LESSON 7

Prompt engineering for programmers

> **LESSON GOAL**
>
> Learn to write requests the model understands correctly first time, and build your own library of proven templates.

What you will learn in this lesson

- The four part structure of a request and why it works.
- Working with examples and with step by step reasoning.
- The power of negative instructions.
- Five ready made templates for immediate use.

7.1 The structure of a request

Most bad answers arise from an incomplete request. The proven structure has four parts and can be used in a chat, in an editor and with an agent.

Part	Answers the question	Example
Context	What environment are we in?	Python 3.12, FastAPI, PostgreSQL, an existing project
Goal	What should come into being?	A function returning jobs past their deadline
Constraints	What must not or must happen?	No new dependencies, do not touch auth
Format	What should the answer look like?	The function code only, no explanation

The Format part is left out most often and yet it saves the most time. Without it the model adds paragraphs of explanation you do not need, or conversely returns bare code where you wanted a comparison of options.

7.2 Examples are stronger than description

When the result is hard to describe in words, show an example of input and output. The model follows a pattern far more reliably than a description, especially with formatting and data transformations.

> **Description versus example**

```
WEAK:
Format the address nicely.

STRONG:
Convert the input to the output in exactly this shape:

input:  {'street':'Main 12','town':'Didcot','postcode':'OX11 8AB'}
output: 'Main 12, Didcot OX11 8AB'

input:  {'street':'Main 12','town':'Didcot','postcode':None}
output: 'Main 12, Didcot'

input:  {'street':None,'town':'Didcot','postcode':'OX11 8AB'}
output: 'Didcot OX11 8AB'
```

Three examples covering missing values as well say more than a paragraph of rules. On top of that they immediately become test cases.

7.3 Step by step reasoning

With more complex tasks it pays to let the model think first and answer afterwards. The difference in quality tends to be large, especially with design and with debugging.

> **Forcing a procedure**
>
> ```
> Before you start writing code:
> 1. List the assumptions you are making about the input.
> 2. Name the edge cases that need handling.
> 3. Describe the solution in three sentences.
>
> Only then write the code. If something is missing from the request,
> ask instead of guessing.
> ```

The last sentence matters. Without it the model would rather supply a probable assumption than ask, and you will only find out about it during testing.

7.4 Negative instructions

What the model must not do is as important as what it should do. Without negative instructions the model tends to take the initiative in ways that do not suit you.

Problem	Negative instruction
It installs new libraries	Use only the standard library and what is already in the project
It rewrites surrounding code	Change only the body of this function, nothing else
It adds explanation	Return code only, with no comment before or after
It simplifies the request	Do not ignore edge cases even if they complicate the solution
It changes the code style	Preserve the existing formatting and naming conventions

7.5 Five templates for immediate use

Copy the following templates and adapt them to your stack. They save you from inventing requests over and over and, above all, ensure that you forget nothing.

> **Template 1: A new function**
>
> ```
> CONTEXT: <language, framework, versions, where in the project>
> GOAL: <what the function should do, one paragraph>
> INPUT: <types and an example>
> OUTPUT: <type and an example>
> EDGE CASES: <empty, null, extreme values>
> CONSTRAINTS: <no new dependencies, style, performance>
> FORMAT: the function code and its tests only
> ```

Template 2: Debugging an error

```
ERROR: <full stack trace>
CODE: <the function where it arises, plus the caller>
EXPECTED BEHAVIOUR: <what should have happened>
ACTUAL BEHAVIOUR: <what happened, how often, since when>
ALREADY TRIED: <what you have ruled out>
FORMAT: three hypotheses by probability,
for each how I verify it quickly. Do not fix anything yet.
```

Template 3: Code review

```
CODE: <what is to be checked>
CONTEXT: <what it serves, who calls it, what data it processes>
REVIEW INTENT: security, edge cases, readability, performance
FORMAT: a list of findings ordered by severity,
for each a specific proposed fix. No rewrite of the whole file.
```

Template 4: Deciding between options

```
SITUATION: <what I am solving, what constraints I have>
OPTIONS: A) <...> B) <...>
CRITERIA: <complexity, performance, maintainability, cost, speed of delivery>
FORMAT: a comparison table, then a recommendation,
then the conditions under which the recommendation would stop holding.
```

Template 5: Explaining someone else's code

```
CODE: <block>
MY LEVEL: <I know the language, I do not know this framework>
I WANT TO KNOW:
- the intent in three sentences
- the assumptions about input the code silently makes
- what breaks if I change <specific thing>
FORMAT: structured, no line by line description
```

7.6 Iteration: what to do when the answer does not fit

The first answer is rarely the last. The difference between an experienced and a beginning user lies in how they react to an unsuccessful result.

Problem with the answer	Bad reaction	Good reaction
Overly complex solution	Simplify it	State a constraint: no classes, under thirty lines
An edge case is missing	Add it	List what is missing and have the whole function rewritten
A wrong assumption	No, differently	Correct the assumption in the request and ask again
An invented function	This does not exist	Paste an excerpt of the documentation and have it corrected
A sprawling answer	Be specific	Set the format: code only, no explanation

The shared principle: do not repair the answer, repair the request. Most bad answers are correct answers to a badly posed question.

7.7 Terms and abbreviations in detail

Prompt engineering is not a collection of tricks. It is a way of turning an unclear intention into a precise specification that can be verified. The key concepts below help to separate context, goal, constraints and required output.

Term or abbreviation	Detailed explanation and practical meaning
Prompt	The complete request handed to a model. It may contain a question, context, code, examples, constraints and the required output format. A quality prompt is not a magic sentence. It is a precise specification of the problem with enough relevant information and a clear goal.
Prompt engineering	The discipline of formulating inputs for language models so that results are more accurate, repeatable and easier to control. In programming it is mainly about the right context, splitting up tasks, examples, constraints, verifiable conditions and a suitable output format.
Context	The information a model needs in order to understand the situation. It may include the language, the framework version, the architecture, relevant files, existing conventions and earlier decisions. Context should be complete for the given problem but not flooded with irrelevant data.
Goal	An unambiguous description of what should exist after completion or how the system should behave. A good goal is measurable. For example the function returns jobs past their deadline sorted from the oldest is more precise than make an overview of jobs.
Constraints	Rules that narrow the space of possible solutions. They include a ban on new dependencies, a maximum scope of change, a required library version or files the agent must not touch. Constraints prevent the model from choosing a technically possible but project wise unsuitable solution.
Output format	A precise requirement for the shape of the answer, for example code only, a table of options, JSON or a change plan. Format saves time and reduces ambiguity. If the output is to be processed by another program, it should be defined even more precisely, ideally by a schema.
Few shot prompting	A technique in which the prompt contains several example pairs of input and correct output. From the examples the model orients itself in the required pattern, style or transformation. Examples are especially effective where a rule is hard to describe in words alone.
Zero shot prompting	A request without example pairs, using instructions only. It works well for common and clearly defined tasks. If the required format or business rule is unusual, a few examples often give a more reliable result than another paragraph of general description.
Input and output	Input is the data or conditions a function or model receives. Output is the result. Precise examples of inputs and outputs act both as a specification and as the basis for future tests, because they make concrete what the system is supposed to do.
Schema	A formal description of the structure of data. It may fix field names, data types, required fields and constraints. A schema is more precise than free text and is especially useful when AI is to generate JSON, a database model or an API interface.
Negative instruction	An explicit description of what the model must not do, for example do not add dependencies or do not change auth.py. Negative instructions matter with agents, because without them the model may take the initiative and widen the intervention beyond the original intent.
Assumption	Information that a solution treats as true even though it was never stated. Invisible assumptions are a frequent source of errors. A good prompt asks the model to list uncertain assumptions and, at critical points, to ask rather than guess.
Iteration	Repeated refinement of the request and the result. Effective iteration should not be a series of random additions. It is better to name what does not fit in the answer, adjust the original specification and have a consistent new version produced.
Review prompt	A request aimed at checking code that already exists. It should define exactly what is to be checked, for example correctness, security, performance or conformity with the architecture. A general check this often leads to shallow agreement instead of systematic analysis.