PageCatcher Intro

Thornton House
Thornton Road
Wimbledon
London SW19 4NG, UK

PageCatcher Intro

Contents

PageCatcher Intro

Last updated 27 August 2001. PageCatcher has been made largely obsolete by Report Markup Language, but is still available if you want to use it.

This document provides a basic introduction to using PageCatcher. It includes explanations of what PageCatcher does, how it can be used, what the current limitations are, how to run the demo applications, how to run PageCatcher as a command line program, how to use PageCatcher with the RML2PDF application, and how to use the PageCatcher programming interface within other programs.

What is PageCatcher for?

PageCatcher is an add-on utility for ReportLab's suite of enterprise reporting tools, as well as the most versatile tool for batch manipulation of PDF files. The suite runs on all common computing platforms.

The free ReportLab core API lets you create PDF files directly using the Python scripting language; our commercial RML2PDF Report Markup Language product lets you specify printed documents in easy-to-understand XML and converts these to PDF. PageCatcher allows these packages to reuse complex designs from existing PDF files in dynamically created PDF documents.

Many documents require elements such as fixed form layouts, headers, footers, corporate logos, or other art work which are most cost effectively created by artists or design specialists using visual tools. With Adobe Acrobat, they can use any tools they wish and convert it to PDF. These visual elements can then be seamlessly integrated into PDF reports using PageCatcher.

In addition, many applications require batch or server-side modification of existing PDF documents - adding simple annotations, combining documents or printing 2-up or 4-up. These can all be scripted trivially with PageCatcher. There are many single-purpose programs to append, rearrange and extract text from PDFs; PageCatcher's simple API and a scripting interface provides the most versatile solution on the market.

Running the demos

The demos are for Windows only and are packaged as a zip file. This creates a subdirectory called 'pageCatcher' under the location where you unzip it; so you can safely unzip into C:

This distribution consists of:

- 00README.txt - starting point
- PageCatchIntro.html - this document(in html format)
- pageCatcher.exe - executable program
- sample1.pdf - a U.S. government tax form
- sample2.pdf - first ten pages of Psion's 1997 annual report
- sample3.pdf - a custom page backdrop
- five sample scripts (example*.py) to manipulate the examples
- runall.bat - batch file to run all demos at once

When run as a command line program PageCatcher has many command line options; the first argument is a command. The general command line usage for PageCatcher is

```
pageCatcher.exe COMMAND ARGUMENT1 ARGUMENT2 ...
```

The COMMAND indicates what action pageCatcher should perform. The most general command is the 'exec' command.

The 'exec' command runs a Python script that makes use of PageCatcher's functionality. In the demo distribution we provide five scripts to demonstrate the versatility of the API. Try these commands from a command [MSDOS] prompt:

```
pageCatcher.exe exec example1_fillform.py
pageCatcher.exe exec example2_reverse.py
pageCatcher.exe exec example3_append.py
pageCatcher.exe exec example4_fourpage.py
pageCatcher.exe exec example5_background.py
```

Each results in a PDF file being written which begins with 'out'; look at these as well as the samples to get an idea of the capabilities. You can also use the batch file 'runall.bat' to run all five demos in one go.

How does it work?

There are two logical steps in using PageCatcher. First, pages must be *extracted* into a special data file format using the PageCatcher filter script mode. Second, the extracted pages may be *imported* by ReportLab programs. In many applications, extraction is a one-off design-time step, and the data files produced can then be included in new documents at very high speeds.

The commercial product consists of a compiled Python module (similar to a Java class file) which can be used in 3 ways:

- as a command line application with many useful options
- as a library within Python scripts
- controlled by tags within RML documents

The PageCatcher product can either function as a module in a larger Python installation (which should include the ReportLab core libraries), or as a stand alone executable which contains the ReportLab distribution and all other required software components.

In either mode you can write your own scripts as well as looking at the ones we provided. Please consult the first few chapters of the **Reportlab User Guide**, and to look at the documentation for the **Python** scripting language for additional information on using the ReportLab toolkit and the Python programming language.

PageCatcher also functions as a add on component to the RML2PDF program supported by the *catchForms* RML tag. Please see the RML2PDF userguide for more information on using RML2PDF.

Known Deficiencies and Caveats

PageCatcher does not support PDF pages with stream content arrays compressed using the LZW compression method. (Unfortunately this is used in British tax forms). We are working to add this support.

PageCatcher cannot capture pages that contain "Active PDF Form" annotations (such as checkboxes or fill-in text areas).

You must supply a user password to process encrypted PDF files

```
pageCatcher...  --password MYUSERPASSWORD
```

Since the preprocessor step for PageCatcher parses the entire PDF file, parsing very large files may consume a great amount of computational resources even if only one page is extracted from the file.

Workarounds

If you have a copy of Adobe's Distiller, you can use it to work around the majority of problems. To do this, use Distiller's printer emulation to "print to PDF" and the file created will be digestible by PageCatcher. (One *known* exception: where the PDF file is encrypted and printing is not permitted).

Page Extraction

PageCatcher can extract pages from PDF files into the import data format either using a command line or using a function call from within a python program or script. All extraction options may specify a prefix to use in the form names and also for other internal purposes. **It is important that if a generated document uses several PageCatcher data files that the data files use *different* prefixes.**

Line mode page extraction: In script mode PageCatcher prepares the contents of one or more pages of a PDF file for use in other PDF files.

```
% PageCatcher makeforms pdffile [-s storagefile] [-p prefix]
     [--password password] [--test pdftestfile] [--all] [pagenumber]*
```

This command captures the pages from pdffile and places them in storagefile for later use. If the test option is used then the captured pages are reimported and placed in the test file, overlayed with a centimeter grid.

Note! Pagenumbers start at 0 (zero) (with no necessary relation to the pagenumber shown by a PDF viewer such as Acroread). If the pagenumbers are omitted only the first page of the document will be made into a form.

The "form names" for the forms derived from the pages will be prefix0 for the front page, prefix1 for the following one, prefix2 for the one after that, and so forth.

For example

```
% pageCatcher makeforms picture.pdf -s pic.data -p pict --test pictest.pdf 0 2
```

extracts the first and third page from picture.pdf, archiving them in pic.data for later use, giving them the form names pict0 and pict2, respectively. The test file pictest.pdf will display the captured forms overlayed with a centimeter grid.

If the storagefile is omitted it defaults to "storage.data". If the prefix is omitted it defaults to PF (for "page form"). If the --all option is used then all pages of the document are captured.

Function call mode page extraction: The *storeForms* function extracts a form from within a program or script. The Python programming language signature for *storeForms* is

```
storeForms(frompdffile, storagefile, pagenumbers=None,
      prefix="PageForms", all=None, verbose=0, password=""):
```

The usage of *storeForms* is analogous to the script usage described above, except that there is no option for test output.

- *frompdffile* must provide the name of an existing PDF file to use for extracting the forms.
- *storagefile* must provide a name to use for the storage file in which to store the formatted form data.
- *pagenumbers* if present should be a Python list of integers listing the offsets of the pages to store as forms (with the front page of the document at offset 0 and the next page at offset 1 and so forth).
- *prefix* when used should be a string to use as the form prefix.
- *all* when used and set specifies that all pages of the PDF file should be captured as forms.
- *verbose* if present and set will cause the generation process to print verbose commentary on the extraction process (for debugging).
- *password* will be used if the PDF file has been encrypted. It should provide the User password for the file (which is the empty string if the document is readable without a password).

The return value of *storeForms* is a list of strings listing the names of the forms stored in the storage file.

For example

```
names = storeForms("manual.pdf", "manual.data", prefix="fourpage", all=1, verbose=1)
```

Stores *all* pages from *manual.pdf* in storage file *manual.data* using the prefix *fourpage*, with verbose commentary printed to standard output.

Using Caught Pages

Both the ReportLab RML2PDF product and the ReportLab core Python API can use PageCatcher storage files to place captured graphics in generated PDF files. In addition, PageCatcher provides several built in demo modes listed below.

Catching Forms in RML: If you have production versions of both RML2PDF and PageCatcher you can use a special Report Markup Language tag *catchForms* which imports all forms from a PageCatcher storage file for use in an RML document.

For example: The following RML code fragment draws a caught form *PF0* (stored in storage file *storage.data*) onto a page backdrop.

```
<pageDrawing> <catchForms storageFile="storage.data"/> <doForm name="PF0"/>
</pageDrawing>
```

The *catchForms* tag can occur anywhere where a *doForm* tag can occur.

Catching Forms in Python using the ReportLab core API: You can also use PageCatcher caught pages in documents created using the ReportLab core API for creating PDF programs. The *restoreForms* function imports forms from a storage file into a pdfgen Canvas object.

```
def restoreForms(storagefilename, canv, verbose=0):
```

- *storagefilename* must be the string name of a PageCatcher storage file.
- *canv* must be a reportlab.pdfgen.canvas.Canvas object
- *verbose* if used and set instructs the function to print verbose progress and diagnostic information to standard output (for debugging).

The result of the function is the list of names of the forms extracted from the storage file.

The following example function extracts all pages from a storage file and places them on 4 to a page in a new PDF file.

```
def fourPage(storagefile, testfile, scalefactor = 0.5):
    print "placing forms from", storagefile, "into", testfile, "four to a page"
    from reportlab.pdfgen import canvas
    canv = canvas.Canvas(testfile)
    (width, height) = canv._pagesize
    names = restoreForms(storagefile, canv, verbose=1)
    while names:
        for (xoff, yoff) in [ (0,1), (1,1), (0,0), (1,0) ]:
            thisname = names[0]
            print thisname,
            canv.saveState()
            (x,y) = (xoff*width/2.0, yoff*height/2.0)
            canv.translate(x,y)
            canv.scale(scalefactor, scalefactor)
            canv.doForm(thisname)
            canv.restoreState()
            del names[0]
            if not names: break
        canv.showPage()
        print
    canv.save()
    print "wrote", testfile
```

The *fourPage* function first creates a canvas, extracts the forms for the canvas using *restoreForms*. Then it iterates over the names of the forms placing the first at the upper right part of the page, the second at the upper left, the third at the lower right, and the fourth at the lower left. Then the function continues this process on the next page with the remaining forms until they all the forms have been placed. Finally the function saves the document. For a detailed explanation of the methods of the canvas object please see the ReportLab core API userguide.

Demo Modes

The PageCatcher program also includes a number of built in demonstration modes. These options are provided as an easy way of showing some of the capabilities of PageCatcher without requiring any programming or the use of RML2PDF.

help:

```
% pageCatcher help
```

This mode prints a short explanation of the script options.

note:

```
% pageCatcher note [pdffile]
```

This mode places a text string over the first page of pdffile, storing the result in "annotated.pdf". The font, size and string are read interactively from the console.

4page:

```
% pageCatcher 4page pdffile [--scale scalefactor]
                    [--output pdfoutputfilename] [-s storagefilename]
```

This mode rewrites the pages of pdffile with 4 pages of the input on each page of the output ("save the trees" mode).

exec

```
% pageCatcher exec scriptFileName
```

This mode executes a python script. This mode is provided for the case where PageCatcher is distributed as a stand alone executable for demonstration and evaluation purposes. It allows evaluation customers to try scripting usage without having a Python installation. *Note that not all legal scripts will work with the stand alone evaluation version since the executable does not contain all standard library modules.*

Additional Feature -- Copying and Appending PDF files

There are a number of additional features in PageCatcher beyond the fundamental operation of capturing pages from one document and embedding them in another. These features are available when using "exec scriptname" form or when importing the licensed pageCatcher component into another program. *These features do not require a PageCatcher license to work in production mode.*

```
copyPages(frompdffile, tocanvas, withoutline=1)
```

The function copyPages will copy all pages of a PDF file into a ReportLab document (the document that is being created using the canvas object). The pages will be copied without changes. For example the following script will append any number of pdf files together into a new pdf document.

```
try:
    from rlextra.pageCatcher.pageCatcher import copyPages
except ImportError:
    pass # running inside pageCatcher module?
```

```
from reportlab.pdfgen import canvas

def doappend(topdffile, frompdffilelist):
    canv = canvas.Canvas(topdffile)
    for frompdffile in frompdffilelist:
        print "copying", frompdffile
        copyPages(frompdffile, canv)
    print "\n\nnow writing", topdffile
    canv.save()

if __name__=="__main__":
    # edit this
    doappend("out8_directcopy.pdf", ["sample1.pdf", "sample2.pdf", "sample3.pdf"])
```

If you do not want to include the outline from the copied document in the result unset the parameter withoutline=0.

Note that copyPages may currently be used without purchasing a PageCatcher license. This functionality was included to assist some open source users of our product who were urgently in need of this function. We do not, however, promise to keep this free in future versions.

Feedback

We need and welcome feedback to help make this into a great product! Email info@reportlab.com, or join our group of 200+ existing users by emailing reportlab-users@reportlab.com. Enjoy!