Text in PDF: Introduction

Jay Berkenbilt Follow 13 min read · Sep 8, 2024

Welcome to this five-part series on understanding the representation of text in PDF. The target audience is a technical person who is comfortable with looking at code, though you don't have to be a programmer or have much PDF experience. The post gets more technical as it goes on, so feel free to skim until it reaches your level and stop when it gets to be too much.

Part 1, this post, covers some background knowledge that you should become familiar with to understand the remaining parts including hexadecimal numbering, Unicode, and the basics of PDF text operators and graphics state.

Part 2, *Text in PDF: Basic Operators*, uses a simple, hand-coded PDF to display text. It uses one of the built-in fonts and a standard encoding and has examples of the basic text operators.

Part 3, *Text in PDF: Unicode*, uses a PDF file generated by word processing software (LibreOffice) to demonstrate how arbitrary Unicode characters are represented with font subsets and custom encoding. After this part, you will have all the fundamentals to understand text in just about any PDF file.

Part 4, *Text in PDF: Fonts and Spacing*, demonstrates the technique of adding color commands to PDF code in a binary-safe text editor to match the code with the rendered file so you can find your way around. It uses the same PDF file as Part 3 to examine the code behind kerning, ligatures, font changes, underlined text, and spacing.

Part 5, *Text in PDF: Non-Latin Alphabets*, using the same sample PDF, takes a closer look at a mathematical formula, text in non-Latin alphabets with more complex writing systems, and emoji. It concludes with a reflection how it all works.

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

Sample PDF used in Parts 3 through 5

In this series, I sometimes dig a little deeper than you need to know. When I do this, I mark a paragraph or parenthesized statement with the 🔍 symbol. (🔍 I ripped this off from Don Knuth in the TeX Book where he uses a symbol for “dangerous bend” to mean the same thing.)

Other useful references:

- My earlier post: *The Structure of a PDF File*
- My earlier post: *Examining a PDF File with qpdf*
- From the PDF Association: *PDF Operators Cheat Sheet*
- The Wikipedia article on *Unicode*

Hexadecimal Numbering

This is the only math, and if you don’t follow the math, you can just think of hexadecimal numbers as character labels and move on. Don’t let it scare you off from the rest of the article.

Hexadecimal, often just called “hex,” is a base-16 numbering system that uses `a-f` as extra digits after `0-9` to represent the numerical values from 10 (`a`) to 15 (`f`). Since it’s base 16, that means `10` hex has the decimal value 16, `1f` hex is 31, `20` hex is 32, etc. Hexadecimal is convenient when working with byte-based values because a byte can contain values 0 through 255 decimal, which is `00` through `ff` hexadecimal. That means you can always see the exact number of bytes it takes to represent any hexadecimal value: one byte for every two digits. Leading zeroes are used in some cases (like in PDF files) to create an even number of digits. Just as with decimal, leading zeroes don’t change the value of the number. In this blog, you will see hexadecimal in two places: binary strings an

which are defined later in this post. If you don't understand the math behind this, just think of hexadecimal numbers as labels for characters, and remember about that two-digit per byte thing. Also: a byte is 8 bits, so 16 bits require two bytes.

PDF Syntax Refresher

My blog entitled *Examining a PDF File with qpdf* discusses PDF syntax in a broader context. Here's a quick refresher of what you need to know to read content streams, which is the part of a PDF that describes what is actually shown on the page. PDF syntax has the following kinds of objects:

Type	Description	Example
Null	the null value	`null`
Boolean	true or false	`true`
Number	integer or floating point	`123 , 3.14159`
Text string	delimited by parentheses	`(potato)`
Binary string	hex delimited by angle brackets	`<243f6a8885>`
Name	starts with `/` , used as keys or labels	`/Span`
Array	list of other types in square brakets	`[(One) 2 /Three]`
Dictionary	name/value pairs surrounded by `<<` and `>>`	`<</ActualText <feff03c0> >>`
Operator	short keyword, only in content streams	`BT , Tj , l , EMC`
Comment	`%` to end of the line	`%% Page 1`

pdf-syntax.md hosted with ❤ by GitHub view raw

Summary of PDF Syn

0 s remaining

Of the above, everything except operators can appear in the structural parts of a PDF. Operators can only appear in a content stream.

PDF Operators *follow* their parameters. For example, to display a simple text string, you would have `(potato) Tj`. In this case, the string `(potato)` is the parameter to the operator `Tj`. (🔍 This originates from PostScript, which has a stack of operators and operands. PDF doesn't use a stack in that same way, but operators still follow their operands.)

Transformation Matrices (Simplified)

PDF uses *transformation matrices* to perform arbitrary *linear transformations* on graphics or text. A full description of transformation matrices and linear transformation is out of scope for this post, so I'm presenting a simplified view here. You don't need math beyond basic arithmetic for this simplification.

In the PDF examples used in the remaining parts of this blog series, we only use transformation matrices to scale text and to position it on the page. Text can be scaled independently in the horizontal or vertical dimensions. The text transformation matrix (and also the graphics one) are expressed using six numbers. When transformation matrices are used for the simplified purpose of scaling and positioning, the numbers are

- horizontal scale factor
- zero
- zero
- vertical scale factor
- horizontal position relative to the left edge

- vertical position relative to the bottom edge

The basic unit in PDF is the point, which is 1/72 of an inch. This is the same point value used for specifying font sizes in word processing applications and in the printing industry.

In PDF, the *origin* of the coordinate system is at the bottom-left corner of the page, which is typical for mathematical coordinate systems. This differs from some screen-based graphics models that put the origin at the top-left corner. That means that you need to use *negative* offsets to move down the page.

🔍 If you're interested, let's go one layer deeper. (As mentioned at the beginning of the post, paragraphs like this that are marked with the 🔍 symbol provide deeper information and can be skipped without breaking the flow of the material.) In linear algebra, a transformation matrix is a 3×3 matrix that can be multiplied by a point to perform a linear transformation. A linear transformation can be viewed as a composition of operations (meaning combining them one after the other with a cumulative affect): *scale*, *translate*, *rotate*, and *skew*. Scaling multiplies dimensions uniformly along a specific direction by a scale factor. Translation adds a value to the horizontal and/or vertical position. Rotation causes a uniform rotation of the coordinate axes by any angle. Skew independently tilts the horizontal and vertical axes. When constraining linear transformations to a plane, the rightmost column from top to bottom is always `0 0 1`. The six numbers that are operands to the transformation matrix operators represent the other six numbers in that matrix. When you restrict the allowable transformations to just scaling and translation, the six numbers simplify to the description in the previous paragraph. For more, I would refer you to the PDF spec or your favorite linear algebra text.

Text Layout Operators

PDF has a rich collection of text positioning and layout operators. These perform functions such as altering character, word, and line spacing, applying a transformation matrix to text, or advancing the current text position to the beginning of the next line of text as an offset from the previous line. The example PDF files used in these posts use only two of the text layout operators:

- `hs 0 0 vs x y Tm`: The `Tm` operator sets the text transformation matrix as described above, where `hs` and `vs` are the horizontal and vertical scale factors and `x` and `y` are the horizontal and vertical location of the position where the next piece of text will be displayed.
- `dx dy Td`: The `Td` operator moves the current text position to the right by `dx` points and *up* by `dy` points relative to the beginning of the previous line. Note that `dy` moves up, so we typically see negative values for `dy`. Note also that `Td` sets the position of the line relative to the *beginning* of the previous line, not relative to current position, which means the parameters to `Td` are independent of the text of the previous line. Both the `Td` and `Tm` operators set the text position and the position of the current line. (🔍 The above applies to left-to-right text mode. PDF can support other modes, but our examples use left-to-right mode even when rendering intervening right-to-left text.)

There are other ways to alter the way text is displayed. For more detail, I refer you to the excellent *PDF Operators Cheat Sheet* available from the PDF association or to the PDF specification.

Text Display Operators

There are two main text display operators in PDF: `Tj` and `TJ`. Operator names are case-sensitive. The first one has a lower-case `j`, and the second one has an upper-case `J`. The `Tj` operator shows a single string of text at the current text position and advances the position by the width of the text. The `TJ` operator takes an *array* of strings and numbers and handles them sequentially. If the item is a string, it displays it at the current position and advances the position by the width of the string just as with `Tj`. If it is a number, it moves the current text position *in the opposite direction of text flow* by a unit that is measured in 1/1000ths of the current font size. For example, `[(T) 80 (wo)]` TJ would display the word `Two` but would slide the `wo` to the *left* by a distance equal to 0.08 (that's 80/1000) times the font size. The `TJ` operator is used to implement *kerning*, which is what the previous example did. Kerning is making fine adjustments to character positions to improve the visual appeal of the text. High-quality proportional fonts usually include kerning tables that specify these adjustments for pairs of characters. The `TJ` operator can be used for any purpose that requires making adjustments to text position. Another common use case is for creating even text margins (known as *justification*). The example PDF file used in Part 3 of this series has both kerning and justification.

In addition to these operators, there are other "shortcut" operators that display text and move to the next line at all once or that do those things while making changes to the text state. The previously-mentioned *PDF Operators Cheat Sheet* is a great place to start.

Graphics State

PDF has the concept of *graphics state*. We don't need to go too far into it, but for our purposes, we can think of graphics state as containing things like the current color, fill pattern, transformation matrix, clipping region, and line width along with a handful of other things th

text and graphics operators. There are some things that are not in the graphics state, like the current font, the current text position, which is where the next bit of text will be displayed, or the current point, which is where the next graphics operation will be displayed.

PDF has a pair of operators: `q` and `Q`, which isolate changes to graphics state by saving and restoring the graphics state. It is extremely common for PDF content streams to include many `q/Q` blocks. For example, the usual way to set the text color is to save the graphics state with `q`, then set the *fill color* (which is the color inside a shape, as opposed to the *stroke color*, which is for lines or outlines), then do whatever you want to be done in that color, and then restore the graphics state with `Q`. The PDF examples in Parts 2 and 3 of this series make frequent use of this technique.

🔍 Is the text position part of the graphics state? This turns out to be a complicated question. Based on my reading of the PDF Spec (ISO 32000–2:2020(E)), they are not supposed to be. However, different PDF viewers behave differently in this respect. If you show a text string inside an existing text block and surround it by `q/Q` as in `(A) Tj q (B) Tj Q (C) Tj`, some viewers will show the C on top of the B, and some will show it after. I've even encountered some that drop the B entirely. So, regardless of what the PDF spec says about this, it's best not to rely on whether `q/Q` save and restore the text position. In PostScript, the `gsave` and `grestore` operators, which do the same thing, do explicitly save and restore the current point, but PostScript has just one current point position that applies to everything, whereas PDF maintains the separate concept of text position. Also, in PostScript, it is possible to query the current point and save and restore it manually, while this is not possible in PDF. It's remarkable that something as simple as this can be treated differently by different PDF viewers. I think this is because the PDF specification is big and complex and

ambiguous ways or remains silent about certain questions. Notably, Adobe Reader considers the text position to be part of graphics state, and in the real world, what Adobe does matters more than what the ISO standard says.

Character Encodings

A *character encoding*, also sometimes known as a *coding system*, is a system that assigns a numerical value to each character in the system. The value for a character is called its *code point*. For example, the ubiquitous ASCII coding system assigns a number between 32 and 126 to every printable character on a standard US keyboard and uses most of the other values between 0 and 127 to represent other things like form feed, backspace, tab, the escape key. There's even a value to represent an invalid character. The Unicode coding system aims to provide a mapping for every character in every language as well as a wide range of symbols for just about anything you would ever see in print. Even emoji are included in Unicode. ASCII is a subset of Unicode. In other words, if a character has an ASCII value, its Unicode value is the same as its ASCII value.

The terms *character encoding* and *coding system* are also used to refer to the way a character's code point is represented as a sequence of bytes. Some coding systems, such as ASCII, contain only values below 256 and can therefore be represented with a single byte per character. These are known as *single-byte* coding systems. There are numerous single-byte coding systems that extend ASCII by assigning other characters to the values between 128 and 255. PDF includes three of them: one called *PDF Doc Encoding*, and one each for common coding systems used on older versions of Windows and Mac. (These Mac/Windows coding systems are still in common use, though they have largely been replaced by Unicode on modern systems.)

🔍 Prior to Unicode, there were many single-byte coding systems that used characters in the range from 128 to 255 to represent additional characters. Many of these systems are still in common use. One such system is called *ISO-Latin-1*. It is common in the English-speaking world and contains several accented characters common in widely spoken European languages as well as extra symbols that don't appear on the US keyboard, like ¿, ¢, ×, and ÷. I mention this one because all values below 128 in ISO-Latin-1 are the same as ASCII, and every value below 256 in Unicode matches ISO-Latin-1. If you are older than a certain age and/or speak a language other than English, Spanish, French, or German, you have almost certainly seen other single-byte coding systems, and if you use text editors, you've likely had the experience of opening a file in a coding system that the editor wasn't expecting and seeing gibberish.

Unicode, UTF-8, and UTF-16

Unicode has more than a million valid code points, and so it takes more than one byte to represent most valid Unicode characters. Unicode is divided into "planes" that contain a collection of related characters. There is something called the *basic multilingual plane,* which contains code points from 0 to 65,535 and covers all the characters used in the world's written languages along with numerous common symbols. (What falls outside the basic multilingual plane? Almost everything else: musical notation, emoji, and so much more.) The number 65,535 is the largest number that fits in two bytes. There are several ways of encoding Unicode characters as bytes. PDF uses a format called big-endian UTF-16 (UTF-16-BE), which represents each character in the basic multilingual plane as a two-byte sequence consisting of the four hexadecimal digits that represent the hexadecimal Unicode code point. For example, the capital letter `A` has ASCII hexadecimal value `41`, which is decimal value 65. In UTF-16-BE, that would be written as `0041`. The Greek letter π has hexadecimal code point `3c`

would be represented as `03c0` in UTF-16-BE. Characters with higher codes are represented by a four-byte pair of two-byte values known as a *surrogate pair*. For a little more nuance, read on. For the full story, I recommend the Wikipedia article on *Unicode*.

🔍 The two most common multi-byte coding systems for representing Unicode are UTF-8 and UTF-16. UTF-8 uses a single byte for characters below 128 and has an unambiguous way to represent all valid code points (which go up to `10FFFF`) in no more than four bytes. (Technically, the format allows characters that go longer, but they are not actually valid Unicode characters.) UTF-16 represents characters as either two-byte sequences or four-byte *surrogate pairs* of two-byte sequences. Since a single character is represented as two bytes, there are two variants of UTF-16: one for each possible byte order. For example, the code for π, which is `03c0`, could be represented as `03` followed by `c0` or as `c0` followed by `03`. The `03,c0` order is called big-endian because the high byte is first, and the order `c0,03` is called little-endian because the low byte is first. Why is this even a thing? It boils down to the way 16-bit and larger values are represented by various CPUs. In particular, Intel CPUs represent larger numerical values in little-endian byte order, and ARM CPUs, as well as the old Motorolla chips that the first Macintosh computers used, are big-endian.

🔍 The following table shows four characters with their hexadecimal code points and how they are represented in those three coding systems.

Char	Hex	UTF-8	UTF-16-BE	UTF-16-LE
W	57	57	0057	5700
π	3c0	cf 80	03c0	c003
•	2022	e2 80 a2	2022	2220
🥔	1f954	f0 9f a5 94	d83e dd54	3ed8 54dd

unicode-characters.md hosted with ❤️ by GitHub view raw

🔍 Here are a few things to notice:

- Only `W` fits into a single byte.
- The π and • characters are both part of the basic multi-lingual plane. You can see the hex value intact in the UTF-16-BE encoding, which is the one that is used in PDF.
- Notice that π and • fit in two bytes in UTF-16, but • requires three bytes in UTF-8. UTF-8 is usually more compact, particularly for English text, but there are some characters that take more bytes to represent in UTF-8 than UTF-16.
- For the potato (🥔), notice that each individual two-byte sequence is independently reversed between UTF-16-BE and UTF-16-LE. That's because each 16-bit value is treated independently.

Ready for Part 2

Still here? I hope all that was informative. Armed with the above information, you should have what you need to continue with Part 2, *Text in PDF: Basic Operators*.

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 (3)

Mark Tan

What are your thoughts?

Michael Klink
Jan 24

> Is the text position part of the graphics state?

You may want to take a look at https://github.com/pdf-association/pdf-issues/issues/368 for this. In short:

* Up to ISO 32000-2:2017 **q** and **Q** were special graphics state operators and hence forbidden in text objects. As text position information are… more

1 1 reply Reply

0 s remaining

Michael Klink
Jan 24

Your "Text in PDF" series is a nice introduction into the topic.

Reply

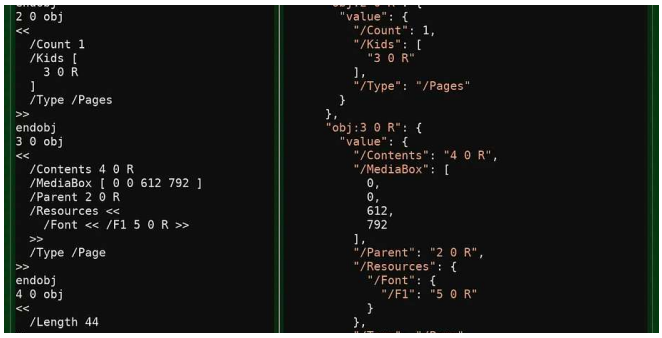
Michael Klink
Jan 24

> In PDF, the origin of the coordinate system is at the bottom-left corner of the page,

It's usually so but not always.

1 reply Reply

More from Jay Berkenbilt



Jay Berkenbilt

Examining a PDF File with qpdf

Introduction

In AWS in Plain English by Jay Berkenbilt

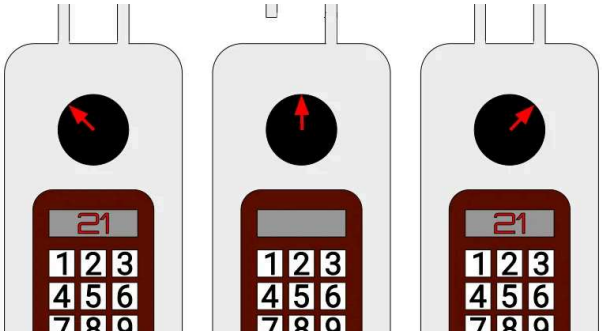
Rotating Secrets with AWS RDS Prox

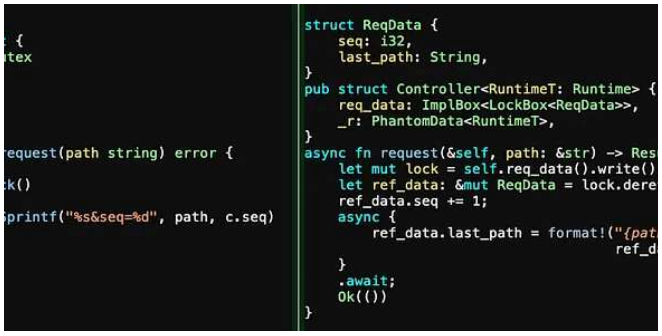
0 s remaining

Nov 7, 2022 94 1

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

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

0 s remaining

Lambert W.

Web Scraping with Claude AI: Python Tutorial

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

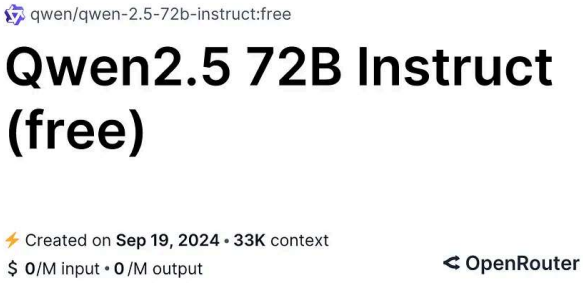
Nov 19 7

RaftLabs

Automating Invoice Data Extraction: OCR vs LLMs Explained

How AI solved our toughest invoice challenges

Jul 8 27 2



Thanyapisit Buaprakhong

Extracting Invoice Data with Qwen2.5-VL and OpenRouter: An...

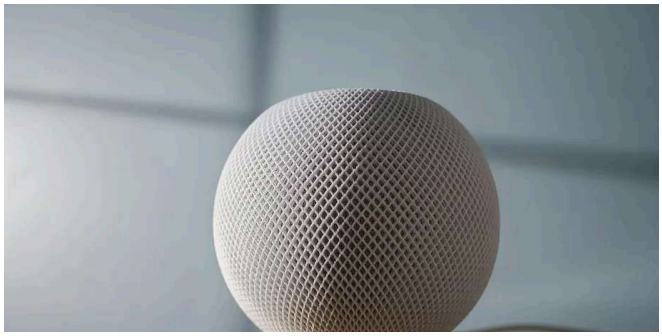
A Step-by-Step Guide to Invoice OCR in Python Using Qwen2.5-VL and OpenRouter

Sep 14

Michael Orozco-Fletcher

How To Use OCR Bounding Boxes

Aug 15

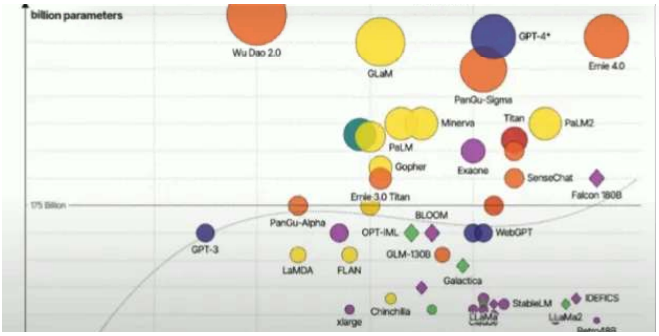


Progressing Llama

How to Make Whisper STT Real-Time Transcription [Part 3]

It's been a while since I last wrote a post. I was pleasantly surprised by how well my earlier...

Aug 9 1



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

See more recommendations

0 s remaining