# Extracting Text and Content From Acrobat .PDF files

**Don Lancaster**
**Synergetics, Box 809, Thatcher, AZ 85552**

**http://www.tinaja.com**
**don@tinaja.com**
**(928) 428-4073**

There are times and places when you might like to programmatically reach into an **Adobe Acrobat** .PDF file and extract text, URL links, or other content. Perhaps to do the **word frequency analysis** we looked at in our previous **GuruGram #45**, for additional spell checking, to verify "grade level" of content, to create indexes, provide some disability voice access, do limited editing, or handle any of a number of other custom actions.

In general, there is no single "place" in a .PDF file where you can reach in and grab fully formatted text. The text can appear in any order and often might end up as a mix of normal text and internal figure alphanumerics. Several routes to .PDF text extraction include…

**ACROBAT PRODUCTS** - Full **Acrobat**, the **Acrobat Reader**, and the Adobe **eBook Reader** are standard .PDF viewing options.

**CUT & PASTE** - Text can be copied to a clipboard and put into a word processor or other ap. Text is in expected order.

**ACROBAT PLUGINS** - These **C-language "attachments"** go in an Acrobat folder and for **extended capabilities**.

**JAVASCRIPT OBJECTS** - Modified **JavaScript code** can easily be attached to a .PDF file and run on specified actions, such as a page view or a mouse positioning.

**OPEN SOURCE** - Programs such as **GhostScript** or **xpdf** can give you alternate and platform independent viewing.

**RAW POSTSCRIPT** - The **PostScript** language can read any disk file in any format and offers extreme flexibility for any unusual or custom .PDF text extraction tasks.

I've created some preliminary **PostScript-as-language** .PDF extraction utilities as **VIEWPDF1.PSL**. This is an ordinary and short (about 30K) ASCII textfile that you modify with any editor or word processor and then send to **Acrobat Distiller** . By using **Distiller as a General Purpose PostScript Interpreter**. The code reads a .PDF file of your choice, returns specific object info, full text content, individual words, and sorted word frequency to the .LOG file produced by **Distiller**.

**VIEWPDF1.PSL** is easily adapted to your own needs. At present, this early code **requires** an **uncompressed** .PDF file having a single piece un-linearized **xref** table and all objects at **revision zero**.

You can uncompress any .PDF file by installing this **Windows Acrobat plug in** and following **these details**. Note that installing this plug-in adds an **Uncompressed .PDF Files** option to your full **Acrobat** "Save As" menu. Note also that revision zero can usually be forced by a "Save As" without web linearization or optimization.

The routines are also presently limited to about 1296 or fewer pages as **/Pages** is only nested four deep. Certain word frequency specific utilities also will assume a **WinAnsiEncoding** font character sequence. There are also a few other "second tier" details not yet addressed.

## Acrobat Structure

Your starting point in exploring the .PDF format is the **PDF Reference Manual**.

When uncompressed, a .PDF file is pretty much an ordinary textfile having lines of mostly ordinary printing characters. The bulk of the .PDF file consists of container like "buckets" that are called **objects**.

Objects might hold dictionaries, strings, streams, arrays, constants, variables, extended **PostScript** commands, and true-false Booleans. Or, most importantly, contain **indirect** link references to other objects in the file..

An object always begins with a **27 0 obj** or similar starting line. Where the first digit is the **object number** and the second is the **revision number**. An object always ends with an **endobj** line. Not all objects are necessarily in use at any given time. Thus, you should…

> **Always work with properly linked objects that are arranged in their specified hierarchy!**

There are many types or **classes** of objects. Ferinstance, an **/Annot** object might hold a web link to a URL. A **/Font** object might hold font info, such as the base font used, individual character widths for substitution, the encoding used, and other descriptors.

The objects of immediate top-down interest to us include…

**/Catalog** — The main directory for the entire document.

**/Pages** — A nested tier of objects listing pages in order.

**/Page** — All info required to image one document page.

**/Contents** — Specific marking instructions for one page.

There will be one **trailer** object at the end of any .PDF file. One of its entries will be **/Root**, which will contain the object number of the top **/Catalog**, in a **59 0 R** format of **object number - revision number - reference letter "R"**.

Immediately before the trailer should be an **xref** cross reference. This is your basic tool to access any object. In its simplest form, the cross reference will begin with an **xref** on one line followed by a revision section and a count on the next.

To find any object, you count the object's number of lines down into the xref table, pick off the first ten digits, lop off any leading zeros, and set this as your **file offset** to reach the start of your desired object. In **PostScript**, you can directly use this value sent to **setfileposition** operator to exactly locate yourself.

You can analyze most any .PDF file by reading the **/Root** object to find your **/Catalog** , using **/Catalog** to find **/Pages** and then **/Pages** to find your actual **/Page** objects. A **/Contents** entry in the individual **/Page** objects will then give you access to the words used and other markings and images involved.

There can be some complications. Ferinstance, **/Pages** objects will have **/Kids** arrays that can nest to any depth. Any individual entry in the **/Kids** array might lead to either a **/Page** object or yet another nested **/Pages** object.

Limiting any **/Pages** object to **six or** fewer **/Kids** makes for fast random access through any very long doc. The **xref** access can also get rather messy if revisions are present or if web linearizing is done.

## Some Code Details

**VIEWPDF1.PSL** has fairly extensive internal documentation and comments you can read while editing. There is also a moderate (but not yet bulletproof) amount of forced error trapping. While my **Gonzo Utilities** are highly recommended to work with this routine, they are not absolutely required at present.

Here is a proc-by-proc tour of the **VIEWPDF1.PSL** code…

**The Gonzo Utilities** — These are my collection of superb text justification and figure drawing utilities found on my **PostScript** web page. I have excerpted my **/mergestr** string merger, the disgustingly elegant **/makestring** array-to-string converter, and a grunt **/popbubblesort** that sorts on numeric popularity.

**File Reading** — Your .PDF file to be analyzed is entered at the very top of your document. If you enter several, only the **last uncommented** filename will get used. Several gotchas: Your .PDF file must be uncompressed and have a single rev 0 **xobj** file. **All PostScript filename strings must use a double reverse slash any time a single reverse slash is needed**. The full Windows pathname is required. The complete pathname gets assembled by merging the folder paths with the actual short filename.

**/getobjlist** — This extracts the .PDF **xref** object list into an **/xrefarray** array. The entire .PDF document is searched till **xref** is found using **anchorsearch**. When it is found, the next line is read to find the number of objects. Then, an **xrefarray** is started. As many additional lines as objects are read. The **PostScript /token** proc is used to extract the offset, revision, and used info. And placing it into an **[ offset revision used ]** array element in **xrefarray**. The net result is a saved directory of object info in a **PostScript** useful format.

**/getobj** — Finds a .PDF object given its number and a 0 revision. Each line of the object is returned as a string in an array. The object start offset is found using the **xrefarray** and the file is repositioned using **setfileposition**. An array is started, and each read line of the object is placed into the array as a string object. The object is thus returned on the stack as an array of strings. Certain rare stream elements may be unreadable using this proc, but they do not seem to occur in the objects currently of interest. One big gotcha: **string dereferencing** seems essential to prevent wrong characters from showing up due to reuse or our main **workstring**. Details on this process appear in **STRCONV.PDF** you can find in our **GuruGram** library.

**/fixbrokenarrays** — There is one nasty minor detail when extracting object lines: Arrays may start on one line and end on another. This utility attempts to find any array that **starts but does not end** on a given line. It then merges that string with the next one. The process continues till an array is complete as a single line entry. A **/tryrepair** subroutine does the actual string merging and reassembly.

**/getcatobj** — Finds Reads **/Root** to determine the **/Catalog** object, then retrieves it. Starting at the beginning of the file, an **anchorsearch** is done to find the **/Root** entry. Returns **/Catalog** object using **getobj** to stack and reports success or error to log file. At present, assumes all are objects are revision 0.

**/getpagesobj** — Reads the **/Catalog** object to extract the top **/Pages** object. An **anchorsearch** of the catalog is done looking for the top **/Pages** array. An **R2obj** service routine is used to convert catalog strings such as **([ 60 0 R 1 0 R 25 0 R])** into PostScript arrays such as **[ 60 1 25 ]**. Again, rev 0 is assumed.

**/getcontentarray** — Things get tricky from here. Any **/Pages** object will normally only have **six** or fewer links in its **/Kids** entry. If there are **less** than six pages in the document, all **/Kids** entries will be new **/Page** objects (without the "s"). But if there are more than six pages in the document, a **/Kids** entry might end up either

a **/Page** object or a nested object of additional **/Pages** links. The way you tell a **/Pages** object from a **/Page** object is that **/Pages** has a **/Kids** entry, while **/Page** has a **/Contents** entry. Our object here is to create an array of all used **/Contents** objects in sequential order. Routines **/firstkids**, **/secondkids**, **/thirdkids**, and **/fourthkids** let us search in depth for documents of 1296 or fewer pages.

**/addtocontentarray** — Any single page can have one or more **/Content** objects. Especially if figures or images are involved. For convenience, two different formats of content lists are provided. **/pagearray** is a simple list of content objects. Which is all you would need for word frequency apps. **/numberedpagearray** is a fancier array-of-arrays in which each entry represents one page. Each entry is an array that can hold one or more content objects. This might be more useful for index generation where you need to remember both the word and the page it is on. Two types of **/Content** objects are distinguished. Those with a **stream** in them are an actual content object, while those with an **array** are a pointer to additional content objects.

**/reportPDFstructure** — This is the high level code that that finds key objects and reports on them, ending with **/pagearray** and **/numberedpagearray** page content lists. These content arrays can then be used to develop applications of your choosing. Such as…

## An Example: Word Frequency Analysis

As we've seen in **WORDFREQ.PDF** that we looked at in our previous **GuruGrams**, word frequency analysis is useful to improve author presentation, create indexes, verify grade level, and a number of other tasks. Some analysis routines provided with **VIEWPDF1.PSL** include…

**/expandline** — Extracts string objects from page content.
**/forcelowercase** — Replaces uppercase characters.
**/getwordsfromstring** — Isolates single words from string.
**/removetrailingpunct** — Removes end periods, commas, etc.…
**/removeleadingpunct** — Removes opening quotes.

**/procword** — High level word counting and entry code.
**/gotone** — Counts and places words in **/worddict**.
**/reportworddict** — High level output sort and report.
**/formattedprint** — Sets details of output presentation.
**/analyzewordfrequency** — Reports .PDF file word frequency.

Note that **/removetrailingpunct** and **/removeleadingpunct** currently assume a **WINansiEncoding** vector. Processing time is quite fast for shorter .PDF files, ranging up to two minutes for a 200 page document. This can be sped up by going to a **better sorting routine** and eliminating intermediate reporting.

There are all sorts of exciting additional possibilities for **PostScript** reading and reporting of .PDF file content. Ferinstance, indexing can be done by using the **/numberedpagearray** data to add a third column to your word reporting that shows which page each word gets used on.

## For More Help

Enhancements and improvements on this fast, convenient, and super flexible .PDF content reader can be made available to you on a **Custom Consulting** basis. Additional GuruGrams are found **here**, PostScript topics **here**, and Acrobat info can be found **here**.

Further **GuruGrams** await your ongoing support as a **Synergetics Partner**.