

# Lab 02 — Inspect Tokenization

Gavin · AI-assisted course design

## Contents

|    |                                                      |   |
|----|------------------------------------------------------|---|
| 1  | Goal and prerequisites .....                         | 3 |
| 2  | Local setup and commands .....                       | 3 |
| 3  | 1. Identify the specimen .....                       | 4 |
| 4  | 2. Predict before inspecting .....                   | 5 |
|    | 4.1 Prose and whitespace .....                       | 5 |
|    | 4.2 Code and identifiers .....                       | 5 |
|    | 4.3 Numbers .....                                    | 5 |
|    | 4.4 Unicode .....                                    | 5 |
| 5  | 3. Collect the observations .....                    | 6 |
| 6  | 4. Inspect special tokens separately .....           | 6 |
| 7  | 5. Separate encoding length from model context ..... | 6 |
| 8  | 6. Interpret and submit .....                        | 7 |
|    | 8.1 Completion criteria .....                        | 7 |
| 9  | More Learning .....                                  | 7 |
| 10 | References .....                                     | 7 |

## 1 Goal and prerequisites

Inspect a real tokenizer and explain what its output does, and does not, tell you about a language model. Compare ordinary prose, code, numbers, whitespace, and Unicode while changing one feature at a time.

Read Lesson 1.2 through the section on tokens and words first. You should be able to distinguish a vocabulary from a sequence of token IDs. Allow approximately 45–75 minutes, including interpretation. Basic familiarity with a Python session is helpful; an LLM study companion can assist with the inspection, but should not invent your results.

This Lab specifies the investigation and required observations. It does not require training, neural-network weights, a GPU, a paid API, or a cloud account. Use a local Python environment with the Hugging Face Tokenizers library and public tokenizer artifacts. Initial software/file downloads require network access; the inspection itself can run locally afterward. Use the supplied harmless strings rather than private messages or credentials.

## 2 Local setup and commands

The downloadable runner and pinned requirements implement the observations below. Save both files in a new local working directory. They were tested with **CPython 3.13.13** and **Tokenizers 0.20.3**, using a CPU. Record predictions before executing the runner.

On macOS or Linux with Python 3.13 installed:

```
python3.13 -m venv .venv
.venv/bin/python -m pip install --no-deps -r tokenizer-requirements.txt
.venv/bin/python inspect_tokenization.py --artifacts tokenizer-artifacts --output
run-01
```

The `--no-deps` installation is deliberate: this runner uses the local Tokenizers backend only, not the optional Hub-client integration. On Windows, create the environment with `py -3.13 -m venv .venv` and replace `.venv/bin/python` with `.venv\Scripts\python.exe` in the other commands.

The initial run retrieves only `tokenizer.json` and `tokenizer_config.json` from the exact revision below. It verifies their SHA-256 hashes before loading them. It never retrieves neural weights, executes remote custom code, or sends the test strings to a service. The repository declares an MIT license; its pinned repository page provides artifact attribution and license metadata.

Open `run-01/manifest.json`, `run-01/results.json`, and `run-01/checks.json` in your text editor. The manifest records software versions, file hashes, pipeline settings, and companion configuration. Results contain escaped inputs, code points, bytes, IDs, display pieces, decoded strings, exact-equality checks, special-token observations, per-ID Unicode decoding, and a measured over-1024-token input. These are your machine observations; compare them with your predictions rather than copying them into an answer without interpretation. If a round trip fails, both strings and their first differing position remain in the results.

After the artifact files exist, repeat without network access into a fresh output directory:

```
.venv/bin/python inspect_tokenization.py --offline --artifacts tokenizer-artifacts
--output run-02
```

Compare `run-01/results.json` and `run-02/results.json`; the measurements should be reproducible for this pinned setup. Manifest timestamps differ. The runner rejects modified artifacts and refuses to overwrite nonempty result directories, preserving unexpected outputs for investigation. Predictions and your written interpretation remain your own work.

### 3 1. Identify the specimen

Use `openai-community/gpt2` from the Hugging Face Hub, pinned to revision `607a30d783dfa663caf39e06633721c8d4cfcd7e`. Its file listing at that revision includes `tokenizer.json`, `vocab.json`, `merges.txt`, and `tokenizer_config.json`. The repository is marked with an MIT license. We use GPT-2 for inspectability, without treating its tokenizer as representative of every modern model.

Download only the tokenizer files you need. The complete `tokenizer.json` is the executable tokenizer specification for this Lab; the separate vocabulary and merge files are useful for inspection. Avoid downloading the repository's much larger model-weight files.

Use `Tokenizer.from_file` to load the local JSON through Hugging Face Tokenizers. The versioned API reference documents this operation and the encode/decode controls used below. Record your installed library version rather than assuming that unversioned documentation describes it.

Create a small run manifest recording:

- Repository identifier and full artifact revision.
- Names and SHA-256 hashes of downloaded files.
- Python and Tokenizers versions, operating system, and run date.
- The tokenizer's model, normalizer, pre-tokenizer, post-processor, decoder, and added-token configuration.
- Vocabulary size, including whether added tokens are counted.
- Padding, truncation, automatic-special-token, and decode settings.

The tokenizer's model here means its segmentation component, such as BPE. It is not a loaded neural language model. Inspecting JSON fields is enough to distinguish those objects.

#### **i** Keep the experiment controlled

Use raw strings, with no pre-splitting into words. Explicitly disable padding and truncation. For the baseline, use `add_special_tokens=False` when encoding and `skip_special_tokens=False` when decoding. These settings address different operations: inserting markers versus removing them during decoding. Preventing automatic insertion does not necessarily disable recognition of a special-token spelling already present in the input.

## 4 2. Predict before inspecting

Write a short prediction for each group below. You do not need to guess numeric IDs. Predict which pairs might have different boundaries or counts, and give your reasoning. Mark uncertainty rather than forcing a confident answer.

For every input, record the exact string in an escaped, unambiguous representation. The following quoted fixtures use JSON-style escapes: `\n` denotes an actual newline, and `\u0301` denotes the combining acute accent. The surrounding quotes are notation, not input characters.

### 4.1 Prose and whitespace

- `"Paint it blue."`
- `" Paint it blue."`
- `"Paint it blue."`
- `"paint it blue."`
- `"Paint it blue.\n"`

Does adding one visible space necessarily add exactly one token? Does lowercasing change only an ID, or can it also change segmentation? Check rather than extrapolating from word counts. GPT-2's documented handling of leading spaces gives a reason to expect formatting to matter.

### 4.2 Code and identifiers

- `"total_count = 12\n"`
- `"totalCount = 12\n"`
- `"total_count=12\n"`
- `"def total(items):\n return sum(items)\n"`

Treat these as text fixtures; do not execute them. Compare spelling, punctuation, and indentation. A readable identifier may span several tokens, and source-code formatting is part of the input.

### 4.3 Numbers

- `"1234567890"`
- `"123 456 789 0"`
- `"12.50"`
- `"12,50"`
- `"0012"`

Predict whether the tokenizer uses single digits, larger chunks, or mixed pieces in each case. An observation about representation does not establish how well a neural model performs arithmetic.

### 4.4 Unicode

- `"café"`, using precomposed é at U+00E9.
- `"cafe\u0301"`, using e followed by a combining acute accent.
- `"東京"`.
- `"👤"`.

Record Unicode code points and UTF-8 bytes for these fixtures. Visually similar text can have different underlying representations. The Unicode Consortium’s normalization annex explains canonical equivalence; do not silently normalize your inputs during this experiment.

## 5 3. Collect the observations

For each case, save:

1. Input label and exact escaped text.
2. Unicode code-point count and UTF-8 byte count.
3. Token count, ordered token IDs, and tokenizer-reported token strings.
4. The result of decoding the complete ID sequence with special tokens retained.
5. An exact equality check between the original string and the decoded result.

The Encoding API documents the ID and token-string fields. Display strings with escapes so that spaces, tabs, and newlines cannot disappear in the presentation. The reported token string is an inspection representation; use the decoder to reconstruct text rather than concatenating those display strings.

When an equality check fails, preserve both strings and inspect their code points. Identify the first difference. Check settings and normalization before attributing the change to BPE itself. A mismatch is evidence to investigate, not a result to conceal by rewriting the input.

For one Unicode example, also decode each individual ID separately, then compare the concatenation of those individually decoded strings with a single decode of the complete sequence. Record what happens. Partial byte sequences may not form valid standalone characters, so per-token display can be misleading. Do not assume every token must be individually readable for full-sequence decoding to work.

## 6 4. Inspect special tokens separately

Find the declared special-token entries in the downloaded artifact. Locate GPT-2’s `<|endoftext|>` marker and record its ID from the file or tokenizer, rather than copying a guessed number.

Encode "Before<|endoftext|>After" with the baseline settings. Decode its IDs once retaining special tokens and once skipping them. Compare the two results. Then encode "Before After" with automatic special-token insertion both enabled and disabled. Did that flag actually add anything for this artifact’s configured post-processor?

Explain the distinction between recognizing an explicit marker, automatically inserting markers, and suppressing them during display. The mere existence of a flag does not guarantee that toggling it changes every tokenizer’s output. Also explain why this tokenizer-only inspection cannot demonstrate how a chat model would obey role boundaries.

## 7 5. Separate encoding length from model context

Inspect `model_max_length` in the companion `tokenizer_config.json`. It records 1024 for this artifact, but loading `tokenizer.json` directly does not automatically load that separate wrapper configuration. Pinned configuration file

With truncation disabled, construct a repeated harmless input whose measured encoding exceeds 1024 tokens. Save the count and round-trip result. Do not run a neural model on it. Explain why a tokenizer accepting the input cannot establish that GPT-2 supports that context length.

## 8 6. Interpret and submit

Produce three artifacts: a run manifest, a machine-readable results file, and a short written interpretation. Keep predictions made before execution alongside the measurements.

Your interpretation should answer:

- Which single-character or formatting change had the most surprising effect?
- Which examples show that tokens, words, code points, and bytes are different units?
- Did complete-sequence round trips preserve exact input? Explain any failures.
- What changed when special tokens were skipped?
- What evidence would be needed before claiming one language tokenizes “better” than another?
- Which conclusions concern this pinned tokenizer, and which concern neural-model behavior you did not measure?

Avoid ranking languages using unrelated short strings or treating token counts as intelligence scores. One carefully explained counterexample is more useful than a large table without interpretation.

### 8.1 Completion criteria

The Lab is complete when all four input groups have recorded predictions and observations; the manifest identifies reproducible artifacts and settings; exact round-trip checks are preserved; and the write-up correctly separates tokenization from model inference. There are no supplied token-count answers to imitate. Unexpected outputs belong in the record, with an explanation or an explicit open question.

## 9 More Learning

- Hugging Face Tokenizers pipeline — Trace normalization, pre-tokenization, segmentation, and post-processing as separate operations.
- GPT-2 technical report, Section 2.2 — Connect the byte-level design to what you observed.
- Unicode normalization annex — Investigate why visually equivalent strings need not have identical code-point sequences.

## 10 References

Artifact and API sources were checked on 7 October 2026. The fixtures and investigation design are original course material. This Lab supplies no fabricated execution results.

- Hugging Face. Pinned GPT-2 artifact listing, Tokenizer API, Encoding API, and tokenization pipeline.
- Radford, A., et al. (2019). Language Models are Unsupervised Multitask Learners.
- Unicode Consortium. Unicode Standard Annex #15: Unicode Normalization Forms.