Episode Book

EPISODE #5487

Why Your Spreadsheet Mangles José's Name

PUBLISHED	23 September 2026 at 13:02 UTC
DURATION	25:48
LISTEN	myweirdprompts.com/episode/utf8-encoding-mojibake-explained/
HOSTS	Corn, Herman, Hilbert
SCRIPT MODEL	deepseek-flash
VOICES	chatterbox-regular (chatterbox-local)
PIPELINE	V5.2
BOOK GENERATED	19 October 2026 at 17:36 UTC

SYNOPSIS

Ever opened a UTF-8 CSV in Excel and watched José turn into "JosÃ©"? That garbled text is called mojibake, and it's the visible symptom of a character set mismatch hiding beneath every keyboard layout, font, and file format you touch. This episode unpacks the three layers people constantly collapse into one — code points, encodings, and graphemes — and shows why the encoding layer being wrong poisons everything above it. From ASCII's 1963 origins and UTF-8's genius backward compatibility, to the byte-length trap, the NFC/NFD normalization split that once made files vanish on macOS, and MySQL's years-long inability to store emoji, we trace how these failures reach real teams. The anchor is a story: a distributed operation preparing a several-hundred-person event across Israel, the UK, and New York, where a CRM export standardized on UTF-8 mangled the names of people about to receive an email. If you work with international data or collaborators, this is the layer you've been ignoring.

TRANSCRIPT



Corn

Here's what Daniel wrote in this week. He says we've spent a few episodes on multilingual TTS and speech to text, on the road bumps you hit when you code switch or try to get accurate output across languages. And underneath all of that, he says, is one thing nobody thinks about: what character set you're actually working in. He wants to talk about that layer. The modern standard is UTF-8, the predecessor is ASCII, and he's seen firsthand the kind of crazy-inducing failure modes that show up when people collaborate on documents without paying attention to the system.



Herman

He's got a story, doesn't he.



Corn

He's got a story. A few years back he was involved in a major technical operation preparing for an event in New York City, several hundred attendees. Distributed team, Israel, the UK, New York, all working out of a Google Sheets attendee list full of names from Norway, Spain, France, accents and diacritics everywhere. He eventually standardized the backend on Airtable, but by then he was left with the hopeless task of trying to sanitize a CRM export that some program had standardized on UTF-8 and, in the process, mangled the names of a lot of people who were about to get an email.



Herman

Hopeless is the right word for it. That's the part people don't understand.



Corn

So that's the shape of today. Why the character set is such a fundamental aspect of computing, and prior even to the keyboard layout variants most of us use. How things get complicated when Unicode runs into different world languages. And what you actually need to know if you're working with international data and international collaborators. Where do we start with that?



Herman

Start with what the thing is, because "character set" is one of those phrases everybody's heard and almost nobody can define. A character set is the mapping between the abstract symbols of written language and the bytes a computer stores. That's it. It's a lookup table. And it sits below everything else you touch. Below the keyboard layout, below the font, below the file format. Everything above it assumes it's correct.



Corn

Which is exactly why nobody notices it until it's wrong.



Herman

Right, and there are three levels here that people constantly collapse into one, and the whole episode falls out of keeping them separate. First, the code point. That's an abstract number assigned to a symbol. U plus zero zero E nine is é. Just a number, no bytes yet. Second, the encoding. That's how that number becomes actual bytes on disk. UTF-8, UTF-16, Latin-1, Windows-1252, all different answers to the same question. Third, the grapheme. That's what a human being looks at and calls one character.



Corn

And those three are not the same thing.



Herman

They are not the same thing, and that's where the bodies are buried. Here's the thesis for the whole episode: when the encoding layer is wrong, nothing above it can be trusted. Not the spreadsheet. Not the CRM export. Not the phonemizer in your text to speech system.



Corn

So the arc is mechanism first, then the failure pattern, then what it means for anyone actually shipping multilingual anything.



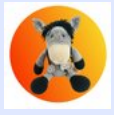
Herman

That's the arc.



Corn

Then start with the mechanism. What ASCII actually is, and how UTF-8 grew out of it.



Herman

ASCII is 1963. It defines one hundred and twenty-eight code points, zero through one hundred and twenty-seven, covering the English letters, the digits, punctuation, and a set of control characters that were originally for teletypes. It's a seven-bit encoding. That's the ancestor. And its ghost haunts every legacy system you will ever touch.



Corn

One hundred and twenty-eight characters. That's the entire world, as far as 1963 was concerned.



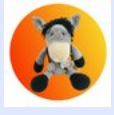
Herman

And it's worth being fair to it. For English, ASCII is complete. Every letter, every mark you need. The problem is that "English" is not "the world," and the people who designed the systems on top of ASCII spent about thirty years pretending otherwise.



Corn

So how does UTF-8 relate to it? Because that's the part I think people get fuzzy on.



Herman

UTF-8 is a strict superset of ASCII. The first one hundred and twenty-eight code points encode to identical single bytes. Byte for byte, an ASCII file is already a valid UTF-8 file. You don't convert anything. You don't migrate anything. You just declare it UTF-8 and it works.



Corn

And that's why it won.



Herman

That's exactly why it won. It's the only design that could be adopted without breaking thirty years of existing files. Every other candidate required a flag day. UTF-8 required nothing. You could roll it out incrementally, file by file, and the ASCII files kept working the whole time.



Corn

Which is a lesson that generalizes way past encoding.



Herman

It's the single most important lesson in standards, honestly. The format that wins is almost never the most elegant one. It's the one that doesn't require anybody to rewrite anything.



Corn

So ASCII is one byte per character, always. UTF-8 is not.



Herman

UTF-8 is variable width. One byte for anything in the ASCII range, two to four bytes for everything else. And that's the root of the first failure class, which is the byte length trap. Byte length is not character count. If you have code that slices a string by bytes, it will split a multi-byte character in half and produce garbage.



Corn

Give me the concrete version.



Herman

The concrete version is a form field with a character limit that counts bytes instead of characters. Somebody types a name with an accent in it, the field says you're over the limit, and they haven't typed anything unusual. Or the reverse. A truncation routine that cuts at byte two hundred and leaves half a character at the end of a name, and now the last glyph in the email is a black diamond with a question mark in it.



Corn

And this is the point where somebody in the meeting says "just strip the accents."



Herman

Somebody always says that. Hold that thought, because that's a whole separate failure pattern and it's the one Daniel ran into.



Corn

Then let's get the other structural facts on the table, because there are a few and they don't blur together. UTF-16.



Herman

UTF-16 is what Windows, Java, and JavaScript use internally. It has big-endian and little-endian variants, which means the same file can be read two different ways depending on which order you assume the bytes go in. That's why UTF-16 needs a byte order mark. The mark tells the reader which endianness it's looking at.



Corn

And UTF-8 doesn't need one.



Herman

UTF-8 does not need one, because there's no endianness to get wrong. It's a byte stream, not a sequence of two-byte units. And a byte order mark in a UTF-8 file is not standard. It's not forbidden exactly, but it's abnormal, and the standards people will tell you not to



Herman (continued)

do it. One practitioner put it more bluntly than I would and called it a Microsoft hack, because endianness has no impact on UTF-8 at all.



Corn

Noted, and we'll come back to the byte order mark, because that's the crux of the Excel problem and it's a trap that looks like a fix.



Herman

It absolutely is. Now the grapheme problem, which is the one that actually matters for anybody doing internationalized work. Take é. That can be stored as one code point, U plus zero zero E nine. Or it can be stored as two: a plain letter e, followed by a combining acute accent, U plus zero three zero one. Visually identical. Two completely different byte sequences.



Corn

So two files can look the same on screen and not be equal.



Herman

Not equal, not sort the same, not match in a search. And here's the line I'd put on a poster. Graphemes are almost always the closest thing to what people mean when they say "character." The human meaning lives at the grapheme level. The bytes live at the code point level. And any code that operates on bytes or code points while the meaning lives at the grapheme level is going to mangle things.



Corn

Which is why the advice is don't split or concatenate strings when you're doing internationalized work.



Herman

Don't do it. Do not pluck out the first grapheme of a name to make an avatar initial. Do not truncate a display name to twelve characters by counting bytes. It will look like nonsense, and the person whose name it is will see it.



Corn

There's a related thing here, normalization.



Herman

Normalization is the formal name for the problem I just described. The same visible character can be stored two ways, precomposed or decomposed, and Unicode defines forms that say which one you're using. NFC is precomposed. NFD is decomposed. And the historical split is that macOS enforced NFD, while most of the rest of the world used NFC.



Corn

And that produced real bugs.



Herman

It produced a nasty class of bug. A file named café, written on Linux in NFC, could appear to be deleted on macOS, because macOS was looking for the NFD form and the file it found didn't match. Git had to add special workaround code to handle it. And it's worse than the simple version, because modern macOS doesn't enforce NFD at the operating system level anymore, but Finder and a lot of Apple applications still use it, and it's Apple's own modified version. So you can type a filename identically and have the find command fail to locate it.



Corn

That's a very specific kind of madness.



Herman

Apple's filesystem had the same problem. APFS was initially unusable with most non-English languages until Apple added normalization-insensitive volumes, and that didn't land until macOS ten point thirteen.



Corn

So a filesystem that would lose your files if your language used accents.



Herman

Not lose them. Fail to find them, which from the user's chair is the same experience.



Corn

Let's do the astral plane, because that one has a great historical punchline.



Herman

Characters outside the Basic Multilingual Plane need four bytes in UTF-8. That's emoji, rare CJK characters, historic scripts. And MySQL famously destroyed those characters outright until 2010, when utf8mb4 arrived in version five point five point three.



Corn

The timing there is not a coincidence.



Herman

The timing is not a coincidence. Emoji started spreading outside Japan around 2008. Two years later, the database that couldn't store them got fixed. I'd like to say that's a coincidence, but that's not really likely.



Corn

So the lesson is that "UTF-8 support" is often a lie.



Herman

"UTF-8 support" is often a lie. A lot of systems say UTF-8 and mean "UTF-8 up to three bytes." They silently truncate anything above that. You don't get an error. You get a replacement character, or you get nothing, and the emoji or the rare character just isn't there anymore. That's the misconception to kill: supporting UTF-8 is not the same as supporting Unicode.



Corn

That's the theory. Now let's talk about what actually happens when this meets a spreadsheet and a team in three time zones.



Herman

This is where Daniel's story becomes the whole episode. Excel on Windows, opening a UTF-8 CSV that has no byte order mark, will not read it as UTF-8. It will read it as a legacy eight-bit codepage. Usually Windows-1252. And what you get is mojibake. José becomes J, o, s, A with a tilde, copyright sign. Björk becomes B, j, A with a tilde, paragraph sign, r, k.



Corn

That's the signature.



Herman

That's the double-encoding signature. It's unmistakable once you've seen it. The bytes were correct. The interpretation was wrong. And because the interpretation was wrong, the program rendered each byte as if it were a separate character in a different alphabet, and now you have two characters where there should be one.



Corn

So Daniel's attendee list, with Norwegian and Spanish and French names.



Herman

Every accented name in that list goes through that transformation the moment somebody opens the file the wrong way. And here's the thing about a distributed team. The person in New York opens it and sees mojibake. The person in Israel opens it and sees it fine. The person in the UK opens it and sees something else. And they're all looking at the same file.



Corn

So the meeting becomes an argument about whose screen is right.



Herman

And nobody knows the word mojibake, so nobody can even name what's happening.



Corn

Now, the fix everybody reaches for is the byte order mark.



Herman

The fix everybody reaches for is the byte order mark, and it's a trap. Adding a UTF-8 BOM does make Excel detect the encoding correctly. That part works. But the standard doesn't permit a BOM in UTF-8, and it can cause problems on other systems. It's a workaround that trades one class of failure for another.



Corn

And there's official guidance on this.



Herman

The UK Government's Open Standards Board looked at exactly this and recommended UTF-8 without a byte order mark. And their guidance for Windows Excel users is to use Import Text rather than double-clicking the file. That's the whole fix. Don't open the CSV. Import it, and tell the importer what encoding it is.



Corn

Which is a much less satisfying answer than a magic byte sequence.



Herman

It's a much less satisfying answer, and that's why nobody does it.



Corn

But the byte order mark isn't even the worst outcome. The silent re-save is.



Herman

The silent re-save is the real killer, and this is the part that explains why Daniel called his task hopeless. You open the CSV, it's already mangled, you fix a couple of names by hand, and then you press Ctrl+S. And Excel saves it in some arbitrary codepage, and now the mangling is written to disk. You end up with a document where the name column reads one, question mark question mark question mark question mark, two, Kevin Dub question mark i s.



Corn

Kevin Dub?is.



Herman

Kevin Dub?is. That's a real example from a practitioner describing exactly this. And the data is destroyed on save, often with no warning at all. The original bytes are gone.



Corn

Gone as in recoverable-with-effort, or gone as in gone?



Herman

Gone as in not reliably recoverable. And this is the part people don't believe until it happens to them. The transformation is lossy. Multiple distinct inputs can map to the same question mark. So when you see a question mark in the output, you cannot know what character was there. Was it an accent? A different letter entirely? A character from a script you don't recognize? There's no way to invert it, because the information isn't in the file anymore. It was thrown away at save time.



Corn

So the "sanitize the CRM export" job wasn't hard. It was impossible.



Herman

It was impossible, and it was impossible in a way that's invisible from the outside. Daniel's description of it as hopeless is exactly right. You can guess. You can look up the person on LinkedIn and see how they actually spell their name. But you cannot recover it from the file, because the file no longer contains it.



Corn

And there's a locale wrinkle on top of all this.



Herman

There's a locale wrinkle. Excel uses locale-dependent field separators. In some countries, a CSV is semicolon-separated, not comma-separated. So the same file opens differently depending on the user's locale, and a comma-separated file opened on a machine expecting semicolons comes in as one giant column.



Corn

So you have two independent failure pattern stacked on the same file.



Herman

Two independent failure pattern, and they interact. You can fix the encoding and still have the columns wrong, or fix the columns and still have the names mangled.



Corn

Now, Daniel specifically wanted the keyboard layout connection, because he says the character set is prior even to that.



Herman

He's right, and it's worth stating cleanly. A keyboard layout is just a mapping from physical keys to code points. QWERTY, AZERTY, Dvorak, Colemak, they're all answering the same question: when I press this key, which code point comes out? The layout determines which code points you can type easily. The encoding determines how those code points get stored. Two completely different layers.



Corn

So you can have a perfect AZERTY layout and still produce mojibake.



Herman

You can have a perfect AZERTY layout and produce mojibake all day long, because the layout did its job and the encoding didn't. And the reverse is true too. A US keyboard can type any Unicode character you want, through Alt codes, compose keys, or an input method editor. The layout is a convenience layer. It's not a constraint on the character set.



Corn

Which means the character set really is the more fundamental thing.



Herman

It's the more fundamental thing, and it's the one you can't fix by buying a different keyboard.



Corn

Alright. Connect this to the speech stuff, because that's where Daniel started and I don't want to leave it hanging.



Herman

This is the payoff for the last few episodes. Every multilingual text to speech system and every code-switching system assumes the text arrives correctly encoded. That's the assumption underneath everything. Garbage in the character set means garbage in the phonemes, which means garbage out of the speaker.



Corn

And there's current work that names this directly.



Herman

There's a paper from September this year, out of the NCMMSC conference, that diagnoses how low-resource multilingual TTS fails. And the failure list is exactly what you'd predict. Numbers, dates, named entities, long sentences, code-switched expressions, punctuation-related structures. And they introduce something called a Text Risk Score, which flags risky inputs before synthesis even runs.



Corn

So they're scoring the text before they try to speak it.



Herman

They're scoring the text before they try to speak it, which is the speech-side mirror of the spreadsheet problem. The structure of the input text determines whether the output is intelligible. And encoding is part of that structure. If your é arrived as two characters, the



Herman (continued)

phonemizer is looking at a letter e and a stray accent mark, and it has to guess.



Corn

And code-switching specifically.



Herman

Code-switching is an active frontier. F5-TTS advertises seamless code-switching capability. There's Indonesian-English work, Hindi and Indian-English accent work, and the thing they all have in common is per-word language identification. You have to know which language each token belongs to before you can pronounce it. And that presupposes clean, correctly-encoded text, because if the token is mangled you can't even tell which language it was.



Corn

The datasets are scaling up too.



Herman

SwitchLingua is four hundred and twenty thousand code-switched text samples across twelve languages, with over eighty hours of audio. CS-FLEURS covers a hundred and thirteen language pairs across fifty-two languages. The field is building the infrastructure to handle exactly the multilingual reality Daniel's attendee list represented.



Corn

So the attendee list was a preview of a research problem.



Herman

The attendee list was a preview of a research problem, and the researchers are solving it at the dataset level while the office is still solving it by hand in a spreadsheet.



Corn

Give me the practical guidance, then. What does somebody actually do.



Herman

Never assume UTF-8, but default to it. Detect, don't guess. If the first bytes of a file are F F F E or F E F F, that's UTF-16, not UTF-8, and you should stop and look at it.



Corn

Prefer UTF-8 without a byte order mark for interchange.



Herman

Prefer UTF-8 without a byte order mark. Use the mark only as a targeted workaround for Excel on Windows, and know what you're trading away when you do it.



Corn

Never round-trip international data through CSV in Excel.



Herman

Save as dot x l s x. Use Import Text for CSVs. That one habit would have saved Daniel's attendee list entirely.



Corn

Normalize explicitly.



Herman

Normalize explicitly, usually to NFC, at your system boundaries. And be aware that macOS and Finder may hand you NFD whether you asked for it or not.



Corn

Don't sanitize names by stripping non-ASCII.



Herman

Don't sanitize names by stripping non-ASCII. That's not sanitization, that's data destruction. If you need to transliterate, do it deliberately, do it reversibly, and keep the original alongside it. Never throw away the source.



Corn

Treat encoding as a schema decision.



Herman

Treat encoding as a schema decision, not an afterthought. Daniel's Airtable standardization worked because it fixed the encoding contract at the platform level. That's the same principle as choosing utf8mb4 in a database. You decide once, at the foundation, and everything above it inherits the decision.



Corn

Test with real international data.



Herman

Test with Norwegian ø and å, Spanish ñ, French é and ç. And crucially, test with four-byte characters. Emoji, CJK. Those are the ones that expose the three-byte truncation bugs, because a system that lies about supporting UTF-8 will pass every European test you throw at it and fail the moment somebody puts a flag emoji in a name field.



Corn

The three-byte lie.



Herman

It's the most common form of the problem and the hardest to catch, because it works perfectly until it doesn't.



Corn

The word you keep using is encoding. I want to push on it, because I don't think it's the word most people would use for this. Everyone I know would call it a character set. You keep saying encoding.



Hilbert

Character set is the wrong word for what went wrong.



Corn

Go on.



Hilbert

A character set is what the characters are. What bit you in the spreadsheet is how they were written down. Different thing. And the reason I know that is I spent about four months doing data migration for an insurance company. Mid-sized outfit, and they'd been running the whole book of business on a system that went in in the eighties and had never been replaced. My job was to move the policyholder records across. Names, addresses, beneficiary details. All of it.



Corn

The encoding was the problem.



Hilbert

Nobody knew what the encoding was. It was proprietary, it predated anybody caring, and there was no documentation. So I reverse-engineered it by hand. Sat there with a printed code table I found in a filing cabinet, matching byte values against it, one at a time, for weeks.



Corn

A printed code table.



Hilbert

Paper. Folded in thirds. Somebody had written their lunch order on the back of it. I kept it.



Corn

You kept it.



Hilbert

It's in a box. But here's the thing I came out to say. The problem isn't only that systems re-encode without telling you. It's that the people who built the original system had no idea they were making an encoding decision at all. They took whatever the vendor gave them. It was invisible to them. And it stayed invisible for thirty years, until it wasn't.



Corn

Then it was your problem.



Hilbert

Then it was my problem. There was one policyholder. Surname had a character in it that mapped to three different byte values depending on which decade the record had been entered. Three. The system had been migrated twice before me and nobody had noticed either time.



Corn

The same name, three different spellings, all in the same database.



Hilbert

All in the same database, all technically correct for the decade they were entered. I don't fully remember what we did with him. I think we picked one and moved on. Anyway. I have to go, the place on Emek Refaim closes at six and I need to collect something.



Corn

Hilbert, that code table is a perfect artifact of the problem. An encoding decision nobody knew they were making, written down on paper, kept in a box.



Herman

The three-byte surname is the whole episode in one record. Same name, three answers, and nobody in the building who could tell you which one was right.



Corn

Here's the thing I keep circling. If encoding is a schema decision, and the guidance is clear, and the failure pattern are this well documented, why do we still treat it as an afterthought? The same mojibake is happening in offices right now that was happening fifteen years ago.



Herman

It gets more critical, not less, as the speech systems scale. Every advance in code-switching TTS assumes clean text going in. The character set isn't a legacy concern. It's the substrate.



Corn

Which means the next time you open a CSV and see JosÃ©, you'll know exactly what happened. And you'll know it might already be too late to fix.



Herman

Thanks as always to our producer, Hilbert Flumingtop.



Corn

This has been My Weird Prompts, the human-AI collaboration podcast. If you want to send us something, email us at show at my weird prompts dot com. We'll be back soon.