

EDGAR EPS Parser (stdlib-only) — v4 Technical Report

This report provides a detailed technical overview of the final codebase for the EDGAR EPS Parser (version 4). It covers the script's purpose, how it processes filings, the heuristics used to identify EPS (Earnings Per Share) values, design decisions, and recommendations for further improvements. The code is invoked as:

```
python3.12 parser_v4.py ./Training_Filings/ ./final_output_v4.csv
```

The parser aims to extract the **latest-quarter GAAP EPS** from a variety of SEC EDGAR HTML filings using only Python's standard library (no external dependencies). The report is organized into sections that explain the end-to-end flow, parsing logic, scoring heuristics, and quality controls embedded in the code. Minor inconsistencies and potential improvements are also noted.

1) Overview & I/O Contract

Goal: The script's goal is to identify and extract the most recent quarter's GAAP Earnings Per Share (EPS) from SEC filings (10-Q, 10-K, press releases in HTML format, etc.). It must cope with heterogeneous HTML layouts, tables, and narratives, and distinguish GAAP EPS from non-GAAP or adjusted metrics. Emphasis is placed on capturing **quarterly EPS** (not annual) and **GAAP** (not "adjusted" or "pro forma") values, including treating "loss per share" as negative EPS where appropriate. The solution uses heuristic parsing (regex and simple HTML parsing) with the Python standard library only.

Input: A directory of `.html` / `.htm` files (e.g., EDGAR filings in HTML format). The code scans all such files in the given input directory.

Output: A CSV file with two columns: `filename` and `EPS`. Each input HTML filename corresponds to a row in the CSV. The `EPS` is formatted to two decimal places if an EPS value is found. If no valid EPS is found (or if an error occurs during parsing), the `EPS` field is left blank for that file. At the end of execution, the script prints the total time taken.

Execution Flow:

- The `main` function iterates over all HTML files in the input directory.
- For each file, it calls `extract_eps_from_html` to get the EPS value.
- Results are written to the CSV line by line inside a try/except block – ensuring that every file yields a CSV row (with blank if necessary) even if a parsing exception occurs.
- After processing all files, the total runtime in seconds is printed.

2) High-Level Architecture

The parser operates in two major phases: **table parsing** and **narrative text parsing**. It first attempts to find EPS in HTML tables (which is where financial statements usually appear). If that fails, it falls back to scanning the plain text for EPS mentions in narrative form.

Key Components:

1. HTML Table Parsing:

2. Uses a custom `TableParser` (built on Python's `HTMLParser`) to extract all tables from the HTML. Each table is represented as a list of rows, where each row is a list of cell texts.
3. Heuristics are applied to identify which table cells likely contain EPS values. This involves scanning for specific keywords in headers (like "earnings per share") and row labels ("Basic", "Diluted"), then examining numeric cells in those contexts.
4. A scoring system ranks candidate EPS values, preferring those in quarterly columns and labeled as Basic/Diluted EPS.

5. Narrative Text Parsing:

6. If no table yields an EPS, the script cleans the HTML to plain text (stripping tags, scripts, styles) and searches the text for phrases like "earnings per share" or "EPS".
7. Three regex-based strategies of varying strictness are applied in sequence to catch EPS mentions in sentences or paragraphs.
8. These also use heuristics to filter out false positives (like share counts, percentages, or non-GAAP figures).

9. Normalization & Validation:

10. All extracted number tokens go through normalization to a float (handling commas, dollar signs, parentheses for negatives).
11. The parser imposes a *hard cap* on plausible EPS values (ignoring anything with absolute value > 20, under the assumption that EPS outside ± 20 are likely not actual quarterly EPS but other figures).
12. Rows or contexts known to be irrelevant (like "shares outstanding", "revenue", "dividends per share") are ignored via a blocklist of terms.

13. Scoring and Selection:

14. Each potential EPS candidate is scored based on context: e.g., being in a "three months ended" column boosts score, being labeled "adjusted" penalizes it.
15. Among all candidates found, the one with the highest score is selected as the EPS for that file.

The result is a resilient heuristic parser that doesn't rely on a formal data format (like XBRL) but instead uses patterns and context to deduce the correct EPS value.

3) Core Building Blocks (Parsing & Normalization)

This section describes the foundational utilities: HTML parsing, text extraction, number finding, and normalization.

3.1 HTML Table Parsing with `HTMLParser`

- `TableParser` **class**: A custom HTML parser that traverses the HTML and extracts table data. It handles `<table>`, `<tr>`, `<td>` / `<th>` tags:
- On encountering a `<table>`, it begins collecting rows into a new Table object.
- For each `<tr>`, it starts a new row list.
- Each `<td>` or `<th>` triggers collection of text for a cell.
- It ignores script and style content entirely (using a `skip` flag for those tags).
- When closing a `</tr>`, it appends the row to the current table (if the row isn't empty).
- When closing `</table>`, it finalizes that table and adds it to `tables`.
- **Note**: This parser does not explicitly handle colspan/rowspan; it treats the HTML as if each `<td>` or `<th>` contributes one cell in the row. In most EDGAR filings (which use fairly simple tables), this is sufficient, but complex merged cells could misalign data.
- `Stripper` **class**: Another `HTMLParser` subclass used to convert HTML to plain text:
- It discards script/style content.
- It accumulates text from other tags, collapsing multiple spaces and preserving content.
- This is used for narrative parsing (to search the entire filing text for EPS mentions outside of tables).

3.2 Regex Patterns for Numbers and Context

A series of regular expressions are defined at the top of the code to identify numbers and key phrases: - `NUMBER_RE`: Matches numeric tokens potentially representing values: - Supports optional `$` prefix. - Allows commas as thousands separators. - Allows an optional sign or parentheses for negative values. - Example matches: `2.50`, `(1.25)`, `-$3,000`, `0.03`. - **Parentheses as negatives**: If a number is enclosed in parentheses (e.g., `(0.35)`), it is interpreted as a negative value, which is common in accounting for losses. - `ADJUSTED_RE`: Matches words like "adjusted", "non-gaap", "pro forma", or "core earnings" (case-insensitive). Indicates a non-GAAP context. - `BASIC_RE` / `DILUTED_RE`: Match the words "basic" or "diluted" (for EPS types). - `EPS_HEADER_RE`: Matches the phrases containing "earnings per share", "income per share", or "loss per share" – typically found in table headers or labels. - `LOSS_PHRASE_RE`: Matches phrases indicating a loss context, e.g., "loss per share" or "net loss". - `PER_SHARE_RE`: A broader pattern for narrative search, covering "earnings per share", "loss per share", or constructions like "per basic share". - `EPS_TOKEN_RE`: Matches the literal token "EPS", case-insensitive.

These regexes allow the code to find relevant sections in text and table cells.

3.3 Numeric Normalization and Scoring Helpers

- `normalize_num_token(tok: str, loss_ctx: bool) -> Optional[float]`: Converts a string token to a float:

- Strips parentheses, currency symbols, and commas.
 - Converts to float, applying negative sign if it was in parentheses.
 - If `loss_ctx` is True (the surrounding context indicates a loss), any positive number is flipped to negative (because sometimes a loss is stated as a positive number but contextually it means a negative EPS).
 - Returns `None` if conversion fails.
- **Magnitude filters:** The code imposes a **HARD_ABS_CAP = 20.0**. Any candidate number with absolute value greater than 20.0 is immediately discarded as implausible for EPS. This is a safety net to avoid picking up things like share counts or dollar amounts that are not per-share earnings.
 - **Scoring helper functions:** To differentiate between multiple candidate values, a scoring system is used:
 - `plausibility_penalty(v: float) -> float`: Soft penalty for large magnitudes:
 - 0 penalty if $|v| \leq 20$ (within expected range).
 - Gradual penalty as $|v|$ increases beyond 20, but given the hard cap, this mostly affects borderline values.
 - `decimal_bonus(tok: str) -> float`: +0.5 bonus if the number contains a decimal point. Rationale: EPS is usually reported in dollars and cents (e.g., 0.35), so a pure integer like "2" might be less likely an EPS unless context strongly says so.
 - `integer_penalty(tok: str, val: float) -> float`: If the token has no decimal and the absolute value is somewhat large:
 - If it's an integer and $|val| \geq 10$, a penalty of -2.0 is applied (because an EPS of 10 or more, without cents, is unusual in typical quarterly results).
 - If it's an integer but smaller (<10), a smaller penalty (-0.5) is applied, just to prefer a precise decimal if available.

These bonuses and penalties fine-tune candidate scoring to prefer values that "look like" typical EPS (e.g., something like 0.XX or 1.23 is more plausible than a round 10 or 15).

3.4 Blocklist for Irrelevant Rows

A crucial heuristic is the `BLOCK_ROW_RE` regex, which is used to filter out table rows that are very unlikely to contain the EPS. Before considering a row as an EPS candidate, the code checks if the row's text matches this regex, which includes: - Terms about shares (e.g., "weighted average shares", "shares outstanding", "common shares"). - Other financial metrics (revenue, sales, EBITDA, margins, percentages). - Dividends (e.g., "dividends per share", "cash dividends"). - Book value or NAV (net asset value).

If a row's text matches any of these cues, the row is *ignored* in the EPS search. This prevents common mistakes like picking up "book value per share" or "dividends per share" or using a share count as EPS.

4) Table Extraction Logic

The function `extract_from_table(rows: List[List[str]])` implements the logic to find an EPS value from a parsed table (represented as a list of rows of cell texts). It works in two passes within each table:

4.1 Identifying EPS Rows and “Bands”

EPS Header Rows: First, it scans the table for any row that contains the key phrase “earnings per share” (or “income per share” or “loss per share”), using the `EPS_HEADER_RE` regex. If found, such a row likely indicates the start of an EPS section in a financial statement (e.g., a header or label for EPS).

When an EPS header row is found at index `i`, the code considers a small block of subsequent rows as part of the EPS section – specifically `rows[i]` through `rows[i+3]` (up to 3 rows following the header). This is because often financial statements will have a structure like:

```
Earnings per share:  
Basic    $X.XX  
Diluted  $Y.YY
```

Or sometimes the header might be a combined line and the next one or two lines are Basic and Diluted.

This block of 1–4 rows is treated as a group where EPS might appear. Within this group: - Each row is skipped if it matches the blacklist (e.g., if an “Earnings per share” header is present but some following row is clearly not EPS data, it will be ignored). - For each row in the band, a label string is created (the concatenation of all cell texts in the row, lowercased) to examine context (does it mention basic, diluted, adjusted, loss, etc.). - A boolean `is_loss` is set true if the label or the original EPS header indicated a loss context (meaning the values should be treated as negative if positive).

4.2 Column Selection – Focusing on the Latest Quarter

If the table has identifiable header rows (determined by `row_is_header_like` which checks if a row has at least 60% non-numeric cells), the parser tries to determine which column is likely the latest quarter’s EPS column: - The function `pick_best_quarter_column(header_rows)` examines up to the first 4 header-like rows and scores each column index: - **Quarter cues:** If the header text in that column mentions “three months ended”, “quarter ended”, “Q1/Q2/Q3/Q4”, etc., that column gets a large positive score (e.g., +12). - **Annual cues:** If the header mentions “year ended”, “twelve months ended”, etc., that column is penalized heavily (e.g., -8) because we prefer quarterly data. - **Year recency:** It extracts any year (4-digit number) in the header text for that column. Newer years contribute a positive score. For example, seeing “2023” is scored higher than “2022”. - **Column index bias:** A slight penalty is added proportional to the column index ($-0.15 * \text{index}$), biasing the leftmost columns. This is based on the observation that some filings list the most recent quarter first (on the left). (Note: This assumption may not always hold, but the year check somewhat counters it by rewarding the latest year even if it appeared on the right.)

The column with the highest score is chosen as `best_col`. If no particular column stands out (or no headers present), `best_col` may remain None.

Use of `best_col`: When scanning for numeric values in the table, if `best_col` is identified, the code will by default only consider numbers in that column (and possibly `best_col + 1`). This is to ignore other columns (like prior quarters or yearly data) that might also have EPS numbers, focusing on the latest quarter. If `best_col` is None (no clear column determined), the code will cautiously skip numbers in any column whose header looked like an annual column, but otherwise consider numbers from any column.

4.3 Scoring Table Candidates

Within each candidate EPS row or “band,” each numeric cell is considered as a potential EPS, and a score is calculated for it: - It first merges tokens that are split across cells in rare cases (e.g., a "(" in one cell and ")" in the next). - It then normalizes the token to a float using `normalize_num_token`. If normalization fails or the value exceeds the hard cap (20), that candidate is skipped. - Several factors contribute to the score: - **Base score from context label:** - If the row's label contains "adjusted" or "non-GAAP" keywords, **-5.0** penalty (we want GAAP EPS). - If the label contains "Basic", add a base score (different in two scenarios, explained below). - If the label contains "Diluted", add a base score. - In the EPS header band scenario: - The code currently adds **+4.0** if "basic" is mentioned, and **+1.0** if "diluted" is mentioned. - (This suggests a slight preference for Basic EPS in that context.) - If the row label doesn't explicitly mention basic, diluted, or EPS, a small penalty (-0.5) is applied to deem it less likely an EPS row. - **Header context score:** If header_rows exist, it adds `header_context_score(column_index)`: - This is similar to the quarter/annual cues above, but focused on the specific column for that cell. - For example, a number in a column whose header says "Quarter Ended June 30, 2025" will get a strong positive boost, whereas a number under "Year Ended 2024" gets a penalty. - **Numeric form adjustments:** - `decimal_bonus` (+0.5 for decimals) and `integer_penalty` (up to -2 for large integers) as discussed, to favor plausible EPS formats. - **Plausibility penalty:** If the value is large (close to the cap), a slight penalty is applied (though anything above 20 is already excluded). - All these components sum up to a total score for that candidate value.

The code collects all (score, value) pairs for candidates and will later choose the highest score.

4.4 Fallback: Basic/Diluted Rows without Explicit Header

If no EPS was found via the above approach (no explicit "earnings per share" header row in the table or nothing selected from those bands), the parser takes a broader look: - It scans every row of the table for the presence of the words "Basic" or "Diluted" (which often appear even if an "earnings per share" header isn't explicitly labeled). - For any such row (again skipping blocklisted rows), it performs a similar evaluation: - The base scoring here is slightly different: - For these general Basic/Diluted rows, the code adds **+2.0** if "basic" is in the label, and **+3.0** if "diluted" is in the label. (Interestingly, in this fallback path, Diluted is given a higher base score than Basic, which is an inconsistency compared to the earlier band scoring.) - Adjusted still incurs a -5 penalty. - It then considers numeric cells in the preferred column(s) similarly, normalizes, checks caps, and scores with the same header context and numeric form adjustments.

Any candidates from this fallback path are added to the list of candidates (if the primary path had none, then these become the only candidates).

4.5 Selecting the Best Table Candidate

After scanning the table via both methods, if any candidates were found, they are sorted by score. The highest-scoring candidate's value is selected as the EPS from that table. If multiple tables are in the HTML, `extract_eps_from_html` (which calls this) will compare candidates from all tables and pick the best among all tables.

If no candidate is found in any table, the function returns `(None, -inf)` for that table, which leads the outer logic to proceed to narrative text parsing as a fallback.

5) Narrative Text Extraction — Three-Pass Strategy

When table parsing does not yield a result, or if the filing might have the EPS mentioned only in prose (e.g., in a press release summary or MD&A section), the parser switches to text mode. It uses the plain text (extracted by the `Stripper`) and runs several regex-based extraction strategies in order, from most strict/targeted to more general:

5.1 Strict Phrase Search (`extract_from_narrative_strict`)

This pass looks for the exact phrases **“earnings per share”**, **“income per share”**, or **“loss per share”** in the text (case-insensitive). When such a phrase is found: - It takes a window of 120 characters *after* that phrase and searches for a number. - It specifically looks for a decimal number (requires a decimal point in the token for higher confidence). - If found, it normalizes the number and applies context: - If the phrase itself or the immediate text after it contains "loss", it sets `is_loss` true to flip sign if needed. - It applies a small scoring: e.g., +0.75 if "diluted" is nearby in the window, +0.25 if "basic" is nearby. This is because sometimes the sentence might be "Diluted earnings per share were X". - It ignores any number that's above the magnitude cap or not a proper float. - Among all matches of the strict phrase, it will choose the one with the best score (usually the first valid one, since not many instances of such a precise phrase should appear).

This strict search aims to catch straightforward statements like: "Earnings per share (diluted) was \$0.45 for the quarter..."

5.2 “EPS” Token Search (`extract_from_text_eps_token`)

If the strict phrase didn't yield anything, the next approach looks for the token **“EPS”** itself in the text, which is a common shorthand in press releases. For each occurrence of "EPS": - It takes a window (roughly 140 characters before and 160 after the token for context). - It then tries to find a number, preferring one immediately following "EPS". If none immediately follows, it will take the first number anywhere in that window. - It requires that the number contain a decimal point (to filter out things like "EPS of 2 cents" which might be written as "\$0.02" — but if it were "2", it could be ambiguous). - Scoring context within this window: - If words indicating a quarterly context appear (e.g., "three months", "quarter", "Q1/Q2..."), **+2** score. - If words indicating annual or multi-month context appear ("year", "twelve months", "six months"), **-2** penalty. - If the word "adjusted" appears near "EPS", **-5** penalty (likely a non-GAAP adjusted EPS). - If "basic" or "diluted" appear, small bonuses (+0.5 for basic, +0.25 for diluted) in this window. - Loss context (if "loss" appears in the window) is noted to possibly flip sign. - As always, the candidate number is normalized and

filtered by magnitude. - This method can catch sentences like: "Q4 2024 GAAP EPS was \$1.05 (diluted) and adjusted EPS was \$1.10."

All candidates from all "EPS" occurrences are scored and the top is chosen if any.

5.3 General Per-Share Pattern (`extract_from_text`)

The final fallback is a broader search for any mention of "**per share**" in the text (using the `PER_SHARE_RE` which covers various forms like "earnings per share", "loss per share", or "per basic share", etc.). For each such occurrence: - It takes a window around the phrase (120 chars before and 160 after). - It tries to locate a number associated with that phrase: - Priority is given to a number immediately *after* the phrase (the idea being typically it might read "... loss per share was \$0.50"). - If none found after, it will look immediately *before* the phrase (e.g., in case the wording is reversed, though less common). - If that fails, it will take any number within the window. - It then builds a context string of the text around that phrase (both before and after) to check for cues: - Determines `is_loss` if "loss" is in that broader context. - Checks for "adjusted", "basic", "diluted" similarly to apply base score adjustments. - The candidate number is normalized (must have a decimal) and checked against the cap. - Scoring for this method uses similar rules: - Adjusted → -5, Basic → +2, Diluted → +1 (these weights are similar to the table context weights). - A decimal and magnitude bonuses/penalties are applied as well.

Multiple candidates might be found in text; the one with the highest score is chosen.

Note: The narrative search is careful about requiring a decimal point in the number for most cases. This is because a plain integer in text could easily be something like a year or some index. By requiring a decimal, we align with the assumption that EPS is usually reported with cents. This does risk missing an EPS that is exactly a whole number (e.g., \$1.00 might be written as \$1), but such cases are rarer and the strict phrase search might catch them if phrased clearly.

5.4 Combining Table and Text Results

The function `extract_eps_from_html` coordinates these strategies: - It first uses `TableParser` to extract tables and tries `extract_from_table` on each table. If any table returns an EPS value, `extract_eps_from_html` will return the best one found across all tables. - If all tables return `None` (no candidate), it then: 1. Tries `extract_from_narrative_strict`. If that returns a value, it returns it immediately. 2. If not, tries `extract_from_text_eps_token`. If that returns a value, returns it. 3. If not, finally tries `extract_from_text` (general search) and returns whatever that yields (which could still be `None` if nothing found). - Thus, the preference order is: Table-based EPS (most structured) over any narrative results; and within narrative, exact phrases over generic patterns.

6) Scoring Philosophy & Preference Rules

Throughout the code, various scoring tweaks reflect a philosophy of what a "good" EPS candidate looks like. Here's a summary of the key preferences encoded:

- **Prefer GAAP over non-GAAP:** Any hint of "adjusted", "non-GAAP", "pro forma", or similar terms in proximity to a number causes a significant penalty (-5). This means if both GAAP and adjusted EPS are mentioned, the GAAP one (usually not labeled "adjusted") should score higher.
- **Prefer Quarterly over Annual:** The logic actively looks for quarterly indicators (three months, quarter ended dates) and penalizes annual contexts. This is done both in table column selection and in narrative scoring. The assumption is that the user wants the latest quarter's EPS, not an annual EPS. A column or sentence referencing a year or 12-month period is less likely to contain the target value.
- **Recency Bias:** Within multiple quarterly columns, the code biases towards the most recent quarter. If multiple years are present in headers, the highest year gets more points. There's also a slight left-column bias assuming many tables list the latest period first. This bias is small and mainly serves as a tiebreaker if year cues are absent.
- **Basic vs. Diluted EPS:** Companies usually report both Basic and Diluted EPS. The code's intended behavior is a bit nuanced:
 - In the table scanning's "EPS band" section, the scoring gives Basic a higher boost than Diluted (+4 vs +1), implying a slight preference for Basic EPS (maybe assuming Basic is listed first or is the one to pick if both given).
 - In the fallback table scan and in narrative text, the weights are different (+2 for Basic vs +3 for Diluted, or +0.5 vs +0.25 in some narrative contexts), which actually prefers Diluted in those contexts.
 - This inconsistency is noted and likely unintentional. A consistent approach should be decided (e.g., always pick Diluted EPS as the final answer if both are present, or always Basic, depending on user needs). Currently, depending on which logic finds the value first, the output could be Basic or Diluted. This is flagged for review (see section 7).
- **Loss as Negative:** If the context suggests it's a loss (e.g., "net loss per share of 0.35"), the code will invert the sign of the EPS to -0.35 after parsing. This ensures the output reflects a loss correctly as a negative number even if the text or table showed it without a minus sign (common in accounting to not use negative sign with "loss").
- **Avoid False Positives:** The combination of the blocklist for rows, ignoring values without a decimal in narrative, and the plausibility limits all serve to reduce picking up the wrong number:
 - For example, a table might have a row "Weighted average shares: 1,234,567" right below EPS. The blocklist would exclude this row entirely despite containing a number.
 - Or a sentence might say "Earnings per share increased 2% to \$0.75 from \$0.50" – the presence of "2%" could confuse a naive search, but the parser focuses on numeric tokens that fit the dollar pattern and context, not percentages.

- By capping at 20, values like “\$45.00 per share” (which might be something like a stock price or buyout offer in the text, not EPS) would be ignored.

In summary, the scoring is tuned to find a plausible EPS figure (usually a small dollar amount with cents, in a quarterly context, labeled as EPS) and ignore other numerical data.

7) Notable Inconsistencies and Potential Fixes

While the code is functional, a couple of subtle inconsistencies in scoring have been identified, particularly around how Basic vs. Diluted EPS are handled. These do not necessarily break the parser, but they could lead to unpredictable selections if both Basic and Diluted are present:

1. **Comment vs. Code (Basic vs Diluted in Table Band):** In the EPS band extraction code, a comment suggests “Prefer Diluted slightly over Basic (sample set expectation)”, but the actual scoring does the opposite (Basic +4 vs Diluted +1). This likely means either the comment is outdated or the weights are inverted accidentally. It should be clarified and aligned with the desired behavior.
2. **Inconsistent Basic/Diluted Preference Across Paths:** The table EPS-band path prefers Basic, whereas the fallback path prefers Diluted (and some narrative logic slightly prefers Basic again). This inconsistency means the extracted value could depend on which path found it. For example, if a table clearly labels both Basic and Diluted, the table logic might choose Basic (due to higher score). But if only narrative parsing catches it, and both “basic EPS” and “diluted EPS” appear in text, the weighting might choose diluted. It is recommended to standardize this:
3. If the requirement is to always report Diluted EPS (which is common in financial reporting as the primary EPS), then weights should consistently favor Diluted in all contexts.
4. Or vice versa, if Basic is desired. Either way, making it uniform avoids confusion.

Neither of these issues stops the parser from finding an EPS, but addressing them would make the output more predictable and aligned with user expectations. The code’s author should adjust the scoring values to be coherent and update/remove misleading comments.

8) Complexity & Performance

This parser is designed to be efficient on typical filings:

- **Time Complexity:** The table parsing is linear with respect to the size of the HTML. It goes through each tag and builds tables. Scoring is then roughly $O(N)$ for table cells and $O(M)$ for text length with regex, where N is number of cells and M is length of text. In practical terms, financial filings are at most a few MB of text, and the operations (regex and string processing) are quite fast in Python. The code avoids heavy nested loops except iterating within small windows or small tables, so performance should be on the order of a fraction of a second per filing. For dozens or hundreds of filings, this is easily manageable.

- **Memory:** The data structures are lightweight (lists of lists for tables, a single concatenated text string for narrative). The parser does not construct a full DOM, which saves memory. Instead, it processes streaming data with HTMLParser callbacks. Memory use should remain low (tens of MB at most, even for large filings).
- **Robustness:** By wrapping each file's processing in a try/except, the script ensures one bad file (or an unexpected HTML format causing an exception) won't crash the entire batch. It will log an empty EPS for that file and continue.
- **Runtime Output:** The script prints the total time after processing all files, which can be useful for profiling on large batches.

Performance note: One potential heavy operation is the regex searches on large text. However, each narrative regex (strict phrase, EPS token, general per-share) will scan through the text. Python's regex engine is quite optimized in C, and the patterns are not extremely complex, so this is typically fine. If needed, performance could be improved by combining some searches or adding early breaks once an EPS is found, but given the small number of targets, it's acceptable.

9) Strengths of Version 4

Compared to earlier iterations, v4 of the parser introduces several improvements that increase accuracy:

- **Lower False Positives:** The introduction of a comprehensive blocklist for irrelevant rows (e.g., NAV, dividends, share counts) and the strict hard cap on values greatly reduce the chance of picking up a wrong number. For instance, previous versions might have mistakenly output a dividend per share if it appeared in text near "per share", but v4 will ignore it due to keywords like "dividend".
- **Quarter Targeting:** The new header-based column selection logic means the parser is much better at zeroing in on the correct column in a financial statement table. If a table has multiple columns of data (as most do), v4 will usually select the column corresponding to the latest quarter. This prevents errors like taking an annual EPS or an earlier quarter's EPS.
- **Robust to Different Formats:** By combining table parsing with narrative parsing, the parser covers a wide range of filing styles:
 - Some companies present EPS only in a table – we catch that.
 - Some mention EPS in the MD&A text or in a press release summary – we catch that via regex.
 - Some use "EPS" shorthand, others spell it out – both are handled.
 - Losses and profits are handled uniformly (sign-adjusted).
- "Basic" and "Diluted" lines can be captured even if the term "earnings per share" isn't explicitly repeated on those lines (thanks to scanning the band and fallback on Basic/Diluted rows).
- **No External Dependencies:** The solution meets the requirement of using only Python's standard library, which makes it easy to run in restricted environments and avoids complicated package

installations. Using `HTMLParser` and regex might be less fancy than an HTML DOM parser or an XBRL library, but it's effective for the task and keeps the tool simple and portable.

- **Output Formatting and Safety:** The output is neatly formatted to two decimals, which is appropriate for EPS. The code safely handles exceptions per file, meaning one problematic file won't halt the entire process.

10) Limitations & Edge Cases

No solution is perfect, and there are scenarios where this parser might struggle:

- **Complex Table Structures:** If an HTML table uses extensive `rowspan` or `colspan`, the simplistic way we collect cells might misalign data. For example, if a table has a multi-level header that spans columns, our `TableParser` might treat one logical column as multiple or vice versa. This could lead to picking the wrong cell as `best_col` or misidentifying which number corresponds to EPS. Most filings keep things simple, but this is a known limitation.
- **Unusual EPS Formats:** While we assume EPS is a fairly small decimal number, there could be outliers:
 - A company with a very high stock price might have an EPS above \$20 in rare cases (e.g., Berkshire Hathaway has EPS in the hundreds or thousands of dollars because their stock is extremely high-priced). Our parser would currently miss such values due to the hard cap.
 - Conversely, companies sometimes report EPS in different units (e.g., in cents, like "\$0.05" or even "5 cents"). We handle the \$ sign, but if something was reported like "5 cents per share", our numeric regex might catch "5" but then ignore it because it lacks a decimal. Thus, the parser could miss cases where EPS is given as a whole number of cents without a decimal (though the presence of the word "cents" might exclude it anyway since we look for \$ or decimal format).
- **Context Ambiguity:** The parser relies on context keywords. If a filing is oddly worded such that none of our cues (earnings, per share, EPS, basic, diluted) appear near the number, we might not capture it. For example, a very terse summary like "Q1 2025: \$0.45 vs \$0.40 prior year." Without "EPS" or "per share", our regex might not know \$0.45 is EPS (it could be a revenue per share or something else in theory). Such cases seem rare, but they highlight reliance on expected phrasing.
- **Date Order Assumption:** The logic assumes the latest quarter can be detected by year or position. If a table is laid out with latest quarter on the rightmost column (some companies do this, listing older periods first and most recent last), our left-bias might be counterproductive. We do look at year, which helps if the year changed, but within the same year, quarter sequence (Q1 vs Q2) wouldn't be known. A more robust date parsing of column headers (looking at actual dates and comparing them) is not implemented. Thus, there's a small risk the parser picks the wrong quarter's EPS if the table ordering is unexpected and year alone doesn't tell the whole story.

- **Multiple EPS Metrics:** Some filings might list GAAP EPS and also additional EPS metrics (like continuing operations EPS, or an unusual item adjusted EPS) in close proximity. Our parser might pick up the wrong one if it isn't clearly labeled "adjusted". We do penalize "adjusted", but if both are GAAP measures (just different breakdowns), we might not differentiate.
- **Non-English Text:** The parser is tailored to English phrases like "earnings per share". If filings were in another language or format (unlikely for EDGAR, since it's US-based, but possibly for some exhibits), the regex would fail.

In summary, while the parser covers most common cases, extremely complex filings, unconventional data presentation, or extraordinary values could challenge it. However, these should be outliers and could be addressed with further enhancements.

11) Quick QA Checklist for Verification

When testing the parser on a sample of filings, here are some quick things to verify to ensure it's working as intended:

- **Correct Column Picked:** In a table that has both quarterly and annual columns (e.g., "Three months ended..." vs "Year ended..."), check that the output corresponds to the quarterly figure, not the annual figure.
- **Basic vs Diluted Handling:** If a filing shows both Basic and Diluted EPS, ensure the value output is the one you expect (by design, decide if it should be Basic or Diluted). Currently, see if it consistently picks one or if it might vary. This helps confirm if the Basic/Diluted scoring issue is affecting results.
- **Negative EPS (Loss):** Find an example where the company had a loss (negative EPS). Verify that the output is a negative number (e.g., -0.35) and not a positive. The parser should correctly flip the sign either because of parentheses or the word "loss".
- **Non-GAAP Exclusion:** If a filing presents something like "Adjusted EPS" alongside "EPS", check that the parser outputs the plain EPS, not the adjusted. The adjusted one should have been penalized or ignored (especially if clearly labeled "Adjusted").
- **No False Triggers:** Check a row like "Cash dividends per share: \$1.00" or "Book value per share: \$25.00" if present in a filing. The parser should not pick those as EPS. They likely would be excluded by the blocklist (dividends, book value) or by context scoring (no "earnings" keyword, likely filtered out). Confirm that such files result in either the correct EPS or blank if EPS isn't found, rather than a wrong value.
- **Performance on Bulk:** Run on a batch of, say, 50 filings and ensure it completes in a reasonable time (a few seconds) and every line in the CSV is filled appropriately. No crashes and the time printed at the end seems plausible.

Performing the above checks can increase confidence that the parser is reliable and the heuristics are functioning as expected.

12) Suggested Next Improvements

Moving forward, if there's time to refine this tool further (still keeping to stdlib-only or minimal changes), here are some ideas:

1. **Standardize Basic vs Diluted Preference:** Decide which EPS to report when both are available. A common choice is **Diluted EPS** (since it's more conservative and often highlighted). If so, adjust the scoring: give "Diluted" a consistently higher base score than "Basic" across all logic branches. This will ensure the parser picks Diluted whenever both are present. Update any comments to match this logic.
2. **Precise Date Parsing for Columns:** Enhance the column selection by actually parsing dates in header cells. For example, if headers are "Three months ended March 31, 2025" and "Three months ended June 30, 2025", you could parse those dates and know June 30, 2025 is more recent than March 31, 2025, regardless of column order. This would remove the need for the left/right bias and directly target the latest date. This can be done with Python's `datetime` if month names are present. This is more involved but would make the "latest quarter" detection bulletproof.
3. **Expand Narrative Patterns (cents, unusual phrasing):** Consider capturing patterns like "X cents per share" or cases where the number might be written as an integer (if followed by "cents"). For example, "earnings of 5 cents per share" currently might be missed because the code looks for a decimal. A regex including the word "cents" and capturing the number could cover this edge case.
4. **Rowspan/Colspan Handling:** The HTML table parser could be improved to handle basic colspan/rowspan by adjusting indexing of cells. A more advanced approach could reconstruct table structure more faithfully. However, this can get complicated quickly. Testing on a few problematic filings to see how often this matters would inform if this is worth doing.
5. **Logging/Debug Mode:** Implement a verbose or debug flag that prints out which candidates were found and why one was chosen over another. This would help users of the script to trace decisions, especially on tricky filings. For example, log: "Found Basic EPS = 1.23 (score X), Diluted EPS = 1.20 (score Y). Chose Basic." or "Skipped value 25.00 as EPS because it exceeded cap." This could be printed to console or a log file for analysis.
6. **Multilingual/Generalization:** If needed beyond EDGAR, consider parameterizing the keywords so that other languages or report types could be handled by swapping out the regex patterns.

Each of these improvements would make the tool more robust or easier to maintain, though some (like date parsing) add complexity. Given that v4 already meets the immediate needs, these are just ideas for future development.

13) Summary

The EDGAR EPS Parser v4 is a comprehensive, heuristic-based solution for extracting the latest quarterly Earnings Per Share from SEC filings in HTML format using only Python's standard libraries. It parses HTML tables to find relevant rows and columns, uses context keywords and scoring to isolate the GAAP EPS figure, and falls back to searching narrative text when tables don't yield an answer. Key strengths include its avoidance of common pitfalls (like confusing EPS with other per-share metrics) and its emphasis on the most recent, GAAP values. The design balances accuracy with simplicity, achieving good results without needing complex XML parsing or external financial data libraries.

In usage, the tool should quickly process a batch of filings and output a CSV of filenames with their corresponding EPS. The internal logic, while reliant on carefully tuned heuristics, has proven effective on diverse filing formats. By addressing the noted inconsistencies (especially around Basic vs Diluted preference) and potentially enhancing date recognition, the parser can be further refined. Nonetheless, in its current state, it provides a reliable automated solution to pull out EPS data for financial analysis or reporting needs, demonstrating how far one can go with smart text parsing techniques in pure Python.
