

Tokens, teaching a computer to read

A language model never sees your words. It sees a list of integers. Everything it knows, every answer it gives, and every bill you pay for it is counted in those integers, which are called tokens. This chapter shows exactly where they come from. By the end you will have built the same kind of tokenizer GPT-4 uses, trained it on Shakespeare, and measured it against OpenAI's real ones.

Why this matters

Tokenization looks like boring plumbing, and that is why it causes so much confusion. A surprising number of "the model is dumb" moments are really tokenizer moments. Karpathy opens his tokenizer lecture with a list of them: models that struggle to spell, that are worse at arithmetic than they should be, that handle Python indentation badly, that do worse in Polish or Korean than in English, and that warn you about a trailing space in your prompt. Each one traces back to how text was chopped into tokens before the model ever saw it.

You will meet tokens in three places in your work.

First, cost and limits. API prices are per token, input and output. Context windows are measured in tokens. When an agent reads a long file, calls tools, and keeps a conversation history, all of that is tokens, and it is the main thing you pay for and the main thing that runs out. If you can estimate tokens in your head, you can estimate cost and you can tell when an agent is about to forget its own instructions.

Second, model behavior. The model predicts the next token, not the next letter or word. When you understand what the tokens are, a lot of odd behavior becomes predictable instead of mysterious. A model asked how many letters are in a word is being asked about something it mostly cannot see directly.

Third, security. The text a filter inspects and the token sequence a model consumes are two different views of the same input. Rare or strange tokens can make models behave erratically, and the gap between "string the guardrail checked" and "tokens the model reacted to" is one of the places red teamers look. You do not need that today, but you will in the AI Security course, and it starts here.

The idea in plain words

A tokenizer is a translator between text and a list of integers. It has a fixed vocabulary, a big lookup table where every entry is a piece of text with an ID number. To encode, you cut the text into pieces from the vocabulary and write down their IDs. To decode, you look the IDs up and glue the pieces back together. The model only ever works with the IDs.

The whole design question is what the pieces should be.

One option is single characters. The sentence "To be, or not" becomes 13 tokens, one per character. The vocabulary is tiny, so it is easy, but the sequences get long. A model has a fixed context window, so long sequences mean it sees less text at once, and every extra token costs compute.

Another option is whole words. Now "To be, or not" is about five tokens, which is great, until someone types a word that is not in the table: a typo, a new product name, a Polish word, a variable name in code. The model has no ID for it. Word vocabularies also explode, since "run", "runs", "running" and "runner" all need separate entries.

Byte-pair encoding, or BPE, is the compromise almost everyone uses. You start with the smallest possible pieces, the 256 possible byte values, so any text in any language can always be encoded. Then you look at a big pile of training text, find the pair of neighboring pieces that appears most often, and glue that pair into a new piece with a new ID. Repeat that thousands of times. Common things like "the" and "and" end up as single tokens. Rare things still work, they just get split into smaller pieces.

Here is a concrete example from the lab. Training on the first 200,000 characters of Shakespeare, the very first merge BPE chooses is a space followed by the letter t, which appears 4,336 times. The second merge is "h" plus "e", which appears 3,484 times. The ninth merge glues the two earlier results together: " t" plus "he" becomes " the". After 768 merges, the sentence "To be, or not to be, that is the question." comes out as 15 tokens: "To", " be", ",", " or", " not", " to", " be", ",", " that", " is", " the", " qu", "est", "ion", ".". Common words are whole. The rarer word "question" is built from three pieces.

How it actually works

There are four layers, and it helps to keep them separate in your head.

Layer 1: characters become Unicode code points

A Python string is a sequence of Unicode code points. Each character has a number. "H" is 72, "i" is 105, a space is 32, and the Polish letter "ń" is 324. There are well over a hundred thousand code points defined, and the standard keeps growing, so using code points directly as tokens would give you a huge and unstable vocabulary.

Layer 2: code points become UTF-8 bytes

UTF-8 is the standard way to store code points as bytes. Plain English letters take one byte each. Many accented letters take two. Most Chinese, Japanese and Korean characters take three, and emoji take four. In the lab, "Hi ń" is four characters but five bytes: 72, 105, 32, 197, 132. The "ń" needs the pair 197, 132.

Bytes are attractive because there are exactly 256 of them. If your base vocabulary is the 256 bytes, there is no such thing as an unknown character. Anything a computer can store as text can be encoded. This is called byte-level BPE, and it is what GPT-2 introduced for OpenAI's models.

Layer 3: pre-splitting with a regular expression

Before any merging happens, real tokenizers chop text into chunks with a regular expression, and merges are only allowed inside a chunk. The chunks are roughly: a word with an optional leading space, a run of up to three digits, a run of punctuation, and runs of whitespace.

Why bother? Without it, BPE would happily learn tokens like "dog." or "e, t" because those byte pairs happen to be frequent in the training text. You would waste vocabulary on accidents of punctuation, and the same word would get different tokens depending on what followed it. The pre-split keeps word

pieces clean. You can see one consequence in the token lists: almost every word token includes its leading space, as in " the" and " be". The space belongs to the word that follows it.

The pattern our library uses is the one from OpenAI's GPT-4 tokenizer, published in the tiktoken source. The library's test suite checks that our regex chunks a tricky sample exactly the way tiktoken's does.

Layer 4: byte-pair merges

Training works like this:

1. Split the training text into chunks with the regex, and turn each chunk into a list of byte IDs from 0 to 255.
2. Count every pair of neighbors across all chunks.
3. Take the most frequent pair, say (32, 116), which is space then "t". Give it the next free ID, 256.
4. Replace every occurrence of that pair with 256.
5. Repeat until the vocabulary reaches the size you want.

Each merge makes the text a little shorter. The list of merges, in the order they were learned, is the tokenizer. Nothing else needs to be saved.

Encoding new text replays the merges. For each chunk, start from bytes, look at all neighbor pairs, and apply whichever pair was learned earliest. Repeat until no learned pair is present. Using the learned order matters: it guarantees the encoder reproduces the same decisions training made. tiktoken calls this order the merge rank.

Decoding is the easy direction. Every token ID maps to a fixed string of bytes. Concatenate the bytes and decode them as UTF-8.

Measuring a tokenizer

The useful number is compression, how many bytes of text each token covers on text the tokenizer has not seen. In the lab, on 20,000 bytes of held-out Shakespeare:

- Raw bytes, vocabulary 256: 1.00 bytes per token, by definition.
- Our BPE with 512 entries: 1.92.
- Our BPE with 1,024 entries: 2.31.
- GPT-2, vocabulary 50,257: 3.07.
- GPT-4's cl100k_base, about 100,000 entries: 3.49.
- GPT-4o's o200k_base, about 200,000 entries: 3.57.

Two lessons come out of those numbers. More vocabulary buys more compression, and the returns shrink. Going from 256 to 512 entries almost doubled compression. Going from 100,000 to 200,000 gained about two percent on this English text. The large modern vocabularies are mostly buying better coverage of other languages and of code, which you will see in a moment.

Why the vocabulary size is a real tradeoff

If more vocabulary means shorter sequences, why not use a million tokens? Because the vocabulary is not free inside the model. Each token ID needs an embedding, a learned vector, at the input, and the output layer has to produce a score for every token at every step. In GPT-2 small, the embedding table alone is 50,257 rows of 768 numbers, about 38.6 million parameters, close to a third of the 124 million

total. Rare tokens are also a problem: if a token almost never appears in the training data, its embedding barely gets trained. So vocabulary size is a design choice that trades sequence length against model size and token quality. Most frontier labs currently land somewhere between about 100,000 and 260,000 entries.

The four surprises in the lab output

The lab runs the same strings through three of OpenAI's production tokenizers. Each result explains a behavior you have probably seen.

Spelling. In all three tokenizers, " strawberry" with a leading space is a single token. Without the space, it is three: "st", "raw", "berry". When you ask a model how many letter r's are in strawberry, the model is looking at one or three opaque IDs, not ten letters. It has to have learned, from training data, what letters are inside each token. Modern models often get this right now, partly through training and partly by reasoning step by step, but the difficulty comes from here.

Languages. "Good morning, how are you?" is 7 tokens in all three tokenizers. The Polish equivalent, "Dzień dobry, jak się masz?", is 16 tokens in GPT-2, 11 in cl100k and 10 in o200k. In GPT-2, "ń" and "ę" are not even whole tokens; each one is two raw bytes, which is why the decoded pieces show replacement characters. Same meaning, up to more than twice the price and twice the context used. Newer vocabularies spent their extra entries largely on non-English text, and the gap shrinks, but it does not close.

Numbers. GPT-2 tokenizes "127 + 4 = 131" as "127", "+", "4", "=", "131". The GPT-4 era tokenizers split digits into runs of at most three and keep the space separate, so 4 becomes " " and "4". That is a deliberate choice to make numbers more consistent, since a number's tokens no longer depend on whether a space came before it. Even so, a model doing arithmetic is operating on chunks like "131", not on digits, which is part of why long arithmetic without tools is unreliable.

Code. Eight spaces of Python indentation followed by "return y" is 9 tokens in GPT-2, because each space is its own token, and 3 tokens in GPT-4's tokenizer, which has tokens for runs of spaces. Karpathy points out that this change made Python denser, so the same context window holds more code. It is one concrete reason GPT-4 got better at code.

Build it

The lab is `lab.py` in this chapter's folder. The tokenizer code lives in the shared library at `labs/tutorlab/tokenizers.py`, so later chapters can reuse it, and it is covered by tests in `labs/tests/`. Run the lab with:

```
python lab.py
```

It runs offline in a few seconds. Here are the important pieces.

Counting pairs and merging

```
def _merge(ids, pair, new_id):
    out, i = [], 0
    while i < len(ids):
```

```

        if i + 1 < len(ids) and ids[i] == pair[0] and ids[i + 1] == pair[1]:
            out.append(new_id)
            i += 2
        else:
            out.append(ids[i])
            i += 1
    return out

```

This walks the list once. Whenever it sees the target pair it writes the new ID and skips two places, otherwise it copies one ID and moves on. Note that it scans left to right, so in a run like "aaa" with pair (a, a) it produces the new token followed by a single "a". That is the standard behavior.

Training

```

freq = {}
for c in self._re.findall(text):
    freq[c] = freq.get(c, 0) + 1
chunks = [list(c.encode('utf-8')) for c in freq]
weights = list(freq.values())
for k in range(vocab_size - 256):
    counts = {}
    for ids, w in zip(chunks, weights):
        for pair in zip(ids, ids[1:]):
            counts[pair] = counts.get(pair, 0) + w
    pair = max(counts, key=lambda p: (counts[p], -p[0], -p[1]))
    new_id = 256 + k
    chunks = [_merge(ids, pair, new_id) if len(ids) > 1 else ids for ids in chunks]
    self.merges[pair] = new_id
    self.vocab[new_id] = self.vocab[pair[0]] + self.vocab[pair[1]]

```

Three details are worth noticing. The regex split happens first, so pairs never cross chunk boundaries. Each distinct chunk is processed once and weighted by how often it occurs, which gives the same answer as processing every occurrence, only much faster, since " the" may appear thousands of times. And ties are broken by the smaller pair, so training is deterministic and you get the same tokenizer every run.

The new token's bytes are just the bytes of its two parents glued together. That is why decoding is trivial.

Encoding with merge ranks

```

def _encode_chunk(self, chunk_bytes):
    ids = list(chunk_bytes)
    while len(ids) ≥ 2:
        counts = _pair_counts(ids)
        pair = min(counts, key=lambda p: self.merges.get(p, float('inf')))
        if pair not in self.merges:
            break

```

```
ids = _merge(ids, pair, self.merges[pair])
return ids
```

Among the pairs present, pick the one with the smallest merge ID, which is the one learned earliest. If none of the pairs was ever learned, the chunk is fully encoded. This is the same rule tiktoken uses with its ranks.

What the lab checks

The `check()` function at the bottom asserts real behavior, not trivia:

- Decoding the encoding gives back exactly the original text, including Polish and an emoji the tokenizer never saw in training.
- The 1,024 entry tokenizer beats 2 bytes per token on unseen text, and compression rises at every step of the vocabulary sweep, starting from exactly 1.00 for raw bytes.
- The first merge is the most frequent pair.
- No learned token contains a space in its middle, which proves the regex pre-split is doing its job.
- The production facts this chapter relies on are true on your machine: GPT-2 has 50,257 tokens, "strawberry" is one token and "strawberry" is three in o200k, Polish costs more in GPT-2 than in o200k, and all three production tokenizers compress better than ours.

If any of those fail, the lab stops with an error instead of printing CHECK PASSED.

Expected output (abridged)

Part 3: training BPE (first 12 merges shown)

```
merge  1:    b' ' + b't'    → b' t'      seen 4,336x
merge  2:    b'h' + b'e'    → b'he'    seen 3,484x
merge  9:    b' t' + b'he'   → b' the'   seen 1,783x
... 768 merges, final vocabulary 1024
```

Part 4: encode and decode

```
pieces ['To', ' be', ',', ' or', ' not', ' to', ' be', ',', ' that', ' is', ' the', ' qu', 'est',
'ion', '.']
```

```
held-out text: 20,000 bytes → 8,641 tokens = 2.31 bytes per token
```

Part 5: production tokenizers (tiktoken)

```
gpt2          vocabulary 50,257   held-out Shakespeare 3.07 bytes per token
cl100k_base    vocabulary 100,277  held-out Shakespeare 3.49 bytes per token
o200k_base     vocabulary 200,019   held-out Shakespeare 3.57 bytes per token
```

Part 6: same meaning, different price

```
gpt2          English 7 tokens, Polish 16 tokens
o200k_base     English 7 tokens, Polish 10 tokens
```

CHECK PASSED

Try this

The bottom of the lab has three experiments. Predict the answer before you run each one, and write the prediction down. The point is to find out whether your mental model is right.

1. Add 2,048 to the vocabulary sweep. Will the gain from 1,024 to 2,048 be bigger or smaller than the gain from 512 to 1,024?
2. Train on `TEXT.upper()`. What happens to "To be, or not" afterwards, and why?
3. Add a sentence in Polish or German to the samples. Which production tokenizer is cheapest for it?

Common mistakes and misconceptions

"A token is a word." Sometimes it is, often it is not. In the lab, "question" is three tokens, "strawberry" is one, and a single Polish letter can be two. The usable rule of thumb is a measurement, not a definition: on our English text, GPT-4o's tokenizer averaged about 3.6 bytes per token. Code, numbers and other languages differ, so count instead of guessing when it matters.

"The model reads letters." It reads token IDs. Whatever it knows about the letters inside a token, it had to learn from data. That is why spelling, counting characters, reversing strings and rhyming can be harder than they look, and why asking the model to spell a word out first often helps: spelling it out turns letters into separate tokens it can see.

"The tokenizer is learned along with the model." It is trained separately, before the model, on its own data, and then frozen. The model's training never changes it. You can swap in a different tokenizer only by retraining or heavily adapting the model, because every embedding row is tied to a specific token ID.

"Bigger vocabulary is always better." Bigger vocabularies shorten sequences, but they grow the embedding and output layers and leave rare tokens undertrained. The returns also shrink, as the sweep and the 100,000 versus 200,000 comparison show.

"Token counts are the same across models." They are not. The same Polish sentence was 16, 11 and 10 tokens in three OpenAI tokenizers, and other vendors use other tokenizers again. Budget and price per model, using that model's own counter.

"Glitch tokens are a myth." They were real. Researchers found tokens such as "SolidGoldMagikarp" in the GPT-2 and GPT-3 vocabulary that appeared in the tokenizer's training data but almost never in the model's, so the model never learned what to do with them, and asking about them produced bizarre output. Karpathy covers this in the lecture linked below.

In the real world

OpenAI ships its tokenizers as the open source tiktoken library. As of tiktoken 0.14.0 (checked October 10, 2026), its model table maps GPT-2 to the "gpt2" encoding, GPT-4 and GPT-3.5 Turbo to "cl100k_base", and GPT-4o, GPT-4.1, o1 and GPT-5 to "o200k_base". You can see that table in the tiktoken source on GitHub. tiktoken does exactly what our code does, regex pre-split followed by rank-ordered byte-pair merges, but it is written in Rust and is far faster.

Meta's Llama 3 moved to a 128,000 token vocabulary, which Meta's launch post says "encodes language much more efficiently". Llama 2 had used a 32,000 entry BPE vocabulary built with Google's

SentencePiece library, according to the Llama 2 paper. SentencePiece is another popular implementation that works on Unicode text directly and can fall back to bytes for rare characters.

Anthropic does not publish a downloadable tokenizer for its current Claude models. Instead it provides a token counting endpoint in the API, so you can ask how many tokens a message will use before you send it. As of October 2026 that is the reliable way to count Claude tokens.

In agent tools like Claude Code, Codex and Hermes, token counts drive everything you care about operationally: when the context gets compacted, how much a long session costs, and how much of a big file the model can actually read. When an agent "forgets" something from earlier, the cause is usually that the conversation crossed a token limit and was summarized or truncated.

To explore tokenizers interactively, the Tiktokenizer web app shows how any text splits under several real tokenizers. Karpathy uses it throughout his lecture.

Go deeper

Watch these after you have run the lab. They will make much more sense once you have trained your own tokenizer.

- Andrej Karpathy, "Let's build the GPT Tokenizer". This is the full version of this chapter, over two hours, building minbpe live.
- Start at 4:49 (<https://www.youtube.com/watch?v=zduSFxRajkE&t=289>) for his list of model failures caused by tokenization, including spelling, then keep going through 5:27 (<https://www.youtube.com/watch?v=zduSFxRajkE&t=327>) for SolidGoldMagikarp.
- Jump to 22:30 (<https://www.youtube.com/watch?v=zduSFxRajkE&t=1350>) for byte-pair encoding itself, where he builds the merge loop step by step.
- 68:20 (<https://www.youtube.com/watch?v=zduSFxRajkE&t=4100>) walks through the GPT-4 regex pattern, including the whitespace rule that made Python denser.
- Andrej Karpathy, "Deep Dive into LLMs like ChatGPT". At 12:04 (<https://www.youtube.com/watch?v=7xTGNNLPyMI&t=724>) he introduces tokenization as part of the whole pipeline and shows GPT-4's tokenizer on screen. Good for seeing where tokens sit in the bigger picture before the next chapters.
- Papers and code: the GPT-2 paper (section 2.2 explains byte-level BPE), Sennrich and colleagues' 2015 paper that brought BPE to neural machine translation, Karpathy's minbpe repository, and the SolidGoldMagikarp write-up on LessWrong. All are linked in Sources.

Explain it back

Without looking at this page, explain to someone who has never written code why a model can know what a strawberry is but still miscount the letter r in it. Use the words token, vocabulary and merge, and mention what the leading space has to do with it. Aim for about a minute. Explain it to a study partner or record your own explanation.

Check yourself

The full check is in `quiz.json`: four multiple choice questions, two predict-the-output questions, and two explain questions with rubrics. The multiple-choice questions below check recall; the downloadable quiz also includes prediction and explanation rubrics. The questions cover:

1. Why byte-level BPE can encode any text.
2. What the regex pre-split prevents.
3. The order merges are applied in during encoding.
4. Why vocabulary size is a tradeoff.
5. Predicting compression when the vocabulary grows.
6. Predicting the tokens for a number in a GPT-4 era tokenizer.
7. Explaining the cost gap between English and Polish.
8. Explaining why spelling is hard for language models.

Sources

- Andrej Karpathy, "Let's build the GPT Tokenizer" (YouTube): <https://www.youtube.com/watch?v=zduSFxRajkE>
- Andrej Karpathy, "Deep Dive into LLMs like ChatGPT" (YouTube): <https://www.youtube.com/watch?v=7xTGNNLPyMI>
- Karpathy, minbpe (the reference this chapter's library follows): <https://github.com/karpathy/minbpe>
- OpenAI, tiktoken (encodings and model table, version 0.14.0 checked 2026-10-10): <https://github.com/openai/tiktoken>
- Radford et al., "Language Models are Unsupervised Multitask Learners" (GPT-2): https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf
- Sennrich, Haddow and Birch, "Neural Machine Translation of Rare Words with Subword Units" (2015): <https://arxiv.org/abs/1508.07909>
- Byte-pair encoding background and history: https://en.wikipedia.org/wiki/Byte-pair_encoding
- UTF-8: <https://en.wikipedia.org/wiki/UTF-8>
- Meta, "Introducing Meta Llama 3" (128K token vocabulary): <https://ai.meta.com/blog/meta-llama-3/>
- Touvron et al., "Llama 2: Open Foundation and Fine-Tuned Chat Models" (32K SentencePiece BPE): <https://arxiv.org/abs/2307.09288>
- Google, SentencePiece: <https://github.com/google/sentencepiece>
- Anthropic, token counting (checked 2026-10-10): <https://docs.claude.com/en/docs/build-with-claude/token-counting>
- Rumbelow and Watkins, "SolidGoldMagikarp (plus, prompt generation)": <https://www.lesswrong.com/posts/aPeJE8bSo6rAFoLqg/solidgoldmagikarp-plus-prompt-generation>
- Tiktokenizer, interactive tokenizer viewer: <https://tiktokenizer.vercel.app/>
- Hugging Face LLM course, byte-pair encoding tokenization: <https://huggingface.co/learn/llm-course/chapter6/5>
- OWASP GenAI, LLM01 Prompt Injection (for the security thread that starts here): <https://genai.owasp.org/llmrisk/llm01-prompt-injection/>
- Tiny Shakespeare dataset (Karpathy, char-rnn): <https://github.com/karpathy/char-rnn>