Prompt Engineering Fundamentals

Copilot Fundamentals | Beginner

Session 03 of 18 | Duration: 3 hours (1hr content + 2hr lab)

Agenda & Time Plan

Time	Activity	Duration
0:00	Why Prompts Matter	5 min
0:05	Anatomy of a Good Prompt	12 min
0:17	The Neighboring Tabs Effect	5 min
0:22	Comment-Driven Development	8 min
0:30	Iterative Prompting	8 min
0:38	Anti-Patterns	8 min
0:46	Live Demo: Same Task, Three Strategies	8 min
0:54	Prompt Libraries & Team Patterns	4 min
0:58	Custom Instructions Preview	3 min
1:01	Wrap-Up & Lab Preview	2 min
1:03	☕ Break	10 min
1:13	Lab Start	—
3:00	Wrap-up	—

Breaks: 10-minute break between trainer content and lab. Additional 5-minute breaks at trainer's discretion.

Why Prompts Matter

The formula:

$$\text{Output Quality} = \text{Model Capability} \times \text{Prompt Quality} \times \text{Available Context}$$

- You **can't** control model capability
- You **CAN** control prompt quality and context
- These are the two levers for getting better results

> Prompting is clear communication. State the work, the context, and the limits.

The Same Task: Weak vs. Strong

Weak prompt:

```
Write a function to process data
```

Result: Copilot has to guess about the data, behavior, and output.

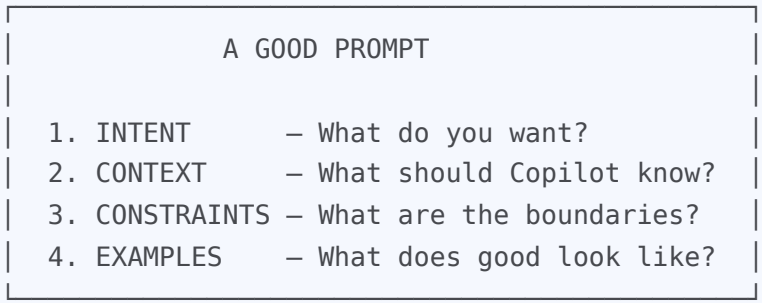
Strong prompt:

```
Write a Python function called process_csv_row that takes
a dictionary representing a CSV row, validates that 'email'
and 'name' fields are present and non-empty, normalizes the
email to lowercase, and returns a cleaned dictionary.
Raise ValueError for invalid rows.
```

Result: The request gives Copilot behavior and failure rules to follow.

The prompt supplies the missing detail.

Anatomy of a Good Prompt



You don't need all four every time. But when a prompt isn't working, this tells you what's missing.

Component 1: Intent: Be Specific

❌ Weak Intent	Problem	✅ Better Intent
"Help with this code"	Help how?	"Explain why this returns None for empty lists"
"Make it better"	Better how?	"Refactor to flatten nested if-else into early returns"
"Write a test"	For what?	"Write a pytest test verifying 20% discount for orders over $100"
"Fix the bug"	What bug?	"Fix the off-by-one error — returns 11 items when page_size is 10"

Specify the **what**, the **action**, and the **scope**.

Component 2: Context: Show, Don't Tell

Context strategies (best to worst):

1. 📂 **File references** — `#file:services/user_service.py`
2. 🔍 **@workspace** — let Copilot search the project
3. 📝 **Explicit in prompt** — "This is a Django REST API using PostgreSQL"
4. 📑 **Open relevant tabs** — the "neighboring tabs" effect

```
Based on the patterns in #file:services/user_service.py,
create a new service method for order cancellation.
```

Show Copilot a relevant example and ask it to follow the pattern. That is usually more useful than describing the style.

Component 3: Constraints

Category	Example
Dependencies	"Use only the standard library"
Performance	"Must handle 10,000 items/sec — use batch processing"
Compatibility	"Must work on Python 3.8+ (no walrus operator)"
Style	"Follow PEP 8, use type hints, functions under 20 lines"
Security	"Never log the API key. Use environment variables."
Output format	"Return JSON with keys 'status' and 'data'"

Without constraints, Copilot makes its own choices about libraries, patterns, and style.

Component 4: Examples

```
Write a function to format phone numbers.
Input examples:
  "5551234567"    → "(555) 123-4567"
  "+15551234567"  → "(555) 123-4567"
  "555-123-4567"  → "(555) 123-4567"
```

Examples are especially powerful for:

- Input/output transformations
- Edge case specification
- Pattern matching across formats

The Neighboring Tabs Effect

Copilot reads your **open tabs** to gather context — not just the current file.

How to use this:

- Open files that show the patterns you want Copilot to follow
- Open test files when generating tests (so Copilot matches the framework)
- Open type definitions when generating functions

The effect is real: Opening a related file can change Copilot's suggestions even if you don't reference it explicitly.

> Open the files that show the pattern you want before you request a suggestion.

Comment-Driven Development

Write comments **first** — let Copilot implement:

```
# Validate email format using regex
# Return True for valid, False for invalid
# Handle edge cases: empty string, missing @, multiple @
def validate_email(email: str) -> bool:
```

Why it works:

- Comments provide intent + constraints in one place
- Copilot sees them as specification, not just documentation
- Comments stay in the code as documentation for humans too

Iterative Prompting

Don't expect perfection on the first try. Build iteratively:

```
1. Write a user registration function
   → Too basic, no validation

2. Add email validation and password strength checks
   → Better, but no error messages

3. Return specific error messages for each validation failure
   → Good, but not using our error format

4. Use the AppError class from #file:errors.py
   → ✅ Matches project patterns
```

After each response, name the specific behavior that is still missing.

Anti-Patterns to Avoid

Anti-Pattern	Problem	Fix
Vague prompt	"Make it work"	Specify what "working" looks like
Over-specified	50-line prompt for a 5-line function	Match prompt length to task complexity
No context	Expects Copilot to know your project	Use #file, @workspace, or inline context
Copy-paste blindly	Accept without reading	Always review and test
One-and-done	Give up if first result is wrong	Iterate — refine the prompt

Bad output? Check three things: unclear intent, missing context, or no constraints.

Prompt Libraries & Team Patterns

Create reusable prompt templates your team shares:

```
## Generate API Endpoint
Create a {METHOD} endpoint at {PATH} that:
- Accepts: {INPUT_SCHEMA}
- Validates: {VALIDATION_RULES}
- Returns: {OUTPUT_SCHEMA}
- Error handling: Use AppError from src/errors/
- Follow patterns in #file:src/routes/users.ts
```

Store these in `.github/prompts/` (covered in Session 06).

> Write a good template once, reuse it everywhere — just like a good function.

Lab Time

Hands-on Exercises (2 hours)

Lab Overview

Exercise	Topic	Duration
1	Prompt Challenge Rounds (5 timed challenges)	40 min
2	Comment-Driven API Development	40 min
3	Context Manipulation Experiments	20 min
4	Prompt Cheat Sheet	20 min

Deliverable: A prompt engineering cheat sheet + a fully prompted REST API endpoint.

Key Takeaways

1. **Four components** of a good prompt: Intent, Context, Constraints, Examples
2. The **neighboring tabs effect** means your open files shape suggestions
3. **Comment-driven development** turns comments into specifications
4. **Iterative prompting** works better than one-shot attempts. Refine progressively.
5. Avoid anti-patterns: vague, over-specified, no context, copy-paste blindly
6. Build **prompt libraries** for team consistency

Questions?

Take a moment to discuss with your trainer

Next Up

Session 04: GitHub Copilot in the CLI

Thank you!