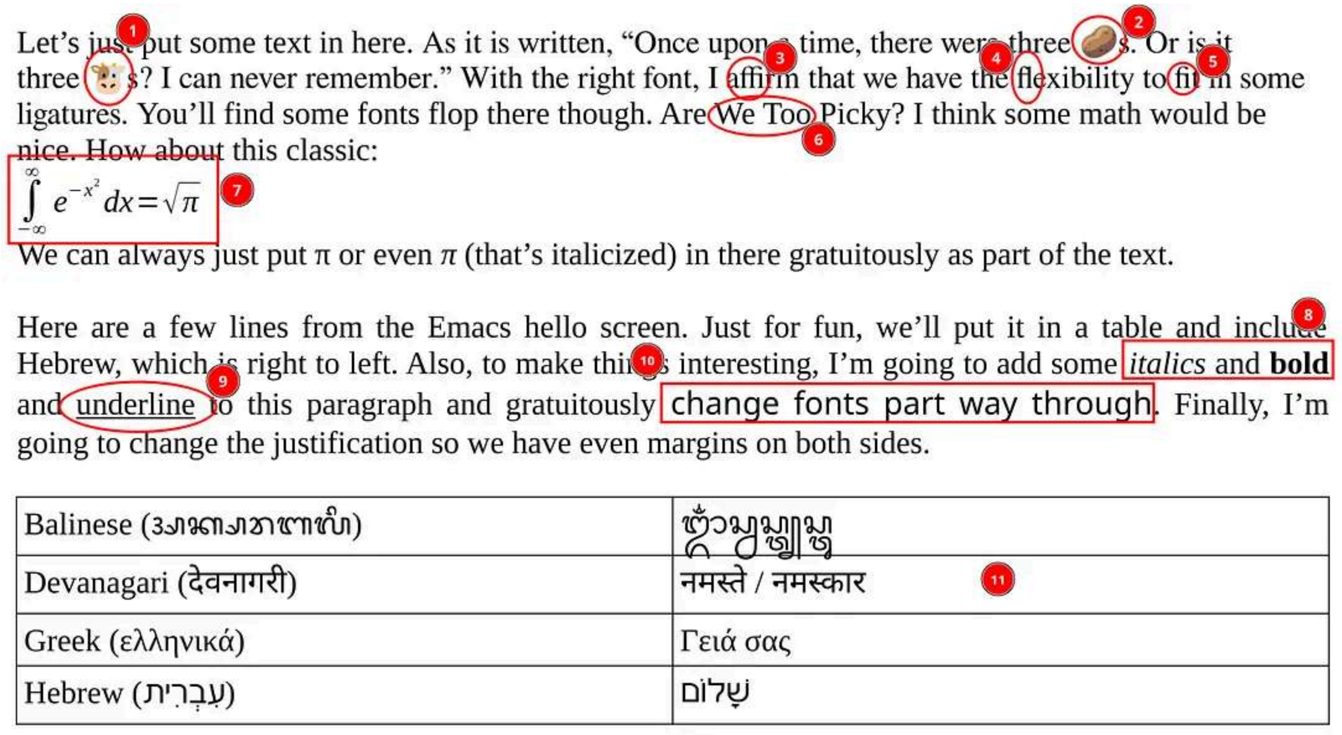
Text in PDF: Fonts and Spacing

Jay Berkenbilt Follow 11 min read · Sep 8, 2024

This post is Part 4 of a five-part series on understanding the representation of text in PDF. This picks up from Part 3, *Text in PDF: Unicode*, and steps you through the PDF code for some of the special features in that example file. Take a look at Part 1: *Text in PDF: Introduction*, for a synopsis of (and links to) all the parts as well as a general introduction to the series. As with all the posts in this series, I mark some paragraphs or parenthetical remarks with the 🔍 symbol when they contain deeper information that will be of interest to some readers but can be safely skipped without losing the flow of the material.

Here's the annotated copy of the PDF image as shown in Part 3:


Let’s just put some text in here. As it is written, “Once upon a time, there were three 🥔s. Or is it three 🐮s? I can never remember.” With the right font, I affirm that we have the flexibility to fit in some ligatures. You’ll find some fonts flop there though. Are We Too Picky? I think some math would be nice. How about this classic:

$$\int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi}$$

We can always just put π or even *π* (that’s italicized) in there gratuitously as part of the text.

Here are a few lines from the Emacs hello screen. Just for fun, we’ll put it in a table and include Hebrew, which is right to left. Also, to make things interesting, I’m going to add some *italics* and **bold** and underline to this paragraph and gratuitously change fonts part way through. Finally, I’m going to change the justification so we have even margins on both sides.

Balinese (ᬅᬓ᭄ᬱᬭᬩᬮᬶ)	ᬒᬁᬲ᭄ᬯᬲ᭄ᬢ᭄ᬬᬲ᭄ᬢᬸ
Devanagari (देवनागरी)	नमस्ते / नमस्कार
Greek (ελληνικά)	Γειά σας
Hebrew (עִבְרִית)	שָׁלוֹם

Annotated rendering of advanced.pdf

In this article, we’ll discuss

- 3, 4, 5: ligatures
- 6: kerning (and the even right margin in the second paragraph)
- 8: bold and italics
- 9: underlined text
- 10: multiple fonts

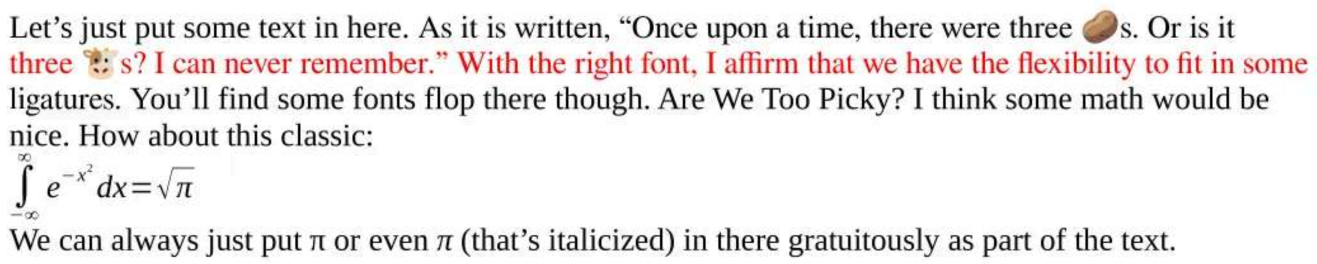
Using Color as an Exploration Tool

Throughout the remainder of this series, I will be showing snippets of code and screenshots of rendered PDF. Here are the first three text blocks, each surrounded by a `q`/`Q` pair. Line 1 in this excerpt is line 65 of the original, so add 64 to the line number if you’re following along.

Beginning of content stream for advanced.pdf

Suppose we want to find out where a particular bit of code is rendered on the page? An easy way to do it is to change its color. In this example, each text block is also in its own isolated graphics state (because of `q/Q`) and starts with an `rg` operator to set its color, so it's really easy. If we replaced the `0 0 0 rg` on line 13 (original line 77) with `1 0 0 rg`, what would happen? Recall that the arguments to `rg` are red, green, and blue values from 0 to 1, so `1 0`

`0` would be red. If you make this edit and save, you see the following in the viewer:



Let's just put some text in here. As it is written, "Once upon a time, there were three 🥔s. Or is it three 🐮s? I can never remember." With the right font, I affirm that we have the flexibility to fit in some ligatures. You'll find some fonts flop there though. Are We Too Picky? I think some math would be nice. How about this classic:

$$\int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi}$$

We can always just put π or even π (that's italicized) in there gratuitously as part of the text.

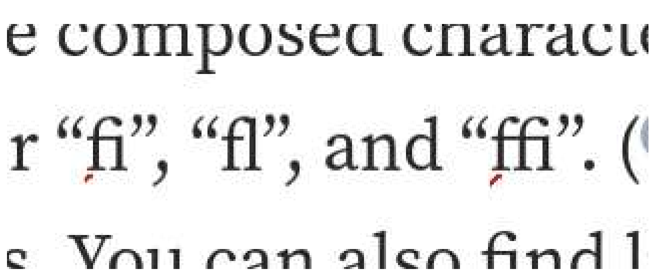
Rendered PDF with second line red

The second line of the text has turned red. This tells us that that particular text block corresponds to the second line. There are lots of other clues we can find, such as word lengths, font changes, and text position, but I personally find changing the color to be the easiest and fastest way to locate things. We'll mix this technique with other techniques throughout the article. In general, the easiest way to see if you're looking at what you think you're looking at is to change something. The trick is knowing what to change.

Ligatures

Now that we've located the code for the second line, we can look at the ligatures. A ligature is a single glyph that represents a joining of multiple characters. Some languages are filled with *composed characters,* which is when there's a different symbol for a combination of letters together than just the letters placed side by side. We'll see this with Devanagari in Part 5. English doesn't have composed characters, but high-quality fonts often contain ligatures for "fi", "fl", and "ffi". (🔍 You may also find ligatures for "ff", "ffl", and others. You can also find ligatures for æ and œ.) If you are reading this on medium, there's a chance that you would see all the ligatures

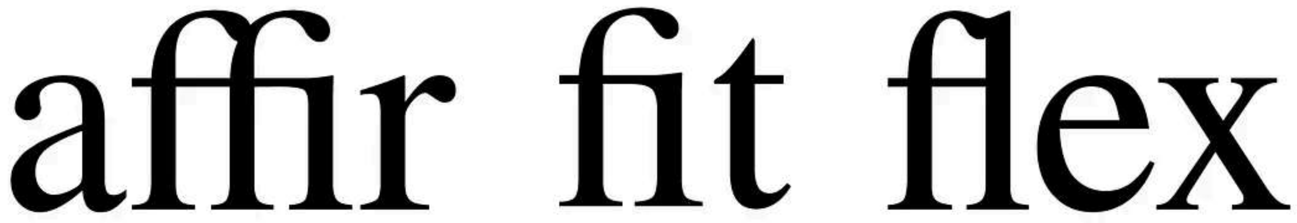
I just mentioned if you zoom in. Interestingly, as I write this, when in editing mode, I see the ligatures, but viewing the published text, I don’t. This is what I see in medium if I edit this post (complete with the red spelling checker marks):



e composed characte
r “fi”, “fl”, and “ffi”. (
s. You can also find l

Ligatures visible in medium while editing

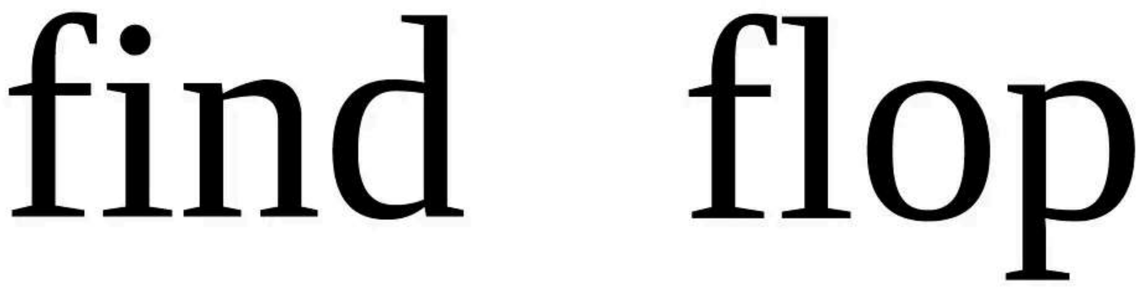
Let’s take a look at it them in the PDF. The sentence beginning with, “With the right font,” contains the words “affirm,” “flexibility,” and “fit” in a font that has ligatures. Here are bits of those words shown at a high zoom level:



affir fit flex

zoomed view of ligatures

Notice that, in “ffi”, the line through the two f’s is continuous and connects to the “i” and that there is no dot over the “i” — the serif at the top of the “f” does double duty. You can see the same in “fi”. In “fl”, you can see that the “l” is in direct contact with the “f”. In the next line, we have the phrase “find some fonts flop.” This is in a more basic font that lacks ligatures. Let’s take a look at that part zoomed in.

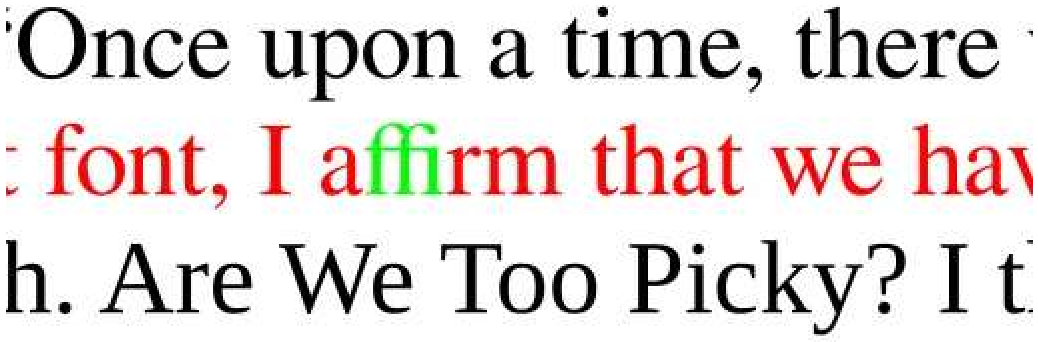


Here you can see a distinct “f” and “i” and a distinct “f” and “l”. Ligatures like this are the norm in high-quality, printed material. Many people think they improve the visual appeal of text, though you can have a well-designed font without ligatures that looks fine as well, of course.

How are the ligatures represented in the PDF? Let’s see. Notice that this text block is in font `/F2`. You can see that on line 15 above. In Part 3, I showed a table that mapped font labels to objects. From that table, we can see that `/F2` is object 11. Following the trail (click the links to follow along), on line 612 of the PDF, we see that the font’s `/ToUnicode` map is object 26, which starts on line 1042. The character map section starts on line 1061. Here is an excerpt of the map starting on line 1079:

Excerpt from /F2's /ToUnicode charmap

We can see something interesting at line 1089, which is line 14 above: the entries for `<1D>`, `<1E>`, and `<22>` map to multiple Unicode characters. What are these? `0066` is `f`, 0069 is `i`, and 006C is `l`. These are, in order, the "ffi," "fl", and "fi" ligatures. How can we be certain? On the previously mentioned line 15 of the content stream, if you scroll to the right, you can find `<050a011601111d>8<0712010513>` within the text. Notice in the middle, we have `1d` at the end of a binary string. Is this the "ffi" in "affirm"? Since `TJ` takes alternating strings and numbers, we can split this up into multiple `TJ` operators. If I replace the above with `<050a01160111>]TJ 0 1 0 rg <1d> Tj 1 0 0 rg [8<0712010513>`, we can see. What have I done there? I've split the original `TJ` around `1d`, creating three separate text showing operations, and I have surrounded the one that shows <1d> in color-changing commands that change the color green (`0 1 0`) and back to red again. If all is well, this should change just the "ffi" to green. Here's an updated image:



Once upon a time, there
font, I affirm that we hav
h. Are We Too Picky? I t

green ffi ligature

🔍 A better way to do this would be to use this fragment: `<050a01160111>]TJ q 0 1 0 rg <1d> Tj Q [8<0712010513>`, which uses a `q/Q` block. Then you wouldn't have to know the old color to restore it. Originally, I coded it that way, but I noticed that different PDF viewers handled it differently. I actually saw three variants: what I've pictured above, the "ffi" shown in green but the "rm" shown on top of it as if the text showing position were part of the graphics state, and the "ffi" entirely missing. It's interesting to note that, if you change the text color in LibreOffice and export to a PDF, it actually creates a whole new text block starting with its own `Td` operator rather changing color between two `TJ` operations. This issue arises from subtle differences to how graphics state is handled between PostScript and PDF as well as changes to the wording in the PDF specification over the years. Your best bet is to avoid `q/Q` blocks embedded inside text objects, but for manual exploratory purposes, do whatever is expedient and works.

This experiment shows that we have a single glyph, represented by a single code point (`1d`) that is mapped to three separate characters in `/ToUnicode`. That way, it displays the high-quality ligature glyph, but text selection and cut and paste still works as expected. The fact that we've replaced "ffi" with a ligature is explicit in the PDF in a way that it wouldn't be in a word processor or HTML file or anything else you're probably used to. That's because a PDF content stream is rendering instructions. Something like this has to be going on in all these other kinds of systems, but you don't see it. In PDF, it's right there for you to see.

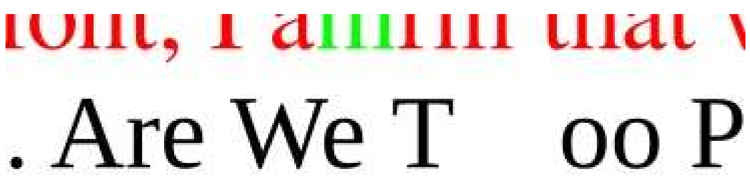
Kerning and Even Margins

In Part 2 of this series, I talked about kerning in my introduction to the `TJ` operator. In my basic PDF example, I eyeballed a text adjustment value for demonstration purposes. In this file, which was generated by LibreOffice, kerning was inserted by LibreOffice based on information in the font kerning tables. I wrote the phrase "Are We Too Picky?" because "We" and "Too" both have significant kerning after the "W" and "T", and since those are close together, I thought it would be relatively easy to find the kerning in the content stream. Also, the second paragraph contains even right margins (it is right-justified). In the block below, I've copied a text object that starts on line 90 and another that starts on line 192.

As we move down the `TJ` on the line above (you'll have to scroll to the right about a third of the), we encounter the fragment

```
1106>55<17100206>18<18>79<0206>18<19>69<0a0a
```

in a `TJ` that mostly strings and not very many numbers. There are two things that make this look like it's the kerning around "We Too". One is the relative frequency of numbers and those numbers being larger than some of the surrounding numbers, suggesting more prominent kerning, and the other is that this ends with `0a0a`, which is the same character repeated as in the word "Too." (The fact that `0a` is a low hexadecimal value and "o" is a common letter is another hint.) So I speculate that the 69 there is probably the kerning in "Too". Let's check it out. If I change this 69, meaning to move 0.069 times the font size to the left and make it -1000, which means to move the font size to the right, the display changes to this:



kerning modified manually

The confirms that the 69 was the kerning for the word "Too." Looking at the line, we see lots of long strings with a few numbers. These numbers are likely all related to kerning.

In the second text block, you will notice a lot more numbers, many of which are around -78 or -79. Also, there is usually the single-character string `<06>` between two numbers. Here's the excerpt of `/F1`'s `/ToUnicode` that we showed in Part 3.

/ToUnicode for /F1

On line 25, we can see that `06` maps to `0020`, which is the space character. What's happening here is that the PDF generator has divided up the amount of extra space that is required to make the margins even and has distributed it out around all the spaces in the line. By keeping the space character there, we get working text extraction. (🔍 PDF has word spacing parameters that can be set, but our example file doesn't use them — it explicitly adds the space. This is reasonable since it already has to know this information for its own UI, and it can get the information from `/ Widths` as provided in the font dictionary.)

Bold/Italics

There are only four bold characters in this document: the letters of the word "**bold**". If there's no kerning in that word, we should find a string `<01020304>` somewhere, and there's a block like that on line 212:

```
BT
532.75 602.4 Td /F6 12 Tf<01020304>Tj
ET
```

This is font `/F6`, which we have not seen before. Following the trail to its object, we can see that it's a subset of Liberation Serif-Bold. The character map from its `/ToUnicode`, from line 1360, shows just this:

```
4 beginbfchar
<01> <0062>
<02> <006F>
<03> <006C>
<04> <0064>
```

What are `0062`, `006F`, `006C`, and `0064`? You guessed it: "b", "o", "l", and "d". Now what if we wanted to edit this and change "bold" to "cold"? We'd be out of luck. We don't have the "c" from that font. This is yet another reason why free editing of text in PDF is generally impractical. You'd have to have access to all the fonts, and what would you do if you didn't have one? You could try to substitute another font, but it may or may not produce good results, especially when you deal with stuff like kerning. Programs that do allow editing of PDF files typically work this way, which is why results vary.

As an exercise, you could look for the italics. I won't paste it all here, but you'll find a `/F5` with the right pattern of code points just a few lines above. If you follow through to the `/ToUnicode map`, the character table starts at line 1299. Here it is:

```
10 beginbfchar
<01> <0065>
<02> <0078>
<03> <0064>
<04> <03C0>
<05> <0069>
<06> <0074>
<07> <0061>
<08> <006C>
<09> <0063>
<0A> <0073>
```

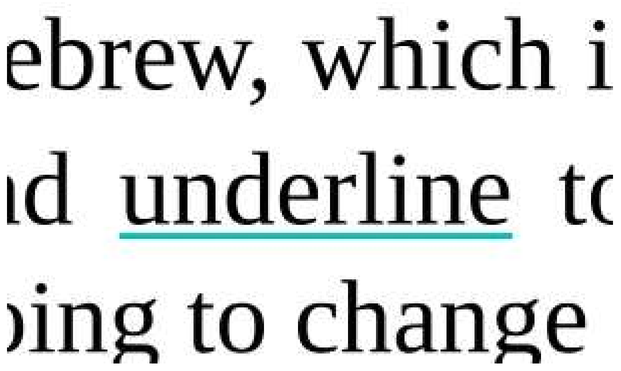
If you look closely, you'll see that all of these except `<04>` map to something between `0061` and `007a`, which are lower-case English letters. `<04>` maps to `<03C0>`, which is π. I'll come back to the italicized π in Part 5.

Underlined Text

We had new fonts for bold and italics. What about underline? Well, it turns out you don't need a font for that. Underlined text is produced by just drawing a line under some letters. We should be able to find the underline not far from bold and italics by looking for an `re` (rectangle) or an `l` (line — that's a lower-case ell) operator. At line 226, we have the following:

```
q 1 0 0 1 79.35 586.95 cm
0.7 w 0 0 0 RG
0 -1.4 m 45.3 -1.4 l S
Q
```

Again, there is no text here, but we do have a line that's 45.3 points wide and lies 1.4 points below the baseline. This time, we have an `RG` operator — that's with capital letters. The `RG` operator sets the *stroke* color, while the `rg` operator sets the *fill* color. The stroke color controls outlines and lines. The fill color is for what's in the inside of a shape, and that includes glyphs. (🔍 "Type 1" fonts, which these are, include information about glyphs using a specialized subset of regular vector drawing commands, so in the end rendering text is just drawing graphics. That's why the PDF spec refers to these as "font programs.") Let's see what happens if we change the color of the line. Using `0 .75 0.75`, a turquoise color, we get this:



turquoise underline

How did the PDF creation software know to make the line 45.3 points wide? There are several ways, but it would work to just add up the numbers in `/Widths` for all the characters in the string.

Ready for Part 5

Now that you've seen some examples of spacing, font changes, and even a few drawing commands, it's time to move onto Part 5, *Text in PDF: Non-Latin Alphabets*. In that article, I'll examine the mathematical formula, non-Latin text, and emoji and will wrap up with some closing thoughts about why it all works like this.

Pdf Pdf Text Extraction Unicode

Written by Jay Berkenbilt

128 followers · 3 following

Follow

Jay is a software engineer/architect with a focus on low-level coding and infrastructure. He is the creator of qpdf, an open source PDF manipulation library.

Responses (2)

Mark Tan

What are your thoughts?

axell badillo cuevas
Jun 6

amazing work, thank u!

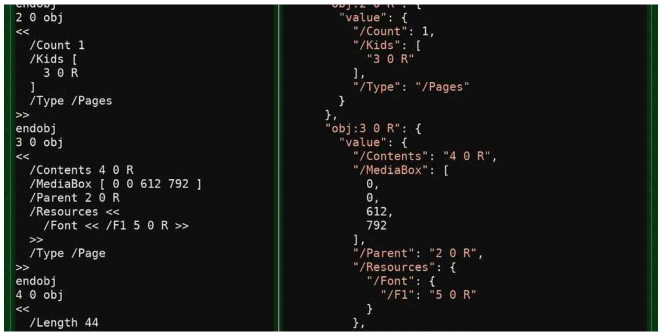
1 Reply

Andrii Maliuta
Sep 21

Thank you!

Reply

More from Jay Berkenbilt



```
2 0 obj
<<
  /Count 1
  /Kids [
    3 0 R
  ]
  /Type /Pages
>>
endobj
3 0 obj
<<
  /Contents 4 0 R
  /MediaBox [ 0 0 612 792 ]
  /Parent 2 0 R
  /Resources <<
    /Font << /F1 5 0 R >>
  >>
  /Type /Page
>>
endobj
4 0 obj
<<
  /Length 44
```

```
    "value": {
      "/Count": 1,
      "/Kids": [
        "3 0 R"
      ],
      "/Type": "/Pages"
    }
  },
  "obj:3 0 R": {
    "value": {
      "/Contents": "4 0 R",
      "/MediaBox": [
        0,
        0,
        612,
        792
      ],
      "/Parent": "2 0 R",
      "/Resources": {
        "/Font": {
          "/F1": "5 0 R"
        }
      },
```

Jay Berkenbilt

Examining a PDF File with qpdf

Introduction

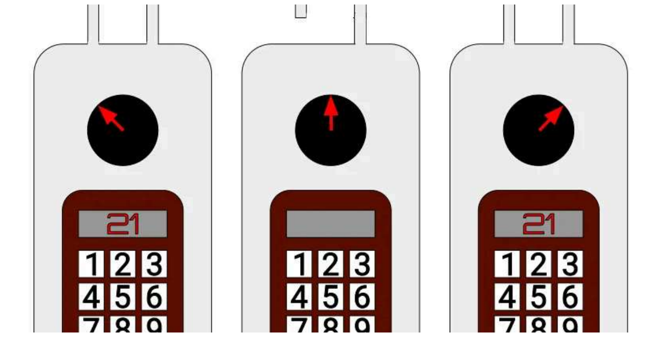
Nov 7, 2022 94 1

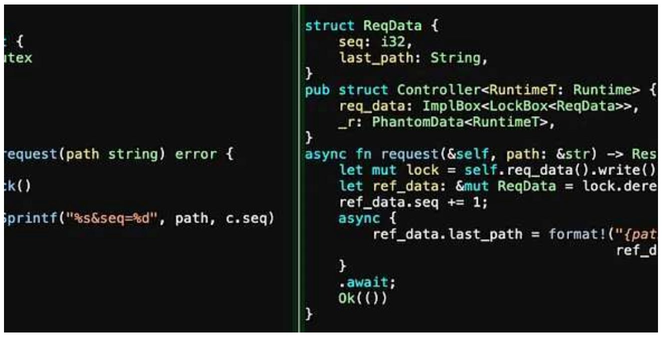
In AWS in Plain English by Jay Berkenbilt

Rotating Secrets with AWS RDS Proxy

AWS makes it easy for you to rotate your RDS credentials using SecretsManager. But what …

Sep 26, 2023 25 4





```
t {
utex

request(path string) error {

k()

Sprintf("%s&seq=%d", path, c.seq)
```

```
struct ReqData {
    seq: i32,
    last_path: String,
}
pub struct Controller<RuntimeT: Runtime> {
    req_data: ImplBox<LockBox<ReqData>>,
    _r: PhantomData<RuntimeT>,
}
async fn request(&self, path: &str) -> Res
    let mut lock = self.req_data().write()
    let ref_data: &mut ReqData = lock.dere
    ref_data.seq += 1;
    async {
        ref_data.last_path = format!("{pat
                                        ref_d
    }
    .await;
    Ok(())
}
```

Jay Berkenbilt

Understanding Why PKI Works

A Mental Model

Jan 14, 2024 3 2

In Rustaceans by Jay Berkenbilt

From Go to Rust 2: Generic async Lock

Generic, async Runtime-Agnostic Locks and impl Types

Jan 3 109 1

See all from Jay Berkenbilt

Recommended from Medium

Zubair Idris Aweda

Getting Started With Alpine.js

Alpine.js is a rugged, minimal tool for composing behavior in your markup. Think o…

Jul 1 2

Lambert W.

Web Scraping with Claude AI: Python Tutorial

The web contains vast amounts of unstructured data. Product catalogs, news…

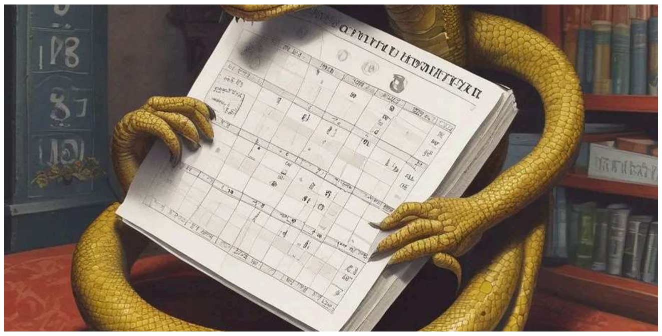
Nov 19 7

The CS Engineer

Forget JSON — The Future of Fast APIs Is Already Here

JSON is killing throughput and your developer patience.

Oct 30 489 7

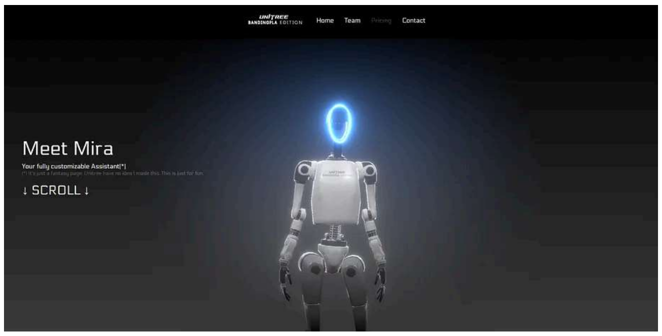


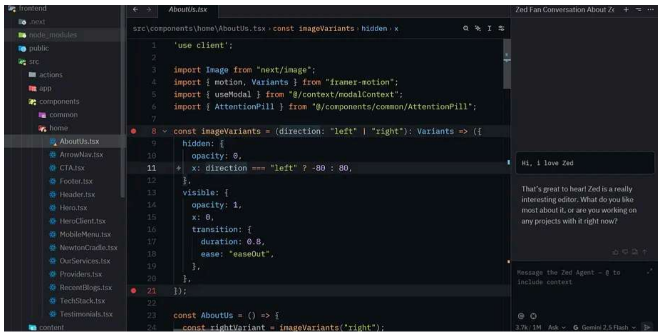
In Top Python Libraries by UnicornOnAzur

Create your own calendar with highlighted dates in Matplotlib

Show dates of interest in a month in a similar style as the Strava month overview using the…

Sep 10 25





```
'use client';

import Image from "next/image";
import { motion, Variants } from "framer-motion";
import { useModal } from "@/context/modalContext";
import { AttentionPill } from "@/components/common/AttentionPill";

const imageVariants = (direction: "left" | "right"): Variants => ({
  hidden: {
    opacity: 0,
    x: direction === "left" ? -80 : 80,
  },
  visible: {
    opacity: 1,
    x: 0,
    transition: {
      duration: 0.8,
      ease: "easeOut",
    },
  },
});

const AboutUs = () => {
  const rightVariant = imageVariants("right");
```

Hi, i love Zed

That's great to hear! Zed is a really interesting editor. What do you like most about it, or are you working on any projects with it right now?

Bandinopla

Scroll Driven presentation in ThreeJs with GSAP

Have you ever landed on a website where the scroll itself feels alive — every movement…

In Coding Nexus by Minervee

Not Cursor, Claude or VSCode, This Is My New Favorite Code Editor

Native AI code completion with a lightning fast code editor for developers

Aug 26 3 1

Nov 18 436 14

See more recommendations