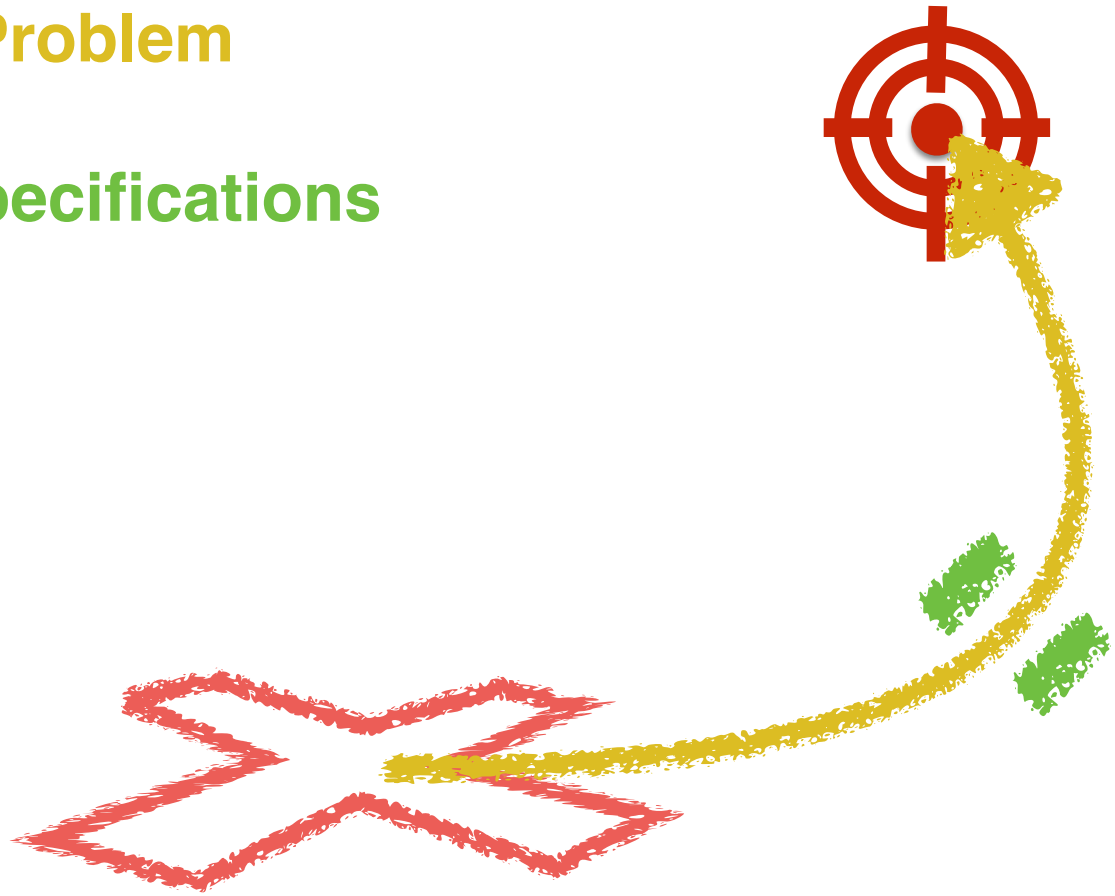
The programming process (idealized)

- Analyze the **Problem**



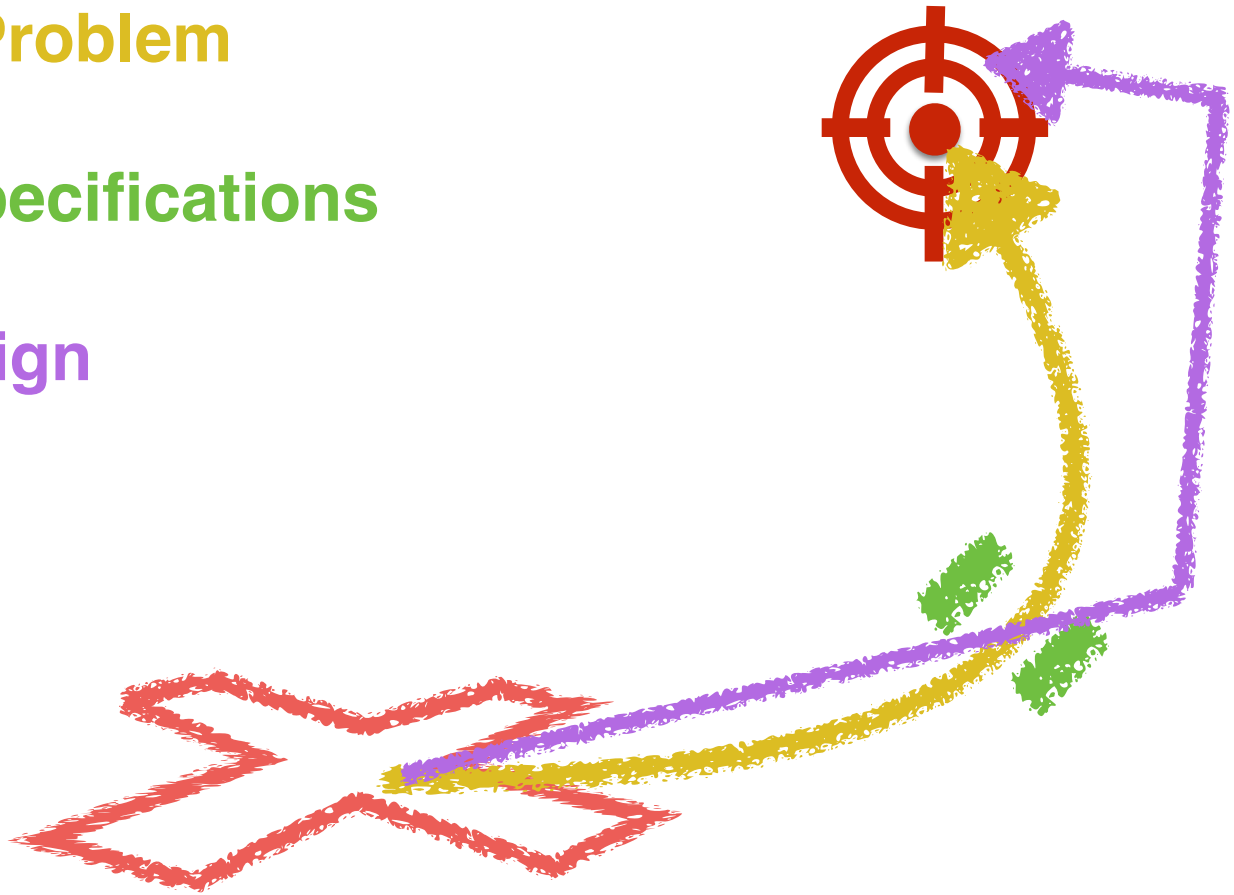
The programming process (idealized)

- Analyze the **Problem**
- Determine **Specifications**



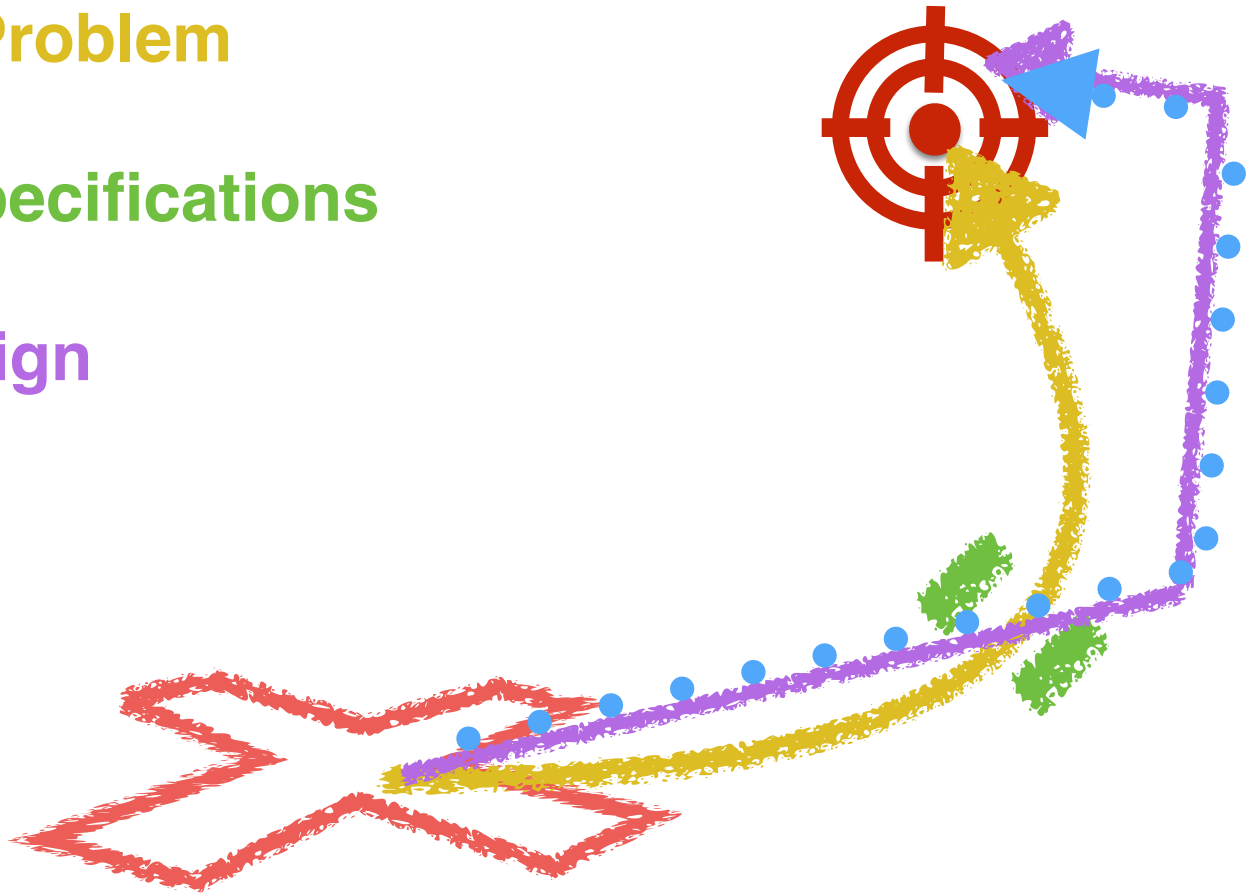
The programming process (idealized)

- Analyze the **Problem**
- Determine **Specifications**
- Create a **Design**



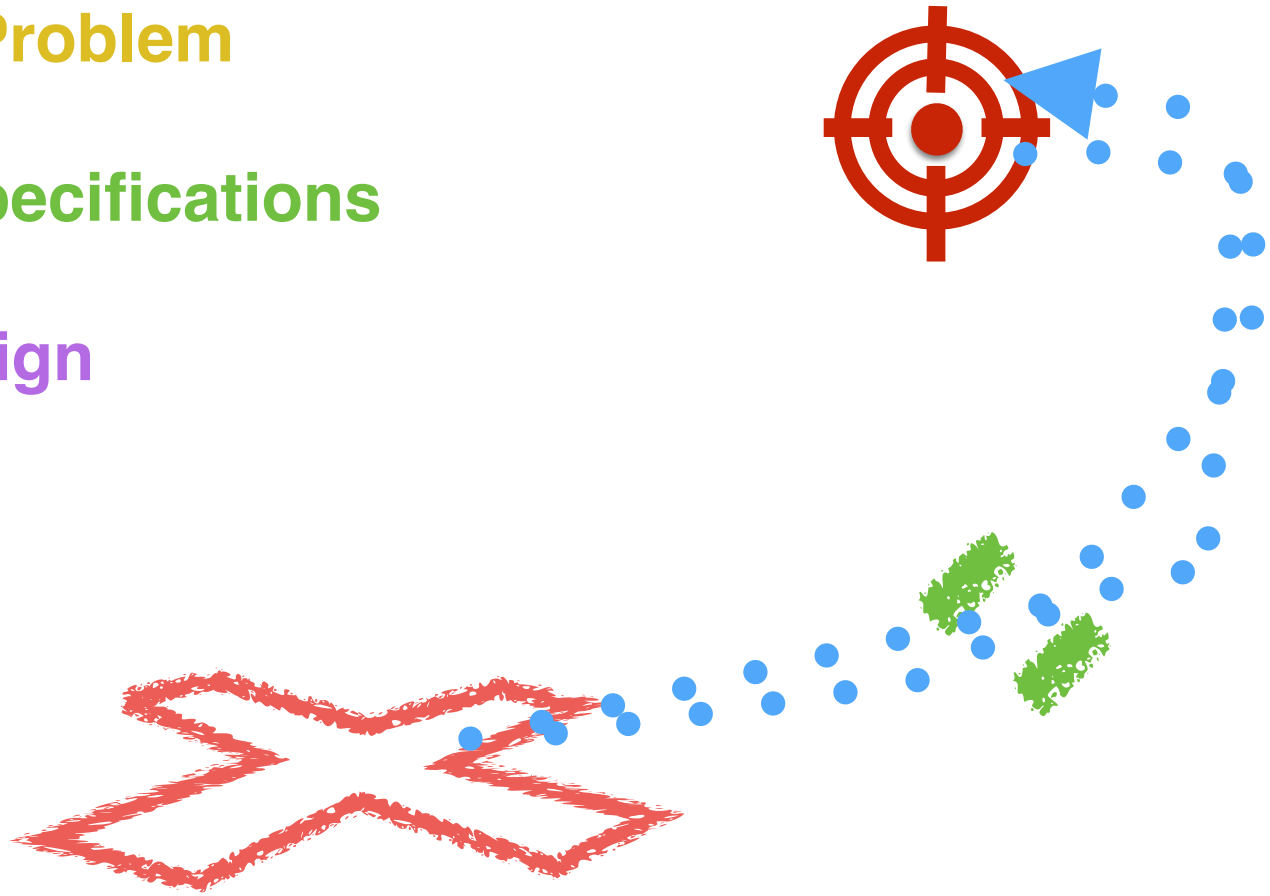
The programming process (idealized)

- Analyze the **Problem**
- Determine **Specifications**
- Create a **Design**
- **Implement**



The programming process (idealized)

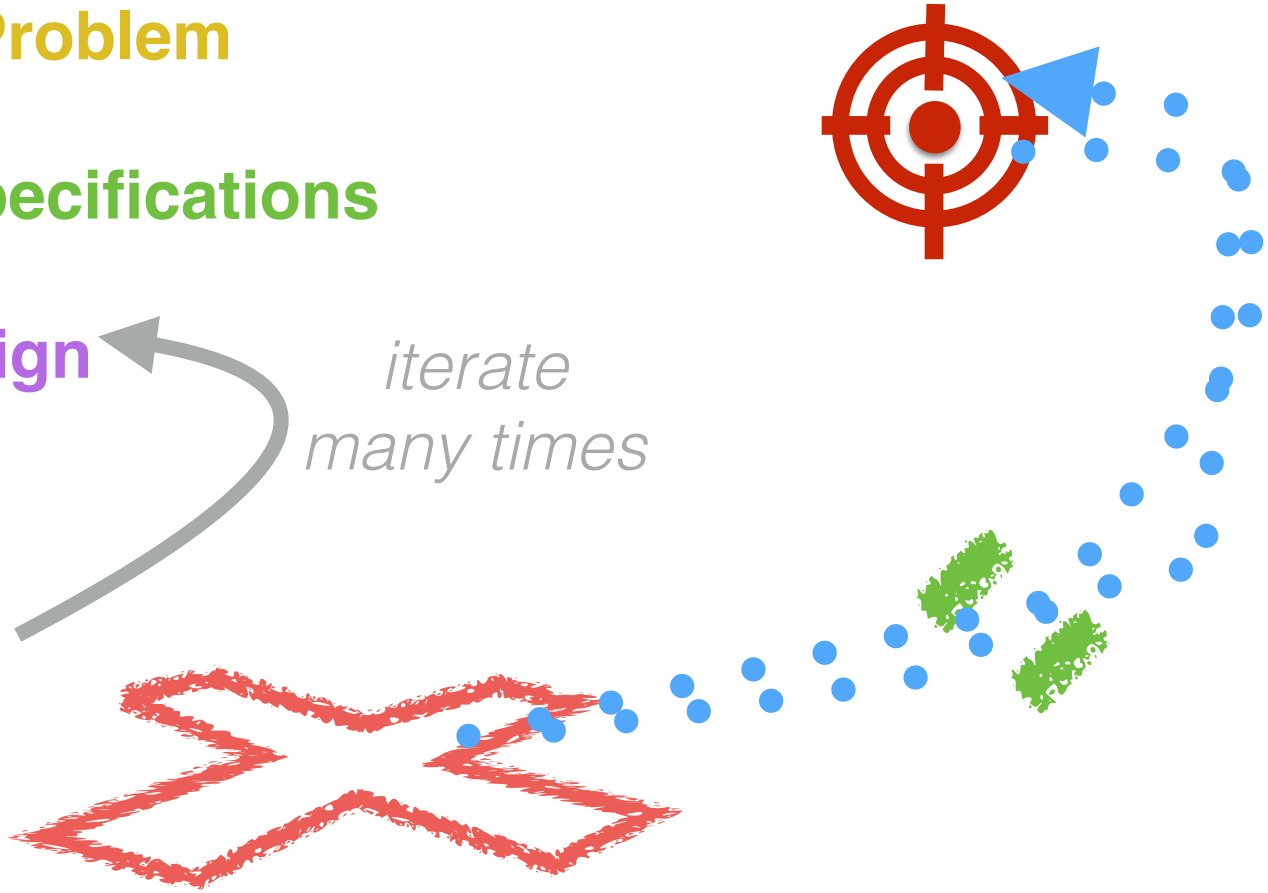
- Analyze the **Problem**
- Determine **Specifications**
- Create a **Design**
- **Implement**
- Test & Debug



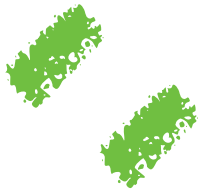
The programming process (more realistic)

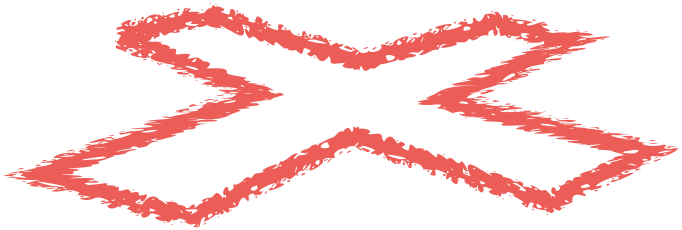
- Analyze the **Problem**
- Determine **Specifications**
- *Refine the* ~~Create a~~ **Design**
- **Implement**
- Test & Debug

iterate many times

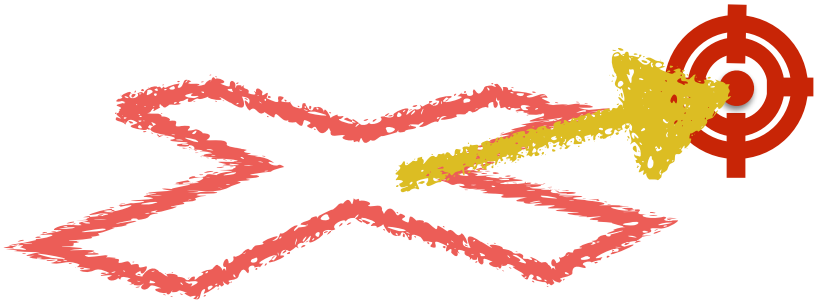


Getting started

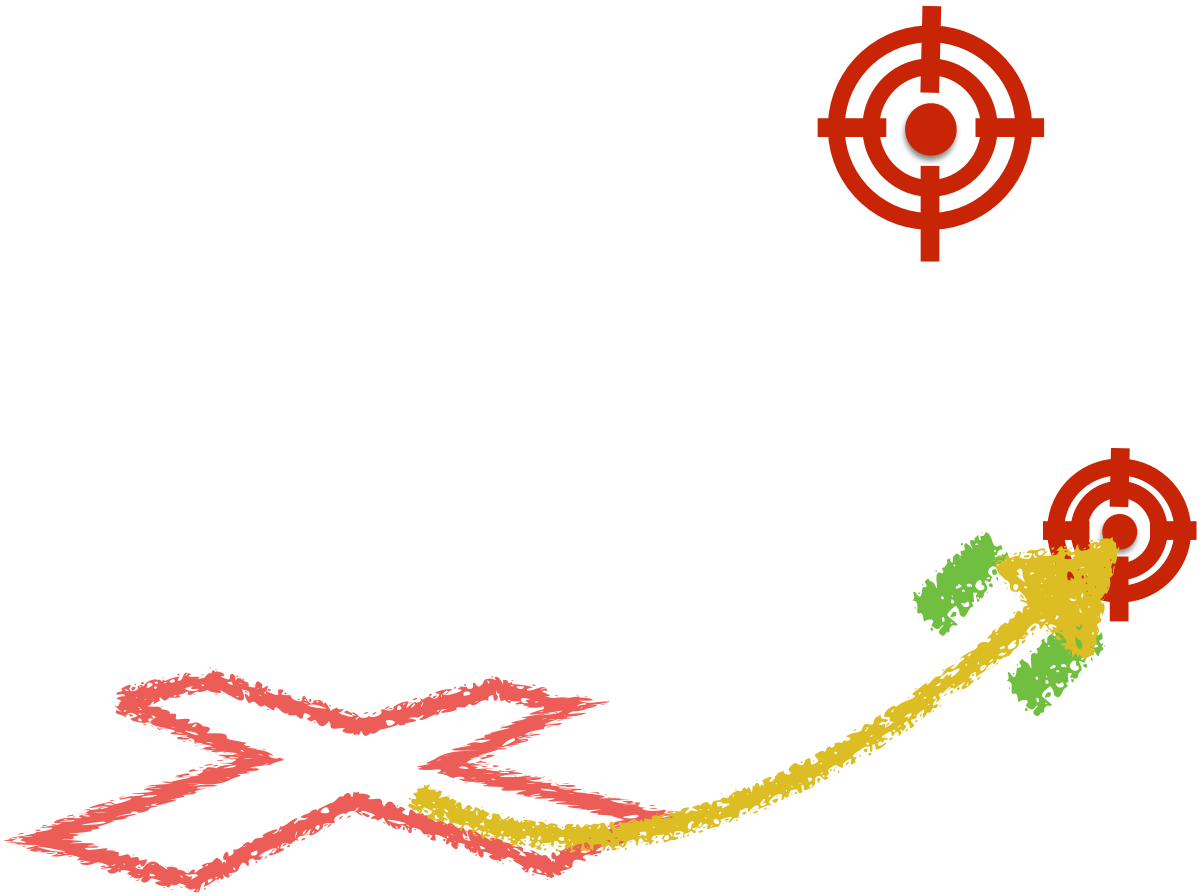




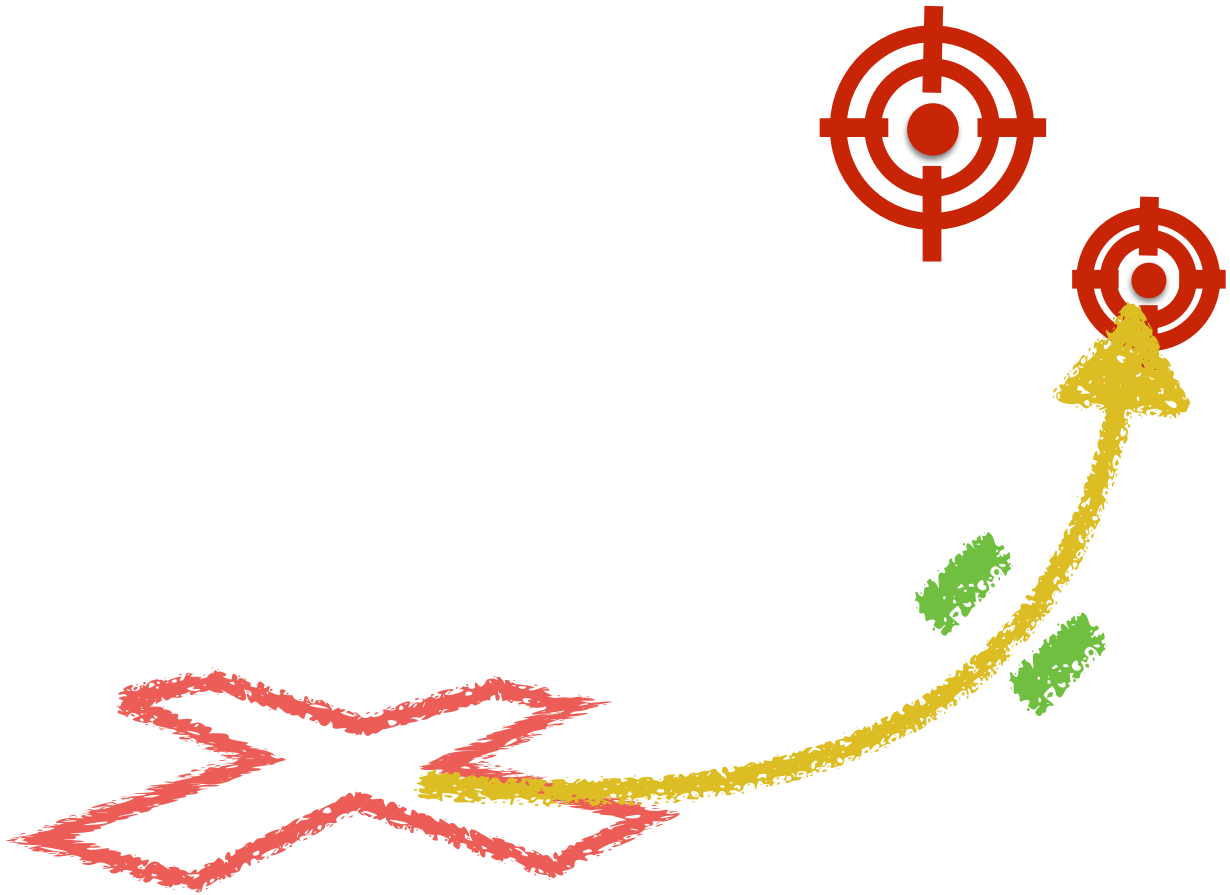
“S^4”: start small | slow | simple



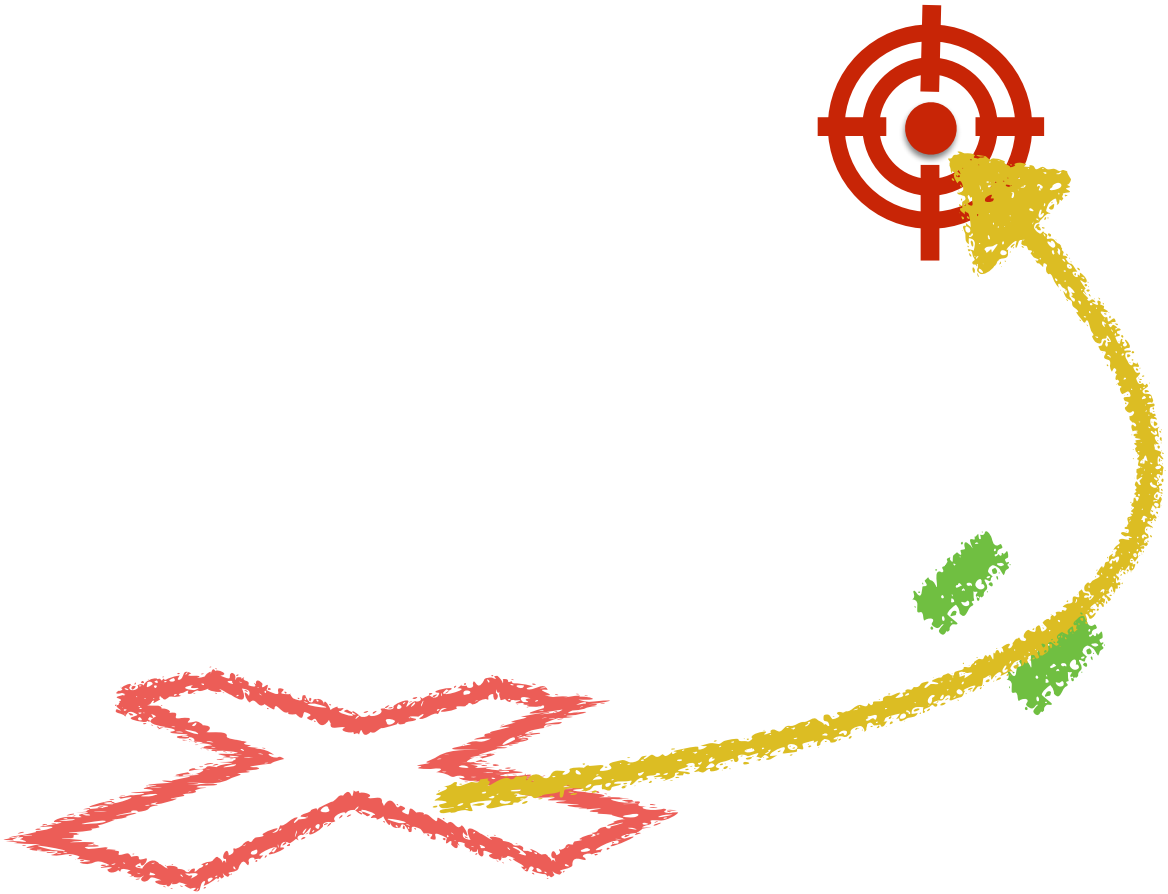
Next: address the constraints



Add additional features



Finally: hit target



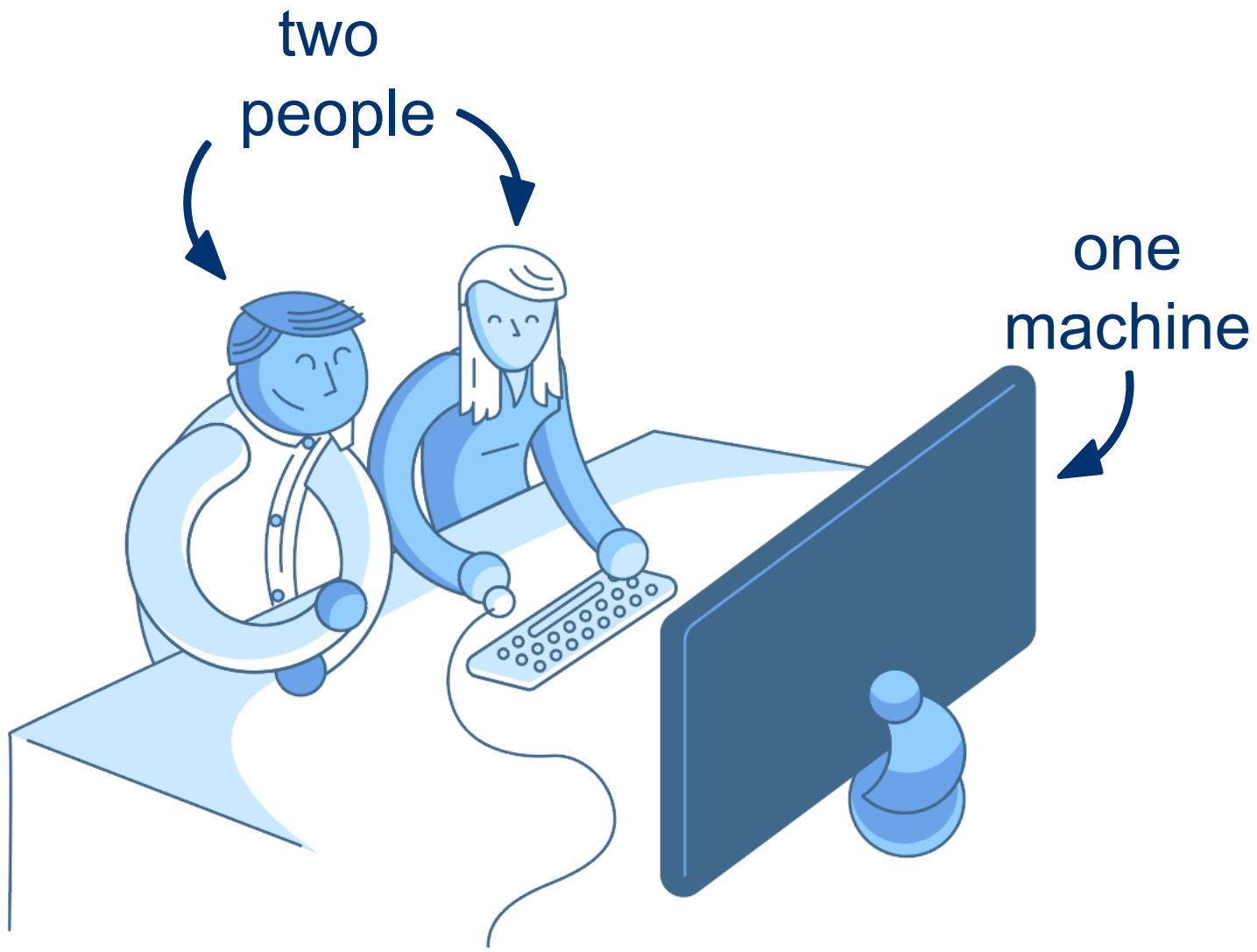
Discussion

So what does this

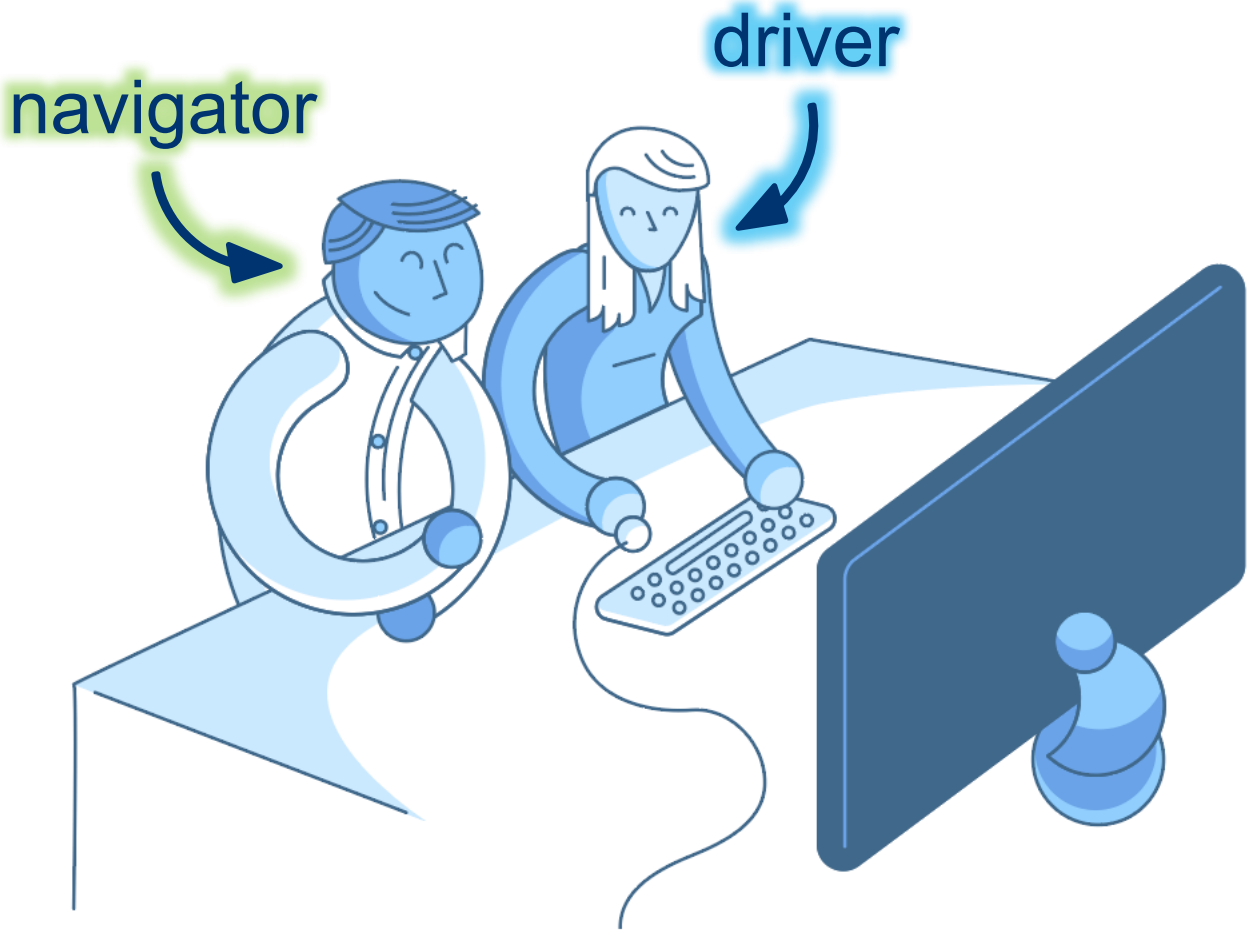
look like in practice?

A problematic (but common) model

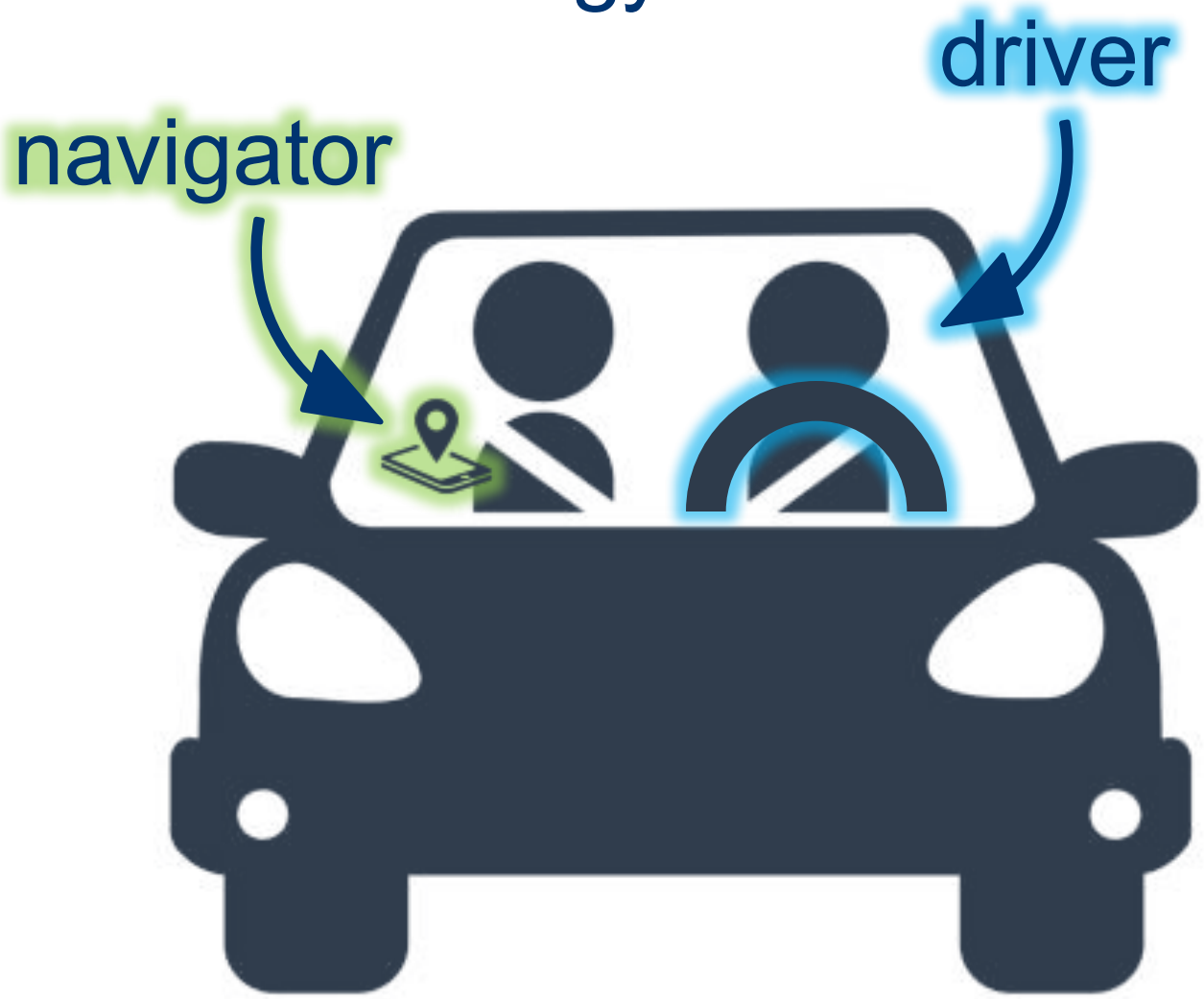
A better model: “pair programming”



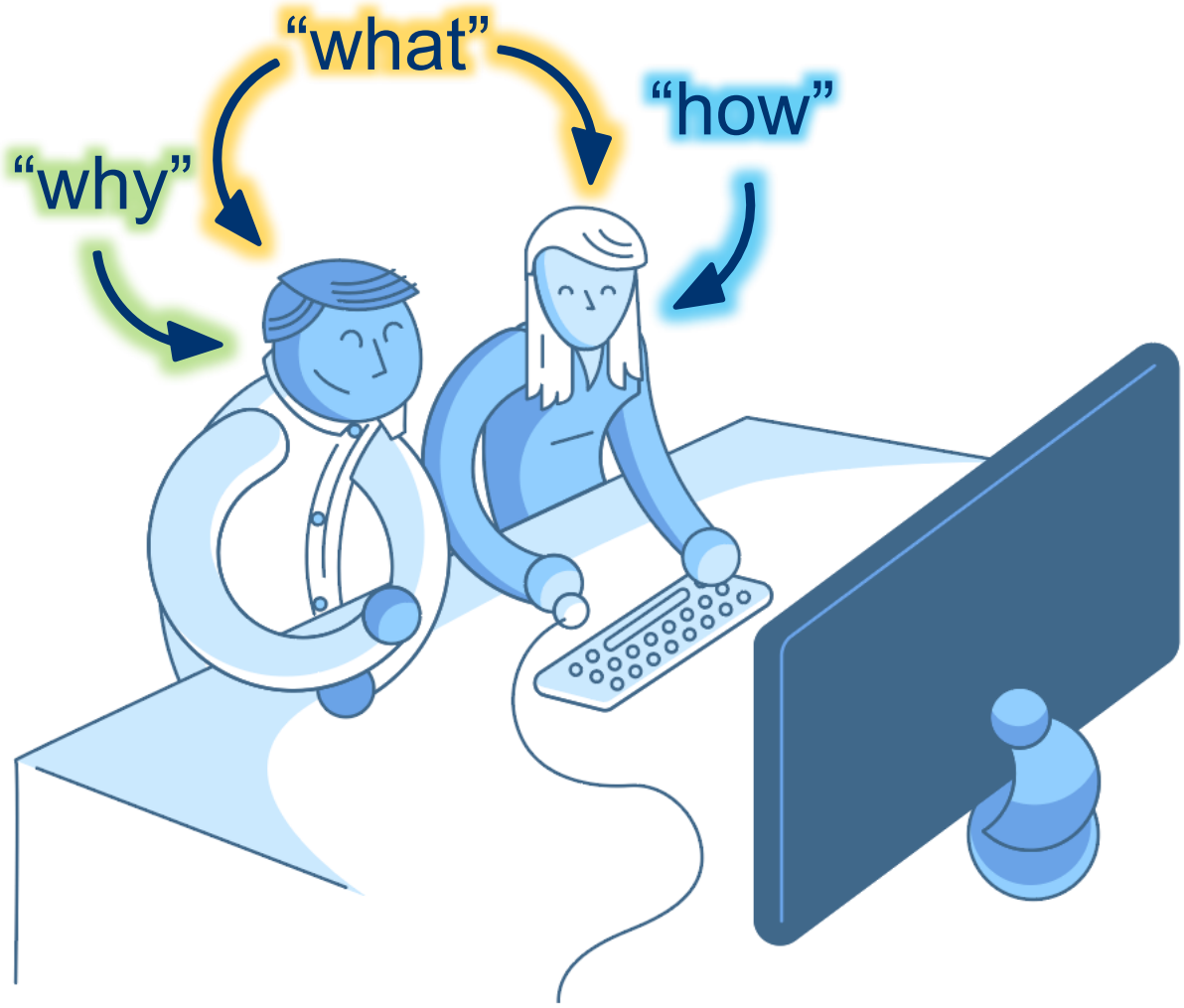
Two complimentary roles



A common analogy



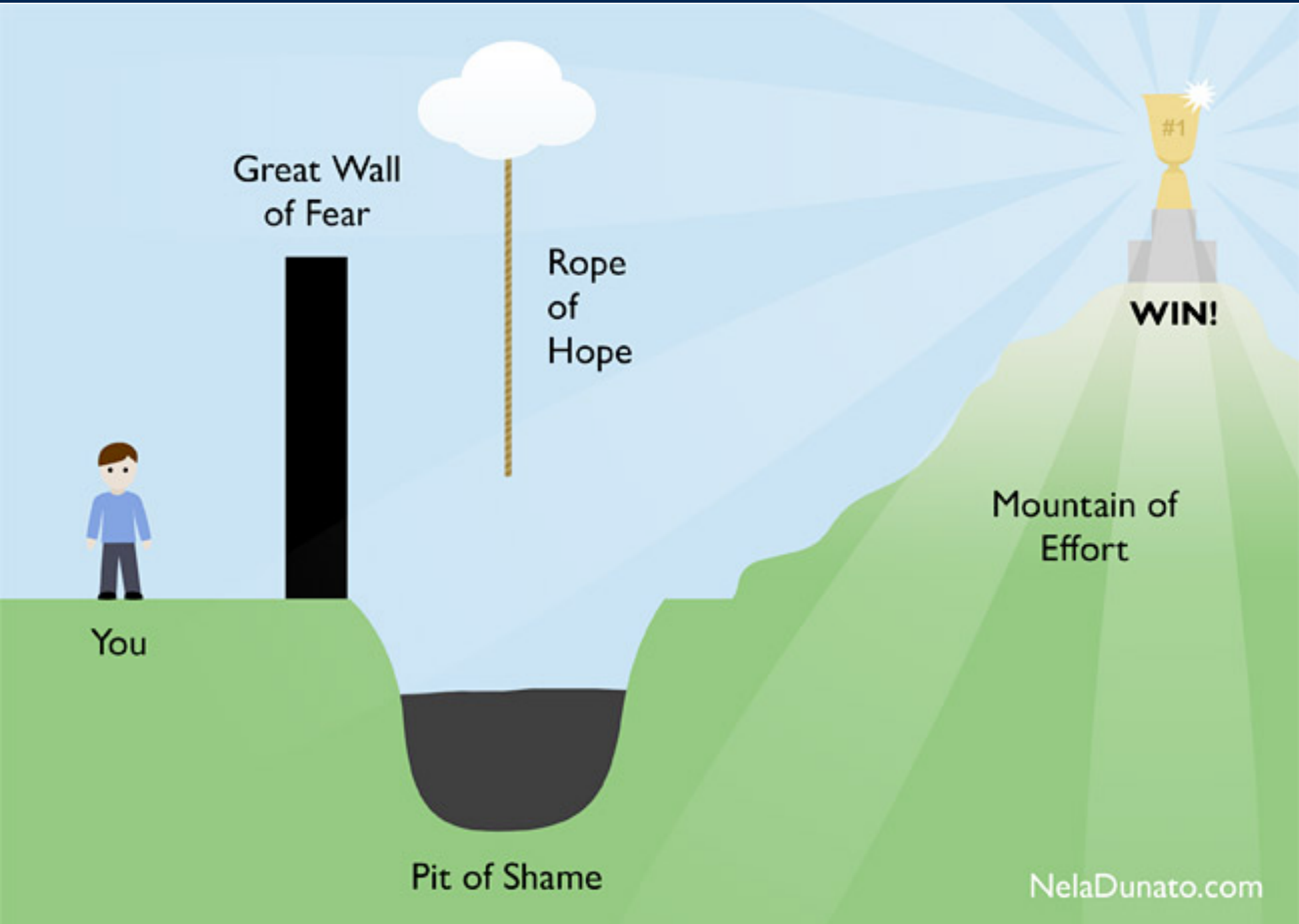
Navigator vs. driver: different focus



A professional perspective (& some tips)

Reality check

What makes this approach
so hard?


Great Wall
of Fear
Rope
of
Hope
#1
WIN!
Mountain of
Effort
You
Pit of Shame
NelaDunato.com

Overview of the week

- ✓Crash course in computers
  - ✓A little history
  - ✓4 key components
  - ✓Quick hardware demo
  - ✓Boolean logic
- ✓Introduction to Python
- ✓Life skill #1: pair programming
- Lab: Getting Started with Python
- User Input

Your first taste of pair programming!