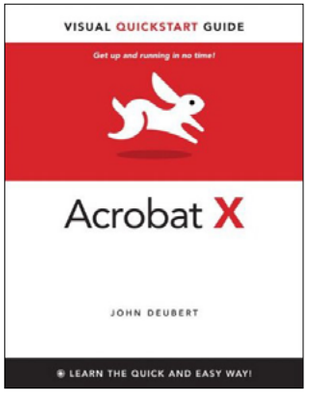
Table of Contents



Journal feedback: suggestions for articles, questions, etc.

Fitting Text to a Specified Width

At right is a situation that comes up more often that you'd think: I need to print a multi-line gob of text within a fixed area; I would like each line of text to extend across this region, as below.

There are two ways of doing this:

Sample Files

As usual, the sample files for this month's article are available on the Acumen Training Resources page. Look for the file *FitText.zip*.

- I can kern the text, offsetting each character in each line by a calculated amount so the line fits across the required space, as at right.
- I can calculate a horizontal scale value, condensing each line's text so it fits across the required space, as below right.

Although I can do this as a one-off for a specific PostScript file, the situation comes up often enough that I long ago automated the task, writing two PostScript procedures that print the text automatically:

- *ShowKernedToWidth* prints the text kerned to the proper total width.
- *ShowScaledToWidth* prints the text horizontally scaled to the proper width.

This month, I share the Inner Mysteries of these procedures.

Kern-to-Fit

Let's start with the kern-to-fit algorithm. We want to fit a piece of text, with some intrinsic width, into a particular space. With the kern-to-fit algorithm, we'll calculate the difference between the space's width and the string's width and distribute the difference between all the string's characters. The characters will print closer together or farther apart than they would normally be, just enough so the string exactly spans the available space.

At right, for example, the string "PostScript" exceeds the width of the rectangle by the indicated amount. We shall need to remove enough space between each pair of characters within the string so it squeezes successfully into the box.

ShowKernedToWidth

We are going to define a procedure named *ShowKernedToWidth* that takes the following arguments:

```
(str) width ShowKernedToWidth
```

These arguments are the string we want to print and the width to which we want to force the string. The procedure prints the string at the current point, kerning the text so it has a total length as specified.

The procedure will ultimately call the PostScript *ashow* operator:

```
ax ay (str) ashow
```

This operator takes an *x* and *y* offset and a string; it prints each character in the string, offsetting the next character by the specified amounts in *x* and *y*. Our *y* offset will always be zero, since our text prints horizontally.

Our *ShowKernedToWidth* procedure will do the following:

1. Determine the length of the string (using the *stringwidth* operator).
2. Subtract this actual string width from the desired width. The difference is the amount by which we

need to increase or decrease the length of the string.

2. Divide this difference by the number of characters in the string minus one (which is the number of inter-character spaces in the string). This is the *kern amount,* the distance we need to offset each character, the a_x argument for *ashow.*

3. Call *ashow.*

Let's look at the procedure definition.

The Definition

Here's a snip of PostScript that defines the *ShowKernedToWidth* procedure and then uses it to print a pair of strings in a 175-point-wide rectangle.

```
/ShowKernedToWidth                % (str) width  ⇒ ---
{    1 index stringwidth pop      % ⇒(str) wid strwid     Calculate the width of the str
     sub                          % ⇒ (str) kern          Subtract from the total width
     1 index length 1 sub         % ⇒ (str) kern count    Count the number of chars - 1
     dup 0 eq                     % ⇒ (str) kern count bool  Is number of spaces zero?
     { pop pop show }             %                        Yes: print the string as-is
     { div 0 3 -1 roll ashow }    %                        No: assemble the ashow call
     ifelse
} bind def

/Helvetica 50 selectfont              % Now let's use the procedure

.5 1 .8 setrgbcolor                   % Draw a rectangle
[ 100 600 175 100 ] dup rectfill
1 .5 .5 setrgbcolor
rectstroke
```

```
0 setgray                                   % Print "Acumen" and "Training"
100 655 moveto                              % kerned to the rectangle's width
(Acumen) 175 ShowKernedToWidth
100 615 moveto
(Training) 175 ShowKernedToWidth

showpage
```

Let's step through the procedure definition in detail.

Step-by-Step

```
/ShowKernedToWidth                  % (str) width  ⇒  ---
{    1 index stringwidth pop        % ⇒ (str) wid strwid    Calculate the width of str
```

The procedure starts by copying the string to the top of the stack and obtaining its printed width with the *stringwidth* operator. Remember that *stringwidth* returns both an *x* and a *y* width; we have no use for the latter, so we discard it with a *pop*.

```
sub                                 %  ⇒  (str) whitespace
```

We then subtract the width of the string from the desired width; this difference is the amount of whitespace that will be left at the end of the line if we were to print the string at its "natural" width. We want to distribute this whitespace between the characters in the string.

```
1 index length 1 sub                %  ⇒  (str) kern count
```

Copying the string to the top of the stack, we then find its length (that is, the number of characters it contains) and subtract 1 from that number; this is the number of inter-character positions in the string.

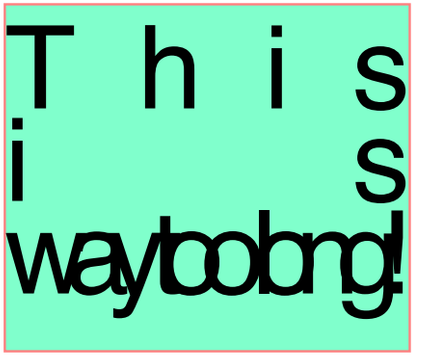
```
dup 0 eq                            %  ⇒  (str) kern count bool
{ pop pop show }
```

We need to check whether the number of inter-character positions is zero; if so, we'll just discard our calculations and print the string as-is.

```
{ div 0 3 -1 roll ashow }
ifelse
```

If the number of inter-character positions isn't zero, we'll do some rearranging of the stack and then call the *ashow* operator.

And that's the end of the procedure; relatively simple, eh?



Note that the kerning technique works regardless of whether we need to tighten or loosen the text spacing, though it works least well at the extremes, as illustrated at right. You get the best results with this technique when the lines of text need only minor adjusting.

The advantage of this technique is that it doesn't distort the character shapes; it affects only the character spacing.

Scale to Fit

The alternative method of fitting text to a specified area is to scale the text—horizontally—so that it fits as needed. Using this technique, the text sample above finishes up looking as at right.

Well, no, it *isn't* pretty. But it is more readable than the kerned version and this is the technique's benefit: characters distort, but they remain distinct.

ShowScaledToWidth

Our *ShowScaledToWidth* procedure will take the same arguments as our earlier procedure:

```
(str) width ShowScaledToWidth
```

As before, we pass the string we want to print and the width to which we want to fit that string. We are going to temporarily change the font to a horizontally scaled version of the current font, using the *makefont* operator.

You remember *makefont,* of course:

```
fontdict  [ sx  0  0  sy  0  0 ] makefont  =>  fontdict2
```

In the above line, *sx* and *sy* are the *x* and *y* scale we want for the new font. Note that *makefont* needs to be followed by a *setfont* before we can use the newly-scaled font.

ShowScaledToWidth will do the following:

1. Determine the length of the string (using the *stringwidth* operator); this is the long arrow in the diagram at right.
2. Divide this value into the space available for the string (the short arrow at right). The resulting value is the horizontal scale we need to make the string fit the proper space.
2. Assemble a six-element array—a transformation matrix—that looks like the following:

   ```
   [ hscale 0 0 1 0 0 ]
   ```

3. Use this array in a call to *makefont*.

Since the procedure changes the current font, it will need to explicitly reset the font to what it was before the procedure was called. (This is easily done with a *gsave* and *grestore.)*

The Definition

Our *ShowScaledToWidth* procedure is defined as follows:

```
/ShowScaledToWidth     % (str) width => ---
{    gsave                          % Save the graphics state
     1 index stringwidth pop        % ⇒(str) wid strwid     Calculate the width of str
     div                            % ⇒(str) hscale         Calc. the needed horiz. scale
     [ exch 0 0 1 0 0 ]             % ⇒(str) [...]          Assemble the makefont array
     currentfont exch makefont setfont   % ⇒(str)          Set the font to our scaled font
     show                           % ⇒---                  Print the string
     currentpoint                   % ⇒x y                  Push current point on stack
     grestore                       % ⇒x y                  Restore original font
     moveto                         % ⇒---                  Pos. current point to end of str
} bind def
```

Let's look at it in detail.

Step by step

`gsave`

The procedure makes an initial call to *gsave*. This allows us to restore the font that is current at the time the procedure is called.

```
1 index stringwidth pop     % ⇒(str) wid strwid
```

As before, we use the *stringwidth* operator to determine the printed width of the string, throwing away the unneeded *y* return value.

```
div                         % ⇒(str) hscale
```

We divide the string's actual width into the desired width, yielding the horizontal scale value we must apply to the current font. This will be the first element of the array we hand to *makefont*.

```
[ exch 0 0 1 0 0 ]          % ⇒(str) [...]
```

This line constructs the *makefont* array. This takes a bit of explanation.

Remember that the open-square-bracket character is a PostScript operator that leaves a *mark* object on the operand stack. The above line starts with the horizontal scale on the stack; we push a mark on the stack—starting an array construction—and then execute an *exch,* dropping the mark to the bottom of the stack and pushing the scale value on top of it. We then pile five constant values (0's and 1's) on the stack and close the array with the close bracket. The final array has the value `[ hscale 0 0 1 0 0 ]`.

```
currentfont exch makefont setfont   % ⇒(str)
```

We push the currentfont on the stack, reverse the font dictionary and array, hand the pair to *makefont* and give the resulting transformed font to *setfont*. Our current font is now a horizontally scaled version of whatever font was current when we executed our procedure.


```
show
```

Now we can print our string, which the earlier code left sitting on top of the stack.

```
currentpoint
grestore
moveto
```

Finally, we clean up after ourselves.

I wanted *ShowScaledToWidth* to move the current point to the end of the printed string, just as *show* does. To this end, we need to push the current point's position onto the operand stack with a call to the *currentpoint* operator. We can then do our *grestore,* which sets the current font back to its original value (and, as a side effect, returns the current point to the beginning of the string), followed by a *moveto,* that restores the current point to the *x* and *y* coordinates sitting on the stack, that is, the end of the string.

That's all there is to it. Call the procedure, passing a string and a width

```
(way too long) 175 ShowScaledToWidth
```

and the text will be printed, scaled horizontally as needed to fill the specified length.

Peachpit Press: Filling Out Paper Forms with the Typewriter Tool

This month's Acrobat article is on-line at Peachpit.com.

You may not be aware that Peachpit Press has a large collection of very informative articles on their web site. (I wasn't until they asked me to write a couple articles for it.) It's a rich source of information on things technical for creative professionals. There are articles, blogs, and (for-sale) eBooks on a wide variety of topics, from digital photography to electronic design to web development.

There will also be some articles on Acrobat X use; that would be me.

This month's article on the site is:

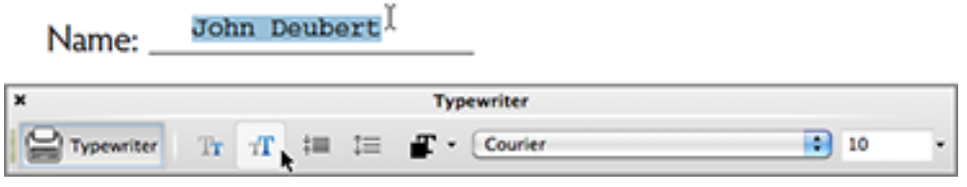
Filling Out Paper Forms With the Acrobat Typewriter Tool

One Time Only

This is the only time that the *Journal* will send you to Peachpit.com.

The next *Journal* will have it's own, independent article, as usual.

The Typewriter tool is one of my favorite tools (of quite long standing) in the Acrobat gizmo box. You can use this to fill out those annoying scanned paper forms that the doctors, lawyers, and other professional folks in your life email to you and ask you to fax it back.



"Fax it back?" How quaint!

Better to fill it out with the Typewriter tool!

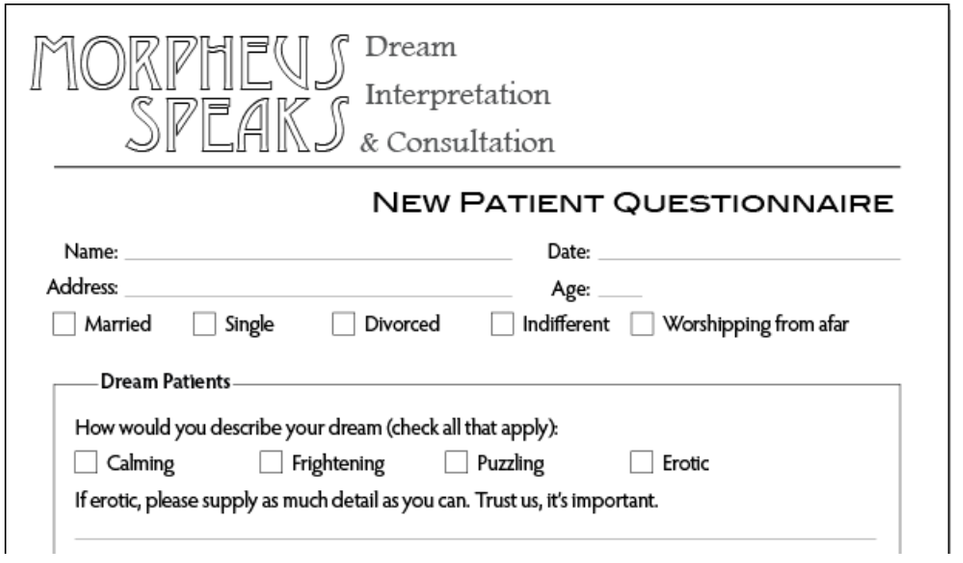
Below is an excerpt from the article; you can read the whole thing by following this link. While you're there, be sure to wander around among the rest of the Peachpit articles. There's a *lot* of interesting stuff just sitting there for the reading.

In fact, by the time you read this, there's probably a second Acrobat article posted.

Filling Out Paper Forms with the Typewriter Tool (Excerpt)

I'll bet this has happened to you.

So, last night I had a dream involving penguins, escalators, and a dentist. Wondering what it all meant, I searched the Web and found a clinic near my home that specializes in dream interpretation. After a brief telephone conversation, they emailed me a New Patient Questionnaire (Figure 1), with instructions to "print it, fill it out, and fax it back." I emphasize this was not an interactive PDF form; it was a paper form they had scanned (or maybe laid out in Microsoft Word) and saved as PDF.



MORPHEUS SPEAKS Dream Interpretation & Consultation

NEW PATIENT QUESTIONNAIRE

Name: _____ Date: _____

Address: _____ Age: ____

☐ Married ☐ Single ☐ Divorced ☐ Indifferent ☐ Worshipping from afar

Dream Patients

How would you describe your dream (check all that apply):

☐ Calming ☐ Frightening ☐ Puzzling ☐ Erotic

If erotic, please supply as much detail as you can. Trust us, it's important.

Figure 1. A paper form sent to me as a PDF file that I need to fill in and "fax" back.

Now, I got rid of my fax machine about the time I sold off my 300 baud modem. I don't fax. I also don't fill things out by hand if I can help it; my handwriting has baffled the finest government cryptographers, so what chance does a clinic desk clerk have?

In the really old days (back when I was still using that 300 baud modem), I would have taken the paper form, rolled it into my trusty typewriter (remember those?), and then typed my name and address and put little typewritten X's in the appropriate checkboxes.

So what do I do in this modern, improved era, when owning a typewriter is at best a mild eccentricity?

Well, you use the Acrobat X Typewriter tool, is what you do.

The Typewriter Tool

The Typewriter tool is, in its modest way, one of the handiest gizmos in the Acrobat X toolbox. With it, you can place pieces of text on top of any PDF page for any purpose. In my case, I wanted to fill in the PDF-format paper form sent to me by the dream clinic.

The Typewriter tool lives in the Content panel of the Tools pane with the title "Add or Edit Text Box" (Figure 2). When you select this tool, Acrobat displays the Typewriter palette, which rides on top of your other Acrobat windows (Figure 3).

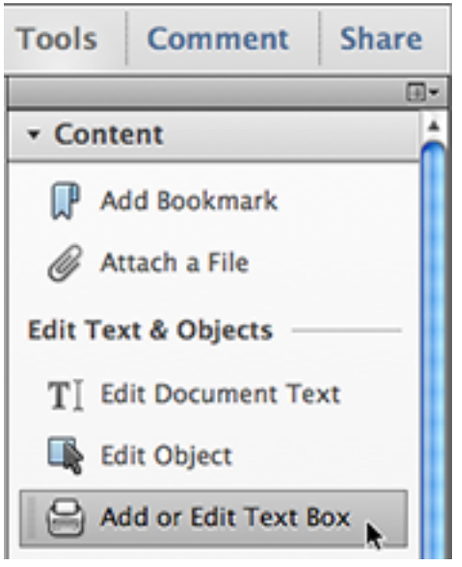


Figure 2. The "Add or Edit Text Box" tool is in the Content panel of the Tools Pane.

End of Excerpt

Thus endeth the excerpt. Again, go here to read the whole thing.

Figure 3. The Typewriter palette contains the tools we'll use in this article. Note that it floats above the other Acrobat windows.

PDF Nuggets

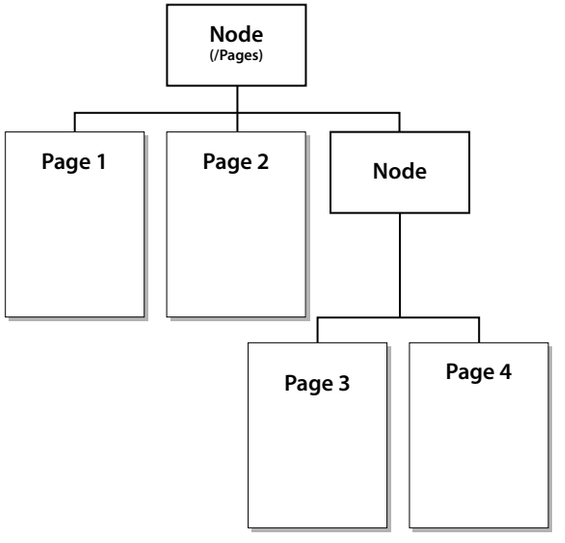
PDF Pages

Among the many ways in which PDF is different from PostScript is that PDF is not *sequential;* that is, the contents of a PDF file do not reside in the file in page order.

In PostScript, all of page 1's stuff comes first, followed by page 2's code, and so forth for the entire document. The benefit of this organization is that a PostScript interpreter can consume the file incrementally. It can read a bit of PostScript code, process it, read some more, repeating until it has finished the entire file. In particular, it doesn't need to store the entire PostScript stream in order to render the entire document.

On the other hand, it makes it more tedious to read the PostScript file in any order other than page order. Rearranging or extracting pages is notoriously difficult.

PDF, on the other hand, stores its pages in a tree structure, as at right. A page's drawing commands, fonts, and other resources may be scattered throughout the PDF file. The PDF file provides a directory to where each page's entry point is located (as well as where to find each resource required by that page).



This has the reverse set of benefits and problems. Accessing arbitrary locations within the PDF file (as when you click on a link in Acrobat) is very easy. On the other hand, if a printer wants to consume a PDF file directly, it must collect and store the entire file (on an internal hard disk, perhaps) before it can begin printing any part of it.

Having enough storage of some sort or other is one of the design requirements for printers that want to print PDF files directly. Happily, that is not too hard to achieve these days, since RAM and hard disks are both relatively cheap.

Schedule of Classes, April 2011– June 2011

At right are the dates of Acumen Training's upcoming classes. Clicking on a class name will take you to the description of that class on the Acumen Training website.

O.C. and On-Site

These classes are taught in Orange County, California and on-site at corporate sites world-wide.

Please see the Acumen Training web site for more information, including an up-to-date schedule.

Class Fee

Classes cost $2,000 per student, with the following exceptions:

- XPS class $1,500
- *Troubleshooting PostScript* $1,500
- *Support Engineers' PDF* $1,000

PDF Classes

PDF 1: File Content and Structure		May 2–5	Jun 27–30
PDF 2: Advanced File Content			
Support Engineers' PDF		May 19–20	

PostScript Classes

PostScript Foundations	Apr 4–8	May 23–27	
Advanced PostScript			
Variable Data PostScript		May 30–Jun 3	
Troubleshooting PostScript		May 16-18	

XPS Classes

XPS File Content and Structure	Apr 18–20		

There is a 10% discount for signing up three or more students.

Note that if you have four or more students that need to take a class, it will almost certainly be cheaper to arrange an on-site class.

Contacting John Deubert at Acumen Training

For more information

For class descriptions, on-site arrangements or any other information about Acumen's classes:

Web site: www.acumentraining.com **email:** john@acumentraining.com

telephone: 949-248-1241

mail: 24996 Danamaple, Dana Point, CA 92629

Registering for Classes

To register for an Acumen Training class, contact John any of the following ways:

Register On-line: www.acumentraining.com/register.html

email: john@acumentraining.com

telephone: 949-248-1241

mail: 24996 Danamaple, Dana Point, CA 92629

On-Site Classes

Information regarding classes on corporate sites is available at www.acumentraining.com/Onsite.html. These courses are taught throughout the world; for additional information on classes outside the United States, go to www.acumentraining.com/OnsitesWorldWide.html.

Back issues

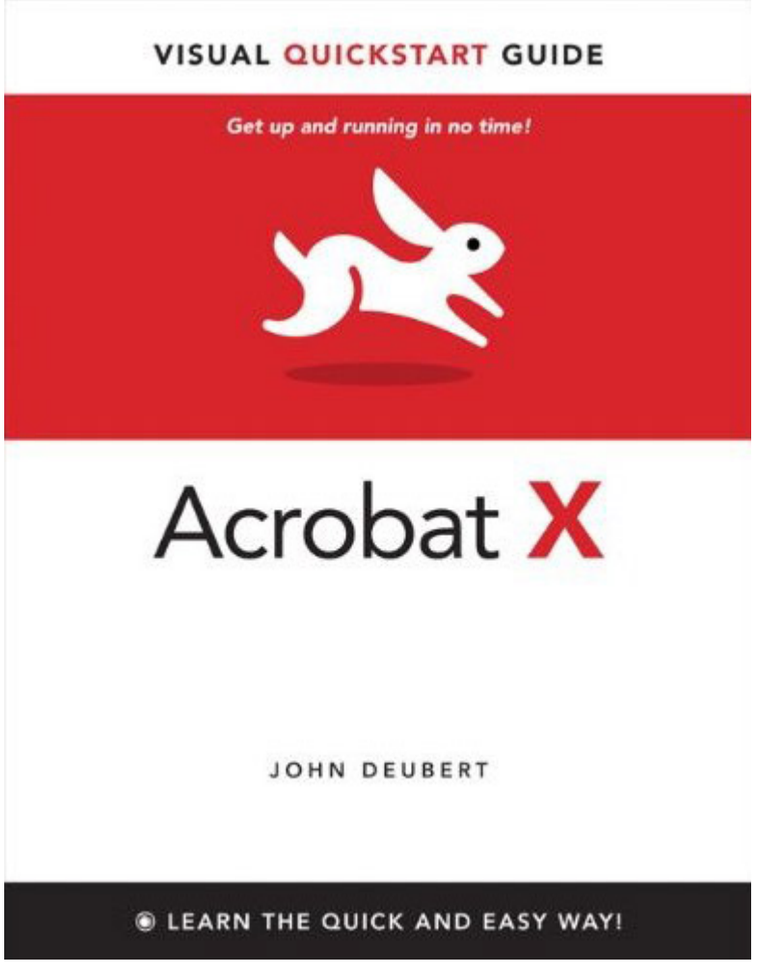
All issues of the *Acumen Journal* are available at the Acumen Training website: www.acumenjournal.com/AcumenJournal.html

What's New at Acumen Training?

Acrobat X Visual Quickstart Guide

I'm sorry that the *Acumen Journal* is so long in coming. It's been a busy six months. However, all the other projects are done and, in particular, the Acrobat X Visual Quickstart Guide is now on your local (and electronic) bookshelves.

Buy one for each of your kids!



MORPHEUS SPEAKS Dream Interpretation & Consultation

NEW PATIENT QUESTIONNAIRE

Name: _____ Date: _____

Address: _____ Age: ____

☐ Married ☐ Single ☐ Divorced ☐ Indifferent ☐ Worshipping from afar

Dream Patients

How would you describe your dream (check all that apply):

☐ Calming ☐ Frightening ☐ Puzzling ☐ Erotic

If erotic, please supply as much detail as you can. Trust us, it's important.

_____