

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 4 · ≈ 40 MIN

Functions, Decomposition & the AI Coding Companion

4

One giant block of instructions is how programs — and engineering projects — go wrong. And how you will actually work with AI on code.

Problem

Model

Compute

Analyze

Validate

Improve



WHERE PART 3 LEFT YOU

The search worked — but it called a name nobody defined

THE SEARCH, AS YOU LEFT IT

```
v = 0.0  
step = 0.5  
while stopping_distance(v + step,  
                        t_r, a) <= d_avail:  
    v = v + step  
print("Max safe speed:", v)
```

ONE PROBLEM

stopping_distance(...) does not exist.

You named an engineering task and used the name as if it were real.
Run it today and Python answers:

```
NameError: name 'stopping_distance'  
is not defined
```

Naming a task before it exists was not a mistake. It was the right instinct — today we make it official.

Today: tasks get names. First the cost of NOT naming them (one giant block) · then an everyday example · then what a function is, precisely · then the search gets its missing piece — and you meet the working method for building code with AI.

Problem



Model



Compute



Analyze



Validate



Improve



THE QUESTION

Should we write one enormous program?

IT WORKS... BUT

```
everything_at_once.py

# everything, all at once...
t_r = 1.0
a = 7.0
d_avail = 50.0
v = 0.0
step = 0.5
while (v+step)*t_r + \
      (v+step)**2/(2*a) <= d_avail:
    v = v + step
print(v)
```

WHAT'S WRONG WITH IT?



Hard to read

The physics is buried inside the loop condition. Where is the model?



Hard to test

You cannot check the stopping-distance formula alone — only the whole thing at once.



Hard to reuse

Tomorrow's problem needs the same formula. Copy-paste? Now a fix must be made twice.



Hard to trust

When the answer looks wrong, where do you even start looking?

This code is CORRECT. Engineers still never build a bridge as one piece. Why would we build a program that way?

Problem



Model



Compute



Analyze



Validate



Improve



The same calculation, three times — with a hidden mistake

WITHOUT A FUNCTION — THREE COPIES

```
c1 = (f1 - 32.0) * (5.0 / 9.0)
```

```
c2 = (f2 - 32.0) * (5.0 / 9.0)
```

```
c3 = (f3 + 32.0) * (5.0 / 9.0)
```

Fahrenheit to Celsius, typed three times — and the third copy has a **+** where a **-** belongs. Would you have spotted it?

WITH ONE — WRITTEN ONCE, USED EVERYWHERE

```
def f_to_c(f):
```

```
    return (f - 32.0) * (5.0 / 9.0)
```

```
c1 = f_to_c(f1)
```

```
c2 = f_to_c(f2)
```

```
c3 = f_to_c(f3)
```

 **f_to_c** called 3 times

The calculation now exists exactly once. The mistake cannot be made three different ways; a fix fixes every use; and the main program reads like a sentence. Temperature conversion today — stopping distance in two minutes. The cure has the same shape every time.



A function is a small algorithm with a name

THE PIECES, NAMED

```
def f_to_c(f):
```

NAME · PARAMETER (the input)

```
    c = (f - 32.0) * (5.0 / 9.0)
```

BODY (the process)

```
    return c
```

RETURN (the output)

A call — `c = f_to_c(68.0)` — jumps in carrying 68.0, runs the body, hands 20.0 back, and the main program resumes where it left off.

LOOK FAMILIAR?

A function is a small algorithm with a name.

Parameters are its **INPUTS**. The body is its **PROCESS** — and decisions and loops may live there too. The return value is its **OUTPUT**.

The five questions from Part 1 apply, unchanged. Nothing here is new — it is your own framework, given a handle.

And because the algorithm has a handle, you can hold it to account: feed `f_to_c(212.0)` and demand 100.0 — a hand-checkable test, thirty seconds, before the function ever enters a bigger program. This testability is the whole reason engineers decompose. Now: the same move on the stopping distance.

Problem



Model



Compute



Analyze



Validate



Improve



Functions: named, reusable engineering tasks

PYTHON — THE TASKS GET NAMES

```
def stopping_distance(v, t_r, a):  
    d_reaction = v * t_r  
    d_braking = v**2 / (2 * a)  
    return d_reaction + d_braking  
  
def is_safe(d_stop, d_avail):  
    return d_stop <= d_avail  
  
# meaningful names, testable pieces:  
d = stopping_distance(20, 1.0, 7.0)  
print(d, is_safe(d, 50.0))  
48.57142857142857 True
```

stopping.py

A FUNCTION IS...

a meaningful engineering task — “calculate the stopping distance”, “decide whether this is safe” — packaged with a **name**, its inputs (**parameters**), and its result (**return value**).



Each piece can now be tested by itself.

Feed `stopping_distance()` a hand-calculated case and check — before it ever enters the loop.



The WHILE search now reads like the pseudocode.

Because the physics finally has a name.

Problem



Model



Compute



Analyze



Validate



Improve



The Chapter 1 formula becomes a tool

FALL_TIME() — YOUR FIRST REUSABLE MODEL

```
fall_time.py

import math

def fall_time(h, g=9.81):
    # Chapter 1 model: t = sqrt(2h/g)
    return math.sqrt(2 * h / g)

print(fall_time(1.20))
0.4946193668294979
```

WHY THIS MATTERS IN THE LABORATORY

In Lab 2 you will measure your own reaction time by catching a falling ruler.

The instrument that converts catch distance d into reaction time is this exact Chapter 1 model — running in the other direction:

$$t_r = \sqrt{(2d / g)}$$

Chapter 1: free fall was the phenomenon we investigated. **Chapter 2:** free fall becomes a measurement instrument.

A validated model becoming an instrument is a deep engineering habit — thermometers, strain gauges, and GPS are all validated models run backwards.

Problem



Model



Compute



Analyze



Validate



Improve



Decomposition is an engineering superpower

Break a large engineering computation into smaller, meaningful pieces that can be individually understood, tested, reused, and validated.



Later in this course

Numerical methods arrive as function-sized ideas: a root finder, an integrator, an ODE step. You will assemble them — because you can decompose.



Real engineering

Teams split systems into subsystems with defined interfaces. A function's parameters and return value are your first interface contract.



AI-augmented work

Directing AI through a big task means handing it well-defined pieces and verifying each. Orchestration is decomposition plus judgment.

A well-decomposed problem is a problem you can delegate — to a teammate, to a function, or to an AI — and still verify.

Problem



Model

**Compute**

Analyze



Validate



Improve



One function at a time — test as you go

1 · SKELETON FIRST

```
def stopping_distance(v, t_r, a):  
    pass    # not written yet  
  
def is_safe(d_stop, d_avail):  
    pass
```

Name every task before writing any. The plan of the program exists — and runs — before the details do.

2 · FILL ONE · TEST IT

```
def stopping_distance(v, t_r, a):  
    return v*t_r + v**2/(2*a)  
  
d = stopping_distance(20,1.0,7.0)  
  
# expect ≈ 48.57 ✓ by hand
```

One function at a time, each held to a hand-checked case before you trust it.

3 · THEN ASSEMBLE

```
while stopping_distance(v + step,  
                        t_r, a) <= d_avail:  
    v = v + step
```

Only verified pieces enter the search. When something misbehaves now, you know where to look.

This is incremental development — and it is how you will direct AI. In the laboratory you hand AI one well-defined piece at a time — a name, its inputs, its expected output — and verify each before the next. A stub is a promise; a tested function is a kept one.

Problem



Model



Compute



Analyze



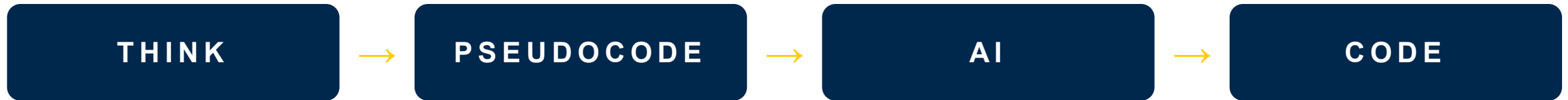
Validate



Improve



The working method



NOT THIS

Problem → “AI, write the solution”

You receive code you cannot judge, implementing an algorithm you never designed, answering a question you never made precise. When it fails — and eventually it will — you have nothing to debug against.

THIS

Understand → describe → pseudocode → AI critiques your logic → you decide → AI helps express it in code → you test.

The algorithm is yours. AI helps with expression — syntax, phrasing, mechanics. Every suggestion meets your judgment before it meets your program.

Pseudocode is what you hand to AI — so you hand over the typing, never the thinking.





AI as coding companion — who does what



AI MAY...

- explain concepts and syntax on demand
- critique your pseudocode and ask guiding questions
- translate your pseudocode into code
- explain error messages in plain language
- suggest debugging strategies and test cases
- help organize code into functions



THE ENGINEER MUST...

- own the engineering problem and the model
- own the algorithm — what should this compute?
- read and explain any code before using it
- test with deliberate cases, including boundaries
- validate and check physical reasonableness
- accept, modify, or reject — and approve the result

You do not need to memorize every programming detail. **You DO need to understand what you are asking the computer to do.** AI assists. Human verifies. Engineer decides.

Problem



Model



Compute



Analyze



Validate



Improve