

FUNDAMENTALS OF AI-AUGMENTED ENGINEERING: A CONVERSATIONAL APPROACH

Taner Onsay, Ph.D. & Sebnem Onsay
Authors
August 2026 · Revision 1.0

© 2026 AI First Engineering

Problem



Model



Compute



Analyze



Validate



Improve

*The engineering workflow you will use throughout this book
by Taner & Sebnem Onsay,*

CHAPTER - 2

PART 2 · ≈ 75 MIN

Decisions & Repetition

2

Real procedures branch — and repeat. A warm-up engineering problem will force both into the open.

Problem



Model



Compute



Analyze



Validate



Improve



Last time: you turned thinking into algorithms

AN ALGORITHM

Precise steps. Nothing left to common sense.

the toast — every step explicit, in order

FIVE QUESTIONS

Inputs · Process · Decisions · Repetition · Outputs

the checklist that exposes any procedure

PSEUDOCODE

GET · SET · COMPUTE · REPORT

the fall time $t = \sqrt{2h/g}$, written for the assistant

RUN IT BY HAND

Trace the boxes. Predict before anything executes.

$h = 20\text{ m} \rightarrow t \approx 2\text{ s}$ — you checked it yourself

Two of the five questions you could ask — but not yet write: DECISIONS (where can the path split?) and REPETITION (what happens again and again — and when does it stop?). Part 1's procedures ran straight through. Today, pseudocode learns to branch and to repeat.

Problem



Model

**Compute**

Analyze



Validate



Improve



Today, pseudocode learns two new words: IF and REPEAT

WHAT YOU CAN ALREADY WRITE

Straight-line pseudocode

```
GET height h
SET g = 9.81
COMPUTE t = sqrt(2*h/g)
REPORT t
```

Inputs, a known constant, ordered steps, an output. Traced by hand, one line at a time.

WHAT IT STILL CANNOT SAY

Two words were missing

```
IF toast is too pale
    PRESS lever again

REPEAT 4 TIMES
    FORWARD 50
    TURN LEFT 90
```

The toast needed a decision; the square begged for a repeat. Today pseudocode gets both — and then Python does.

WHAT WE USED WITHOUT NAMING

The boxes

```
h → [ 20 ]
g → [ 9.81]
t → [ 2.02]
```

Named boxes that hold a value. Programmers call them variables. Ten minutes on them now saves an hour of confusion later.

Plan for this part: variables (briefly) → a decision, first in everyday terms, then on the cargo hoist → repetition, first on a list you can count, then on the hoist → and the one detail that makes a loop dangerous.

Problem



Model



Compute



Analyze



Validate



Improve



Decisions and repetition both need one idea: a **variable**

THE FIRST NEW WORD WILL NEED THIS

A decision asks a question about a value.

`is temperature ≤ 50 ?`

Before the question can be asked, those values have to live somewhere — and keep living there between steps.

Variable → store a value → decision checks it

SO WILL THE SECOND

A repetition changes a value each time the work repeats.

`count = count + 1`

Something must remember where the work stands each time around — or the repetition learns nothing.

Variable → store a value → repetition changes it

That “somewhere” is a variable — a named box that holds a value. You already used these boxes in Part 1 without naming them: h, g, t. Five minutes to make them official — the way programmers see them — and then straight back to decisions and repetition. This short detour is what makes both new words make sense.

Problem



Model



Compute



Analyze



Validate



Improve



A variable is a box with a name on it

A RIDDLE · KEEP TRACK OF ONE NUMBER

	people
You are driving a bus. It starts with 5 people.	5
3 get on	8
2 get off	6
4 get off	2
5 get on	7

You kept one box, and you overwrote it every step. That box is a variable: **a name on the outside, a value inside, and the value changes as lines run.**

THE SAME RIDDLE, WRITTEN DOWN

```
people = 5
people = people + 3
people = people - 2
people = people - 4
people = people + 5
```

Read `people = people + 3` as:
“take what is in the box, add 3, put the result back.”

Not an equation. An action. Right side first, then store.

= is not “equals” — it is “put into”. `x = 5` means: store 5 in the box called x. So `x + 1 = 3` is meaningless, while `x = x + 1` is the most common line in engineering code.

Only the most recent value survives. After `x = 5` then `x = 3`, the 5 is gone. A copy made earlier (`y = x`) keeps the old value — it was copied, not linked.

Names carry meaning; pick engineering ones. `current_load`, `safe_limit`, `temp_c` — not a, b, c. Your AI companion and your future self read names, not intentions.

On the hoist, one box does all the work: `current_load` starts at 0 and gets a crate added each time a check passes — `current_load = current_load + load`. It is the bus, with kilograms.

Problem



Model



Compute



Analyze



Validate



Improve



A variable stores the value. It does not judge the value..

FROM COUNTING PEOPLE TO TRACKING LOAD

current_load (kg)

A hoist platform starts empty.

0

a 250 kg crate goes on

250

a 300 kg crate goes on

550

a 275 kg crate goes on

825

another 250 kg crate goes on ...

1075 ?

Same line every time: `current_load = current_load + crate`

THE BOX CANNOT SAY NO

A variable holds a number. It has no opinion about it.

Straight-line pseudocode — GET, SET, COMPUTE, REPORT — runs every line, every time. It will happily add the fifth crate and store 1075 kg in the box, because nothing in the procedure can react to what the number has become.

Almost every engineering procedure has a limit somewhere: **safe working load, maximum temperature, allowable stress, a tolerance**. At the limit, the procedure must do one thing or another.

That is the word pseudocode is still missing. We will meet it on a problem simple enough that the physics gets out of the way.

Next: the cargo hoist. One number in a box, one limit painted on the wall, four crates. Before each crate someone — or something — must decide. Keep your eye on the box: it is the same `current_load`, all the way through.

Problem



Model



Compute



Analyze



Validate



Improve



The cargo hoist



THE SITUATION

Safe working load: **1,000 kg**

Loads waiting: **250 kg · 300 kg · 275 kg · 250 kg**

Loads are added one at a time.

THE ENGINEERING QUESTION

Before adding each load, someone — or something — must decide: **is it safe to add this one?**

Deliberately simple physics. The new idea is the decision, not the mechanics.

Problem



Model



Compute



Analyze



Validate



Improve



PREDICT



**Current load 825 kg. Next load 250 kg.
Safe working load 1,000 kg.
Can we safely add it?**

Answer as an engineer — no code yet.

Now say your reasoning out loud, slowly. Listen for two words: IF ... OTHERWISE.

Problem



Model



Compute



Analyze



Validate



Improve



IF, in everyday terms — before the hoist

ONE BRANCH · IF

Do something extra — or not

```
GET sky
IF sky is raining
    TAKE umbrella
LEAVE the house
```

Raining → umbrella, then leave. Not raining → just leave. The last line runs either way: the paths rejoin.

TWO BRANCHES · IF / ELSE

Exactly one of two things

```
GET number v
IF v < 0
    SET v = -v
ELSE
    LEAVE v alone
REPORT v
```

Absolute value. The question $v < 0$ has a yes/no answer; the answer picks the path. Every input takes exactly one path.

THREE OR MORE · ELSE IF

Ranges, checked in order

```
GET pressure p      # psi
IF p < 30
    REPORT "low"
ELSE IF p <= 35
    REPORT "ok"
ELSE
    REPORT "high"
```

Tire pressure. Each test is asked only if the ones above were false — so $p \leq 35$ already means $30 \leq p \leq 35$.

ANATOMY OF A DECISION

CONDITION a question with a yes/no answer

TRUE PATH runs only when the answer is yes

FALSE PATH runs otherwise — may be empty

REJOIN everything after runs either way

Problem



Model



Compute



Analyze



Validate



Improve



How a sentence becomes an IF

1 · SAY THE RULE

In an engineer's words

"Add the crate only if the total stays at or under the safe working load."

This is what you said on the PREDICT slide. Every decision starts as a sentence like this.

2 · FIND THE QUESTION

A yes/no question with numbers in it

Is `current_load + new_load` $\leq$ `safe_limit` ?

The IF needs something that is either true or false. Comparisons — $\leq$, $<$, $\geq$, $>$, $=$ — turn a rule into a question.

3 · NAME THE TWO PATHS

What happens on yes; what happens on no

yes $\rightarrow$ accept: box grows

no $\rightarrow$ reject: box stays

Absolute value had a path that did nothing. Here both paths matter — one of them is a safety refusal.

4 · NAME THE BOXES

Every quantity in the question is a variable

`current_load` = 825

`new_load` = 250

`safe_limit` = 1000

Three boxes. Notice which one changes (`current_load`) and which one never should (`safe_limit`).

Run the question with today's numbers: $825 + 250 \leq 1000 \rightarrow 1075 \leq 1000 \rightarrow \text{False} \rightarrow$ the no-path: reject. That is the whole decision. The next slide only writes it down.

Problem



Model



Compute



Analyze



Validate



Improve



IF / ELSE = an engineering decision, written down

YOUR REASONING

```
# PSEUDOCODE
IF current + new <= limit
    accept the load
ELSE
    reject the load
```

THE SAME REASONING — EXECUTABLE

```
hoist_check.py
if current + new_load <= SAFE_LIMIT:
    accept()
else:
    reject()
```



The condition is a Boolean question — it is either True or False.

current + new_load <= SAFE_LIMIT asks a yes/no engineering question. Comparisons like <=, <, >, >=, == are how conditions get written.



The engineering lives in the condition, not the colon.

Should the boundary be <= or <? Is exactly 1,000 kg safe? That is an engineering judgment — and exactly the kind of detail AI-generated code can get silently wrong.

Problem



Model



Compute



Analyze



Validate



Improve



The decision works. Now it must happen again and again.

WHAT YOU JUST BUILT — ONCE

```
IF current + new <= limit
```

```
    accept the load
```

```
ELSE
```

```
    reject the load
```

One crate arrives — one decision, handled.

WHAT THE DOCK ACTUALLY LOOKS LIKE

250 kg

same check

300 kg

same check

275 kg

same check

250 kg

same check

Four crates waiting — every one needs the SAME decision.

Copy the IF/ELSE four times? And tomorrow, when forty crates arrive?

This is the second new word: repetition. Perform the same operation for each item in a collection. Exactly the itch the square gave you in Part 1 — and, as always, we practice on something you can count in your head before we come back to the crates.

Problem



Model



Compute



Analyze



Validate



Improve



REMEMBER THE SQUARE?

Repetition you already wanted: REPEAT — and its limit

PART 1 · THE SQUARE, AS WRITTEN

```
FORWARD 50  
TURN LEFT 90  
FORWARD 50  
TURN LEFT 90  
FORWARD 50  
TURN LEFT 90  
FORWARD 50  
TURN LEFT 90
```

the same two lines, four times over

WHAT WE WISHED FOR

```
REPEAT 4 TIMES  
  
FORWARD 50  
  
TURN LEFT 90
```

Enough — when every pass is identical.
Each square pass does exactly the same thing.

BUT THE CRATES DIFFER

```
FOR each item in a list  
  
do the block with it
```

Pass 1 handles 250 kg, pass 2 handles 300 kg ...

When each pass works on a different item, the loop hands you the next one — that is FOR.

Next: your first FOR — on three plain numbers, nothing else. *Then the crates.*

Problem



Model



Compute



Analyze



Validate



Improve



Add up 2, 4, 1 — and watch the loop work

THE TASK: add up the three numbers 2, 4, 1 and report the total.

THE PSEUDOCODE

```
SET total = 0

FOR each n in [2, 4, 1]
    SET total = total + n

REPORT total
```

n is handed the next number at the top of each pass — 2, then 4, then 1.
total is the running sum — the bus from the variables slide, overwritten every pass.

TRACE IT — SAY IT OUT LOUD

pass	n	total
start	—	0
1	2	2
2	4	6
3	1	7
done	—	7

Three items → three passes → answer 7. You knew the pass count before it started — that is what makes it a FOR.

Problem



Model



Compute



Analyze



Validate



Improve



Same loop, new list, a decision in the body

THE LOOP YOU JUST TRACED

```
SET total = 0
FOR each n in [2, 4, 1]
    SET total = total + n

REPORT total
```

THE SAME LOOP, ON THE HOIST

```
SET current_load = 0
FOR each load in [250, 300, 275, 250]
    IF current_load + load <= 1000
        SET current_load = current_load + load
    ELSE
        REPORT "reject", load
REPORT current_load
```

WHAT DID NOT CHANGE

The shape. Set a box to zero · FOR each item in a list · update the box · report the box.

The two jobs. load is n — handed the next item each pass.
current_load is total — the accumulator, the bus.

WHAT CHANGED

The list. Four crate masses in kilograms instead of three bare numbers — you can still count the passes: four.

The body. The addition is now wrapped in the IF/ELSE from the hoist decision. A pass may add the crate, or refuse it.

A loop with a decision inside it is the most common shape in engineering code. *Next slide: the same seven lines, in Python — trace them before anyone runs them.*

Problem



Model



Compute



Analyze



Validate



Improve



MANY LOADS

The same check, for each load: FOR

Four loads are waiting. Do we write the safety check four times? What if tomorrow there are forty?

PYTHON — ONE CHECK, APPLIED TO EACH LOAD

```
hoist_loop.py

loads = [250, 300, 275, 250]
current = 0
for load in loads:
    if current + load <= 1000:
        current = current + load
    else:
        print("Reject", load, "kg")
print("Final:", current, "kg")
```

kg

ONE-DIMENSIONAL ARRAY
A sequence of values stored under one name.
Here: 4 load values
It could hold 40, 10,000, or millions of values.
In Python, this structure is called a *list*.

accept

TRACE IT BY HAND

250:	$0 + 250 \leq 1000$	→	accept	(250)
300:	$250 + 300 \leq 1000$	→	accept	(550)
275:	$550 + 275 \leq 1000$	→	accept	(825)
250:	$825 + 250 > 1000$	→	reject	(825)

Predict the final value before anyone runs it. Tracing a loop on paper is a core debugging skill.

FOR: “Perform this operation for each known item or case.” The loop variable load takes each value in the list in turn.

Problem



Model



Compute



Analyze



Validate



Improve



FOR vs WHILE — an engineering distinction



FOR

“Perform this operation for each known item or case.”

You know the collection in advance.

Hoist: FOR each waiting load → run the safety check.



WHILE

“Keep performing this operation until an engineering condition changes.”

You don't know how many repetitions — the condition decides.

Coming next: WHILE stopping is still safe → try a higher speed.

Ask the engineering question first: do I know the cases, or do I know the condition? The loop type follows.

Problem



Model



Compute



Analyze



Validate



Improve



WHILE: keep going until the condition says stop

AN ENGINEER'S WHILE

```
WHILE torque is below spec  
  TIGHTEN a quarter turn  
  READ the torque gauge  
MOVE to the next bolt
```



Nobody knows in advance whether it takes three turns or ten. The condition decides — and it is checked at the top, before every turn, including the first.

Something inside the loop must be able to change the answer. Here it is the tightening: every quarter turn raises the torque.

THE SAME SHAPE WITH A NUMBER

```
SET value = 1  
WHILE value < 6  
  REPORT value  
  SET value = value * 2  
REPORT "done"
```



Output: 1 2 4 done

Four checks, three passes. The fourth check sees $8 < 6$
→ no → skip to the line after the loop.

TRACE IT

value	< 6 ?	action
1	yes	report, x2
2	yes	report, x2
4	yes	report, x2
8	no	leave loop

The loop is a decision that jumps back.

A WHILE loop has three parts, and forgetting the third one is the classic mistake: (1) a condition checked before every pass, (2) a body that does the work, and (3) something in the body that moves the condition toward false. The torque reaches spec; value doubles past 6. Take that third part away — and you get the next slide.



A loop must know when to stop

A WHILE LOOP WITH A BAD STOP

```
heater.py
# What is wrong here?
temp = 20
while temp != 60.5:
    heat_one_second()
    temp = read_sensor()
```



!=	not equal to
()	run this function
=	store/update a value

PREDICT, THEN DISCUSS

The sensor reads 60.4 °C ... then 60.7 °C.

The condition `temp != 60.5` never becomes False.

The loop never ends. The heater never stops.

An infinite loop is not a programming curiosity — in an engineering system it is a hazard.

A stopping criterion is an engineering specification.



“Stop when $\text{temp} \geq 60$ ” works; “stop when temp equals exactly 60.5” may never happen. Choosing a robust stop is judgment, not syntax.

Live demo: an infinite loop, on purpose.



Watch what it looks like and learn how to interrupt it — so the first time it happens to you in the laboratory, it is familiar, not frightening.