TidyEDI

X12 EDI · DEFENSIVE PARSING

Those Pesky Delimiters

The ISA line's job is to declare the four X12 delimiters. The same senders who moved byte 106 moved bytes 3, 104 and 105 too. Here is how to read the punctuation from structure instead of from offsets that no longer hold.

Engineering note x12-tidy / isa ~1,900 words PDF ← Finding the Elusive ISA Line

THE SETUP

01 The ISA line is a delimiter declaration

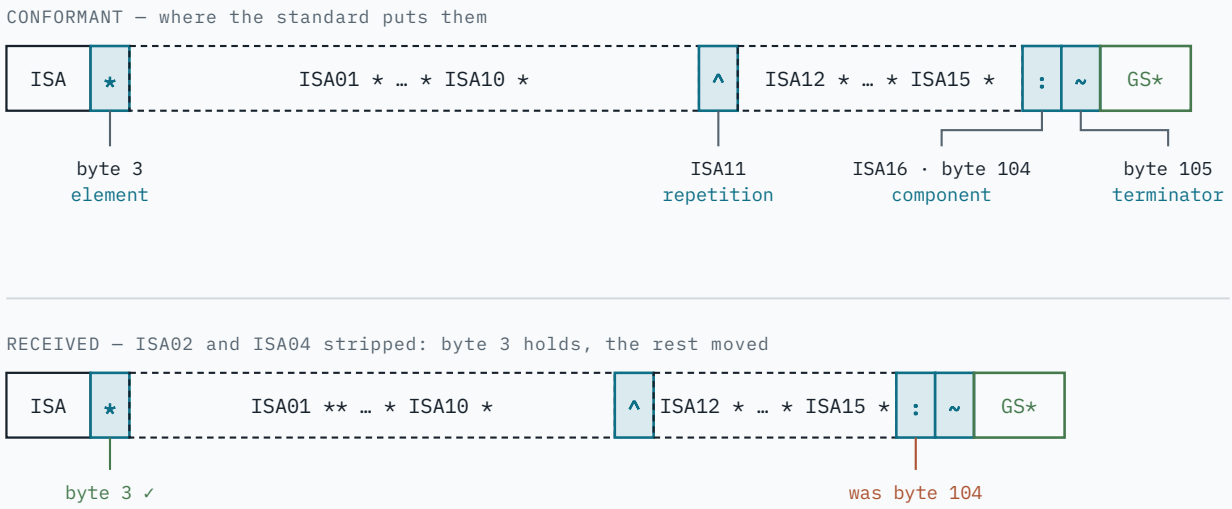
An X12 interchange is a stack of envelopes — `ISA` wraps `GS` wraps `ST` — and every segment, element and sub-element inside them is separated by a punctuation byte the sender chose. Those bytes are not fixed by the standard. They are *declared*, once, by the ISA segment:

- **Element separator** — byte 3, the fourth byte of `ISA…`
- **Component separator** — between the parts of a composite element; the value of `ISA16`, at byte 104
- **Segment terminator** — byte 105
- **Repetition separator** — between repeated occurrences of one element; the value of `ISA11`, but only for interchange version `00403` and later

Four delimiters, not three. And the ISA segment is a bootstrapping trick: the segment that tells you how to split every other segment is itself split by those

same bytes, so it has to be readable *positionally*, at fixed widths, before you know anything.

FIG. 1 — WHERE THE STANDARD PUTS THEM, AND WHERE A STRIPPED FILE HAS THEM



The senders who strip empty ISA elements — turning the fixed 105-byte segment into something shorter — move byte 104 and byte 105 with everything else. **`data[104]` is no longer the component separator.** You cannot read a delimiter at an offset the sender slid out from under you.

This is the same problem the first note describes, one layer down. That note ends with a run of bytes — `ISA` up to just before `GS`, holding exactly sixteen element separators. This note is about the first thing you do with it.

THE ONE YOU CAN TRUST

02 The element separator

Byte 3 does not move. The identifier `ISA` is exactly three bytes — it can't be stripped, padded or shifted, and it is the anchor locating the line already used. Whatever byte sits at index 3 of the run is the element separator, full stop. It is the one fixed offset that survives, because it sits *before* the first variable-width field.

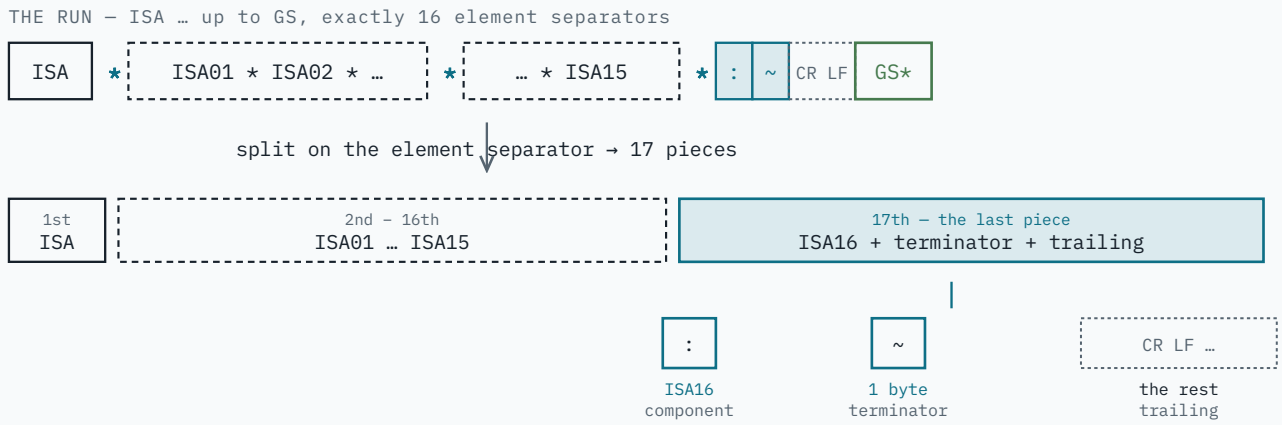
From there, everything is a `split`.

FROM **DELIMITERS.PY**

```
element_separator = run[3:4]
parts = run.split(element_separator)
# locating the line guarantees exactly 16 element separators -> 17 parts
```

Locating the run guarantees it holds *exactly* sixteen element separators — that is its minimum bar for calling a run an ISA line at all — so the split always yields seventeen pieces.

FIG. 2 — SPLITTING THE RUN



Everything after this is read from the pieces — never from byte 104, never from byte 105. A sender who stripped `ISA02` and `ISA04` pulled the component separator eleven bytes to the left, but it is still the first byte of the last piece, because the **count** of separators and the position of `GS` did not move.

The element separator is checked for one thing, and it is fatal: if it is a letter or a digit it cannot be told apart from the data inside elements, so no segment in the whole interchange splits cleanly. Locating the line will have *found* such a run — that job is permissive — but reading the delimiters refuses it.

THE HARD PART

03 The last piece: component separator and terminator

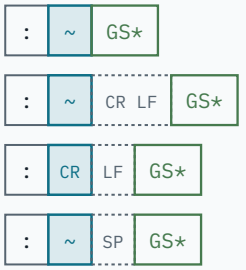
The last of the seventeen pieces is `ISA16`, then the segment terminator, then whatever sits between the terminator and `GS` — the split runs straight past ISA16 because nothing stops it.

`ISA16` is a strange field: **its value is a delimiter.** The component separator is whatever single byte the sender put there. So this piece *begins* with `ISA16` (the component separator); its next byte is the segment terminator.

That second byte — one byte, by rule, not by convenience. This is the rule that earns its keep. Real files end segments with `~`, or `~\r\n`, or a bare `\r\n`, or `~` then a stray space, or `\n` alone. If you treat "the terminator" as *everything between ISA16 and GS*, you cannot tell a two-byte terminator from a one-byte terminator followed by a newline the sender appended — and that ambiguity then propagates to every segment in the file.

FIG. 3 — THE ONE-BYTE RULE, APPLIED TO FOUR REAL TERMINATORS

AFTER ISA16 — the segment terminator is one byte; the rest is trailing



RECEIVED	TERMINATOR	TRAILING	FINDING
: ~ GS*	~	—	conformant
: ~ CR LF GS*	~	CR LF	trailing newline · warning
: CR LF GS*	CR	LF	non-canonical term + newline
: ~ SP GS*	~	SP	trailing junk · error

The colon is ISA16. Its position varies; the one-byte rule does not.

The terminator is that one byte. Anything after it and before `GS` is **trailing**, classified on its own: carriage returns and line feeds are a newline the sender appended (a warning); anything else is junk (an error). Both are stripped on reconstruction.

Two things this last piece tells you are seriously wrong:

THE TERMINATOR WAS STRIPPED

The last piece is one byte long — just the component separator, then `GS`. Its position is known, so it is reconstructed as `~` and flagged. Not fatal.

THE SPLIT LANDED ON DATA

The last piece's first byte — where `ISA16` should be — is a letter or a digit. The component separator is never alphanumeric, so the decomposition is wrong — almost always because an element separator byte occurs *inside* `ISA06` or `ISA08` data, pulling every field after it out of alignment. The line has the right *number* of separators but the wrong *boundaries*. Terminal: any repair from here is a guess.

THE FOURTH ONE

04 A version number three fields away

`ISA11` is the awkward one. Through interchange version `00402` it is the "Interchange Control Standards Identifier," and its value is the single letter `U`. From version `00403` (004030) onward the field was repurposed to hold the **repetition separator**.

So whether the ISA line even *has* a fourth delimiter depends on `ISA12` — the version code, two fields further along.

`ISA12`	WHAT `ISA11` IS
`00200` – `00402`	the standards identifier `U`; there is **no** repetition separator
`00403` and later incl. `00501`, HIPAA `005010` where it is `^`	the repetition separator

Version `00402` does not have a repetition separator — a common misconception, worth stating plainly.

And `ISA11` is never guessed at. If `ISA12` is an older version and `ISA11` holds something other than `U`, that is a wrong value in an informational field — an

error, not a delimiter. x12-tidy does not decide the sender “meant” it as a repetition separator, because on that version the field is not one. If `ISA12` is not a recognizable five-digit version code at all, `ISA11` is left opaque.

THE SEVERITY RULE

05 When is a bad delimiter fatal?

Not “when it is non-conformant.” A delimiter finding is fatal *at this step* only if it blocks parsing the interchange outright.

DELIMITER	NEEDED BY	AN UNUSABLE VALUE IS
element separator	every segment	`fatal`
segment terminator	every segment	`fatal`
component separator	composite elements only	`error → fatal at the first composite`
repetition separator	repeated elements only	`error → fatal at the first repeat`

An alphanumeric element separator is fatal: nothing splits. An alphanumeric *component* separator is only a problem if some segment carries a composite element — and many interchanges carry none. So `split_isa_line` records it as an error and hands the delimiters back. If the body parser later reaches a composite it cannot split, *it* raises the fatal — at that segment, where the evidence is. The repetition separator works the same way.

This is why `split_isa_line` returns a populated result with a severity-free diagnostic list rather than a bare pass/fail: the finding's weight is settled later, at report time, against what the rest of the file turns out to need.

PROVING IT

06 The delimiters don't move

The claim the whole approach rests on is that stripped and re-padded elements do not change the delimiters. That is a property you can test, not a hope:

> For any ISA line — conformant, elements stripped to nothing, elements padded wide — `split_isa_line` returns the same four delimiters.

The test builds all three and asserts equality. Then an adversarial sweep: roughly 180,000 inputs — every byte of a valid interchange mutated at random, delimiters drawn from arbitrary bytes, the file truncated at every possible length — each checked against the invariants: no crash; a returned run starts with `ISA`, is a prefix of the input, is followed by `GS`, holds exactly sixteen separators; and any delimiter that comes back as usable is exactly one byte, with the element, component and terminator mutually distinct.

The sweep found a real bug. When an element separator had been dropped and `ISA11` swallowed the following field, `split_isa_line` returned a two-byte "repetition separator" with no complaint. A two-byte delimiter is not a delimiter. `ISA11` is a fixed one-byte field; the fix makes anything else in it — too long, alphanumeric, colliding with another delimiter — a reported finding with the repetition separator coming back as *none*. The invariant now holds: **a returned delimiter is either absent or exactly one usable byte.**

THE PATTERN

Read the element separator at the one offset that cannot move — the byte before the first variable-width field. Recover everything else from the split, anchored on the guaranteed separator count and the `GS` boundary, never a byte number. Make the segment terminator one byte by rule, so a two-byte terminator and a trailing newline stop being the same thing. Let the version number decide whether the fourth delimiter exists; do not

infer intent. And hold the fatal for the delimiters the interchange genuinely cannot be read without — report the rest, and let the segment that needs a broken delimiter be the one that fails.