Open in app ↗

Medium Search Write

The perfect gift for readers and writers. **Give the gift of Medium**

Text in PDF: Basic Operators

Jay Berkenbilt Follow 9 min read · Sep 8, 2024

10

This post is Part 2 of a five-part series on understanding the representation of text in PDF. In this article, I go over basic text positioning and text showing operators with an example PDF file.

The previous part, *Text in PDF: Introduction*, contains background information as well as an introduction to the overall series and a few links to some reference materials. Check it out for a synopsis of all the parts. This article assumes familiarity with the material covered in the introduction. All articles in this series use the convention of marking 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.

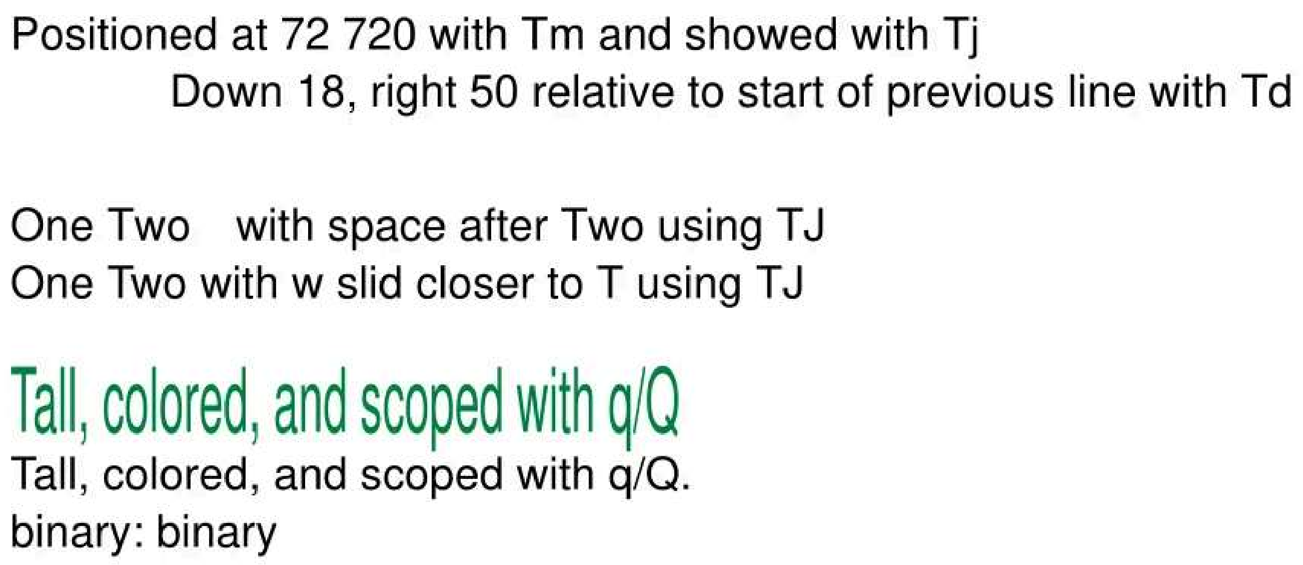
Getting Started

Throughout this article, I will be illustrating concepts using a simple PDF file. In the jberkenbilt/pdf-text-blog GitHub repository, I have stored the PDF file itself along with a textual representation of the PDF file. The textual representation is the PDF file tweaked slightly so GitHub will display it as a text file. It basically looks like what you would see if you opened the PDF file in a text editor that can safely handle binary files, such as GNU Emacs. As I explain concepts and relate them to specific blocks of PDF code, I include links to sections of the companion text file. I invite you to follow along by opening those links, but this is optional as I have embedded fragments of PDF and screenshots of the rendered PDF viewer as needed.

🔍 If you are a Linux user and are comfortable with emacs, you can open the PDF file in emacs and use evince, the GNOME Document Viewer, to load the PDF file. Then, you can save changes in the PDF and see them immediately reloaded in evince. The evince document viewer is forgiving about incorrect stream lengths and byte offsets, and it automatically reloads when the file is changed, so it's a great tool to use when exploring PDF files in an editor. There are other viewers that also automatically reload. I mention evince because of its broad availability across Linux distributions. You can also use emacs's doc-view mode, which is what you get by default when open a PDF in emacs. From there, you can use `ctrl-C ctrl-C` to toggle back and forth between the code view and the rendered view of the PDF file as long as you have ghostscript installed.

Basic PDF File

If you open basic.pdf in a viewer, you will see something like this:



Positioned at 72 720 with Tm and showed with Tj
Down 18, right 50 relative to start of previous line with Td

One Two   with space after Two using TJ
One Two with w slid closer to T using TJ

Tall, colored, and scoped with q/Q
Tall, colored, and scoped with q/Q.
binary: binary

basic.pdf as shown by a PDF reader

If you open this in a binary-safe text editor, you will see the text of the PDF code, starting with this:

```
1   %PDF-2.0
2   %¿÷¢þ
3   %QDF-1.0
4
5   1 0 obj
6   <<
7     /Pages 2 0 R
8     /Type /Catalog
9   >>
10  endobj
11
12  2 0 obj
13  <<
14    /Count 1
15    /Kids [
16      3 0 R
17    ]
18    /Type /Pages
19  >>
20  endobj
```

pdf-basic-line-1.txt hosted with ❤ by GitHub view raw

The first thing to do is to find the content stream for page 1. We could start at the document catalog and follow a trail to the content stream of page 1, but since this file is saved in QDF format, we can just search for the comment `%% Contents for page 1`, which can be found on line 42. The content stream is the text between the `stream` and `endstream` keywords. Here is the text of the content stream:

Throughout the rest of the narrative, I will refer to parts of the content stream based on the line number in the above text. In the actual PDF file and the tweaked text version in the repository, line 1 above is line 48. So if you're following along in the real file, just add 47 to my line numbers.

Text Blocks

The first thing we see here is a *text object*, which is a block of code delimited by `BT` and `ET`. You can see one text block that goes from line 1 to line 7, one from line 8 to 18, and one from line 19 to 24. It's very common for PDF content streams to have multiple text blocks since there are several aspects of the text state are scoped to a single text block.

Setting the Font

On line 2, we have `/F1 14 Tf`. This is the `Tf` operator, which sets the font, preceded by its two parameters, in this case, `/F1` and `14`. This is setting the current font to font `/F1` at 14 points. What is font `/F1`? You can look it up in the page's resource dictionary, which I'll do in Part 3 of this series, but for now, suffice it to say that this is pointing to the built-in Helvetica font (with PDF Doc Encoding, but don't worry about that for now). So we're just setting the font to Helvetica at 14 points. (🔍 Following along in the file? You can see `/F1` mapped to object 6 on line 35 and the definition of object 6 on line 79.)

Setting the Text Matrix

On line 3, we have the code `1 0 0 1 72 720 Tm`. This is a `Tm` operator, which sets the text transformation matrix. I talk about transformation matrices and the `Tm` operator in Part 1 of this series. For the simplified case of setting a text matrix that only scales and positions text, the first and fourth numbers are the horizontal and vertical scale factors, and the fifth and sixth numbers are the offset in points from the left and the offset in points from the bottom of the page. The second and third numbers are always 0 for this simplified case. Since a US letter-sized page is 612 by 792 points, this command is setting the current text position, which is where the bottom-left corner of the next piece of text will be, to 1 inch from the left and one inch from the top of the page. (720 = 792−72, and 1 inch is 72 points.)

Displaying a Simple Text String

Line 4 contains

```
(Positioned at 72 720 with Tm and showed with Tj) Tj
```

This is the simplest example of showing text. The easily-readable string "Positioned at 72 720 with Tm and showed with Tj" is showed using the `Tj` operator. What could be simpler? Well, unfortunately, real-world PDF files are seldom simple like this, but at least this way, you can see a simple example of displaying a text string.

There are some things that you might take for granted here if you are new to text in PDF content streams. In particular, the text string displayed in the rendered PDF appears intact in the PDF code. This only works because the text encoding is a standard encoding supported by PDF, the standard encodings are all supersets of ASCII, and the text editor or browser you are using to see the text knows how to display ASCII text. That is so common and obvious that you may have never even thought about it, but it is actually not the norm for PDF text. Sit tight — I'll come back to this in Part 3.

Starting the Next Line

Line 5 has the code `50 -18 Td`. The `Td` operator starts a new line of text positioned relative to the *beginning* of the previous line, which, in this case, was positioned using `Tm` on line 3. Recall that the positive `y` direction in PDF is *up*, so this statement starts the next line 50 points (about 0.7 inches) to the right and 18 points (🔍 about 1.3 times the font size, which is slightly more than single-spaced type, which is typically 1.2 times the font size) *below* the previous line. You can see that the next line is positioned relative to the beginning of the previous line as described.

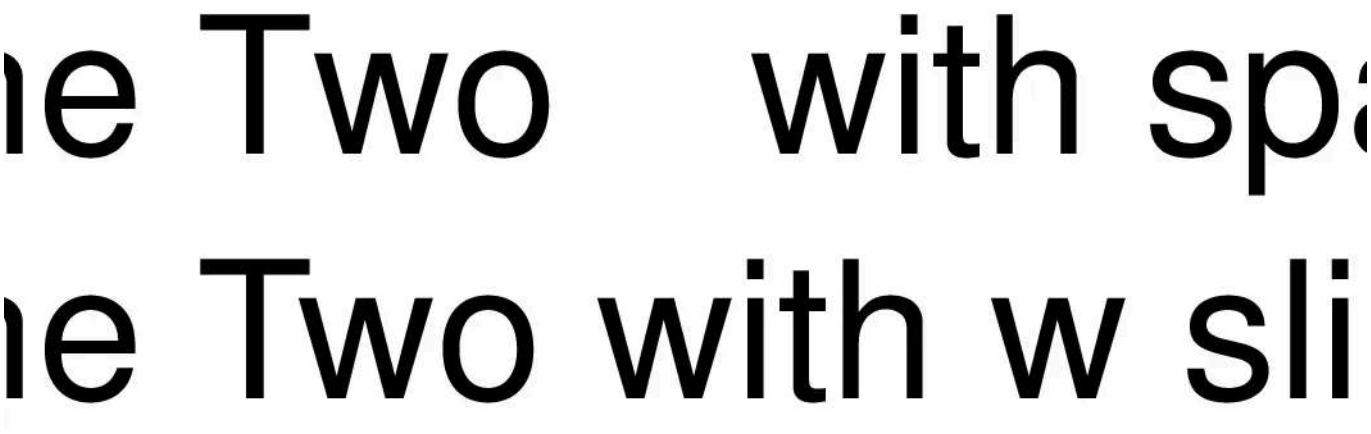
Line 6 is another simple text string showed with `Tj`, as you can see by comparing the PDF code and the rendered image.

Displaying a Text String with Offsets

Now we're ready to talk about the `TJ` operator. As mentioned in Part 1, the `TJ` operator is preceded by an array (delimited by square brackets) that contains a series of strings and numbers. Strings are displayed exactly as with `Tj`, and numbers move the current text position in the *opposite direction* of the flow of the text by the specified amount, which is units of 1/1000th of the font size. Here are lines 11 and 13:

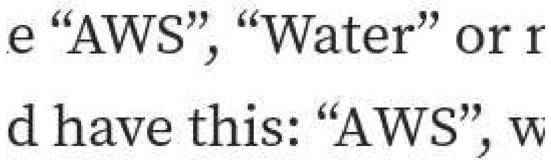
```
11: [(One Two) -1000 (with space after Two using TJ)] TJ
13: [(One T) 80 (wo with w slid closer to T using TJ)] TJ
```

On both lines, you can see a number between two strings. On line 11, the number -1000 appears before the word "with." This pushes the word "with" 14 points to the *right*. That's because these numbers move in the opposite direction of text flow, so to move right, we need a negative number. Line 13 moves the text position 80/1000 or 0.08 times the font size to the *left*, which is opposite the direction of text flow. Let's take a look at a zoomed in view of that part of the page:



Spacing with TJ

Let’s take a closer look. In the top line, you can see that the word “with” is shoved to the right as expected. What’s happening in the second line is more subtle and is an example of *kerning*, though in this case, I just made up a number — I didn’t actually get it from a font’s kerning table. If you look closely, you can see that the far left edge of the “w” in “Two” is just to the right of the far right edge of the “T”. In the second line, you can see that the “w” is slightly below the top of the “T.” Most people would consider the second version to look a little better. High-quality, proportional fonts in the hands of good typesetting software will include kerning like this. Look at just about any printed page to see this. In fact, zoom in on text in this blog post. If you’re reading this on medium, there’s a great chance that the word “Two” has kerning as would words like “AWS”, “Water” or numerous other examples. Without kerning, you would have this: “AWS”, which looks a lot less polished. (🔍 How did I do that? Between each pair of characters, I inserted the Unicode character with hex code point `200b`, which is the zero-width space.) Just in case it doesn’t look right on your browser or in however you are reading this, here’s an image of what I see on my screen:



Kerning defeated with zero-width space

Text Color and Graphics State

Moving onto lines 13 through 17. In line 13, we have the bare `q` operator, and in line 17, we have the `Q` operator. These operators are saving and restoring the graphics state, as discussed in Part 1 of this series. Line 14 is another `Tm` operator, but this time, the fourth number is `2`, not `1`. That means the text will be scaled vertically by a factor of 2. On line 15, we have something new: `0 0.5 0.25 rg`. The `rg` operator sets the *fill color*, which is the color used for the insides of shapes, including characters in fonts as well filled shapes in graphics, so you use `rg` to set the text color. The parameters for `rg` are red, green, and blue values ranging from 0 to 1. The color I have specified is a bluish green color (I’d call it teal). I’ve also snuck in a little manual kerning between the initial “T” and “all” in “Tall”. Notice that the color is reset at after the `Q` operator since color is part of the graphics state. Finally, the line right below the teal line is there to make it easier to see that the font was scaled only in the vertical direction: all the letters have the same width in both lines, so the text lines up. (If you are viewing the PDF file on a mac using Safari or the mac preview application, it’s possible that the tall, teal line may not be visible because of a bug in the interpretation of `q/Q` blocks inside text objects.)

Binary String Representation

Next, on line 23, I have the binary string `<62696e617279>`, which is the word “binary” since `62` is `b`, `69` is `i`, `63` is `n`, and so forth. This shows that regular strings and binary strings can be used interchangeably. (🔍 There’s a syntax for including characters > 127 in regular strings too: you can either embed the byte with that value, which will probably not be displayed correctly in most modern editors, or you can use backslash with octal, but I'm not going to talk more about that.)

Ready for Part 3

Still here? Did you actually make it through Parts 1 and 2? If you stopped here, you'd walk away with a basic understanding of text display in PDF, but the good parts are yet to come. Continue on to Part 3, *Text in PDF: Unicode*, to learn about font subsets and the representation of Unicode characters.

Pdf Pdf Text Extraction Unicode

Written by Jay Berkenbilt

128 followers · 3 following

Follow

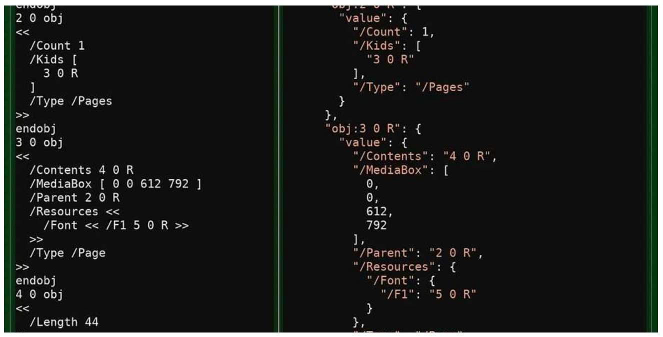
Jay is a software engineer/architect with a focus on low-level coding and infrastructure. He is the creator of qpdf, an open source PDF manipulation library.

No responses yet

Mark Tan

What are your thoughts?

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

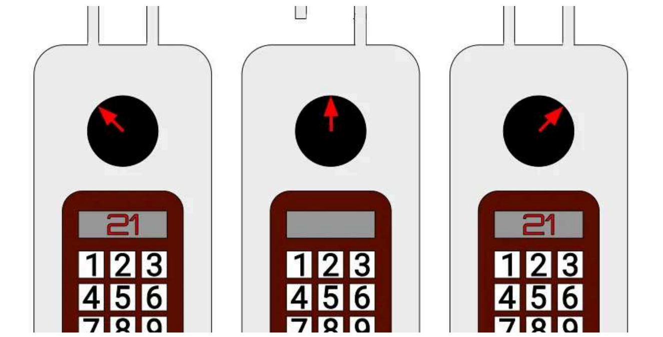
Nov 7, 2022 94 1

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 ...

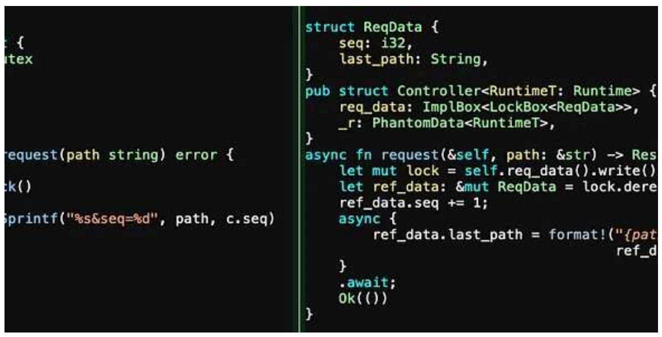
Sep 26, 2023 25 4



Jay Berkenbilt

Understanding Why PKI Works

A Mental Model



```
 {
tex

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(())
}
```

In Rustaceans by Jay Berkenbilt

From Go to Rust 2: Generic async Lock

Jan 14, 2024 3 2

Generic, async Runtime-Agnostic Locks and impl Types

Jan 3 109 1

See all from Jay Berkenbilt

Recommended from Medium

Lambert W.

Web Scraping with Claude AI: Python Tutorial

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

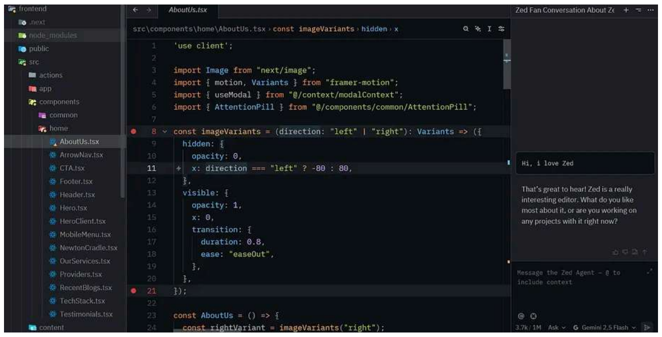
Nov 19 7

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



```
src\components\home\AboutUs.tsx › const imageVariants › hidden › x
'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?

Message the Zed Agent - @ to include context

3.7k / 1M

The CS Engineer

Forget JSON — The Future of Fast APIs Is Already Here

JSON is killing throughput and your developer patience.

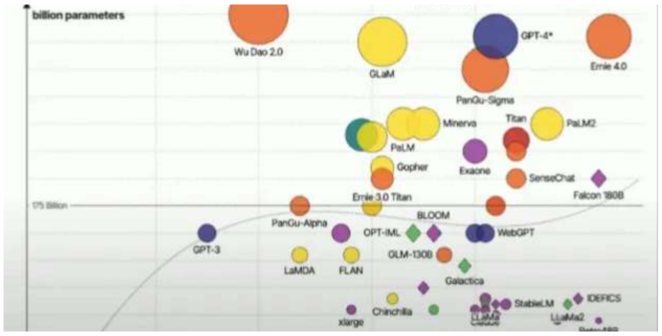
Oct 30 489 7

In 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

Nov 18 436 14



Shravan Kumar

Build a Small Language Model (SLM) From Scratch

At this current phase of AI evolution, any model with fewer than 1 billion parameters...

Jul 26 1.2K 33

Joe Osborne

4 Python Techniques I Use in All My Web Scrapers

Your scrapers can be 10x more efficient with a few improvements.

Dec 4 2

See more recommendations