Dokumente nach Maß

Schriftdokumente müssen auch für menschliche Leser immer mehr automatisch und „on-demand“ generiert werden, sei es wegen der reinen Datenmenge oder um maßgeschneiderte Dienstleistungen für die neuen vom Internet verwöhnten Kunden anzubieten. Eine Open Source-Bibliothek hilft dabei. (Dinu Gherman)

Viele Dokumente werden heute ausschließlich elektronisch erzeugt, gepflegt, ausgetauscht und archiviert. Dies gilt auch für solche, die für den menschlichen Leser bestimmt sind, der ohne ein gewisses Maß an Präsentation nicht auskommt. Die schiere Menge an Daten, Erzeugern und Verbrauchern macht es jedoch immer mehr notwendig, Dokumente automatisch zu erzeugen, und zwar immer dann, wenn der Kunde es will, und sei es, weil sie nur einmal genau für diesen individuell angefertigt werden.
Dieser Artikel beschreibt das ReportLab Toolkit, eine Open Source-Bibliothek, die sehr flexibel Dokumente und/oder Teile davon komplett automatisiert und programmatisch erzeugt. Verschiedene Teile der Bibliothek werden vorgestellt und in einem größeren Beispiel benutzt, nämlich der Konvertierung des ASCII-Quelltextes dieses Artikels im Linux-Magazin nach PDF. Dabei sollte klarwerden, dass die Software auch in kleineren Projekten sehr sinnvoll eingesetzt werden kann.

Aus der Vogelperspektive

Das ReportLab-Toolkit ist eine Open Source-Bibliothek (aktuelle Version ist 1.15), mit der sich schnell professionelle Dokumente sowie Teile davon rein programmatisch und damit völlig automatisiert erzeugen lassen. Dabei werden kleine Programme geschrieben, bei deren Ausführung Dokumentdateien mit allen üblichen Inhalten von Schriftdokumenten erzeugt werden, d.h. neben Text auch Tabellen, Bilder und Diagramme. Diese Dokumente können beliebig komplex sein und von externen Datenquellen abhängen. Dank des frei verfügbaren Quellcodes und seiner sehr freizügigen Lizenz (FreeBSD), läuft man nicht Gefahr, von einzelnen Herstellern und/oder deren proprietären Technologie abhängig zu werden.
Dokumente als ganzes werden in PDF erzeugt, wobei die unterste Schicht der Bibliothek eine „Leinwand“ bereitstellt, auf der sich leicht einfache Text- und Graphikelemente platzieren lassen und dann mit den üblichen Werkzeugen gruppiert und transformiert werden können. Eine Ebene höher implementiert »Platypus« einen flexiblen Layout-Kern, der aus Komponenten wie Überschriften, Paragraphen, Tabellen, Bildern, etc. Dokumente macht. Diese Komponenten werden zu einer „Story“ verbunden, die mit Hilfe von Vorlagen für einzelne Rahmen und Seiten sowie für das Dokument insgesamt „ausgerollt“ wird. Neben den 14 Standardschriften aus Postscript werden auch beliebige andere Schriften im Format Type-1 und (eingeschränkt) TrueType ebenso wie asiatische Zeichensätze unterstützt.
Ein wichtiger Aspekt professioneller Dokumente sind die Graphikmöglichkeiten. Das entsprechende Unterpaket erlaubt mit einer einfachen API, Bitmap- und Vektorgraphiken zu importieren und zu exportieren. Auf unterster Ebene gibt es primitive Elemente (Kreise, Rechtecke, etc.), aus denen man seine eigenen komplexeren »Widgets« zusammensetzen kann. Bereits enthalten ist eine Sammlung von solchen einfachen Widgets, aber auch komplexere Business-Charts.
Die ReportLab-Bibliothek ist überwiegend in Python geschrieben mit nur wenigen optionalen Teilen in C (vorwiegend zur effizienteren Berechnung von Umbrüchen). Die Bibliothek selbst ist somit plattformneutral (auf der Mailingliste tummeln sich Benutzer von Linux, BSD, Solaris, AIX, MacOS 9/X und Windows). Bis auf einige Graphikteile kann sie sogar unter Jython benutzt werden, einer Java-Implementierung von Python.
Die gesamte Dokumentation wurde in bester „bootstrap“-Manier mit dem Toolkit selbst erzeugt. Die entsprechenden Programme sind Bestandteil der Distribution und befinden sich in »reportlab/docs«. Zu den anderen Beispiel-Anwendungen gehört »PythonPoint«, ein Programm zum Erstellen von PDF-Präsentationen aus einfachen XML-Dateien (siehe »reportlab/tools/pythonpoint«).

Eine Frage des Formats

Obwohl das ReportLab-Toolkit mit vielen Formaten umgehen kann, ist das zentrale Format für das, was man als „Dokument“ bezeichnet PDF, das »Portable Document Format«, ein von Adobe entwickeltes binäres plattformübergreifendes Dateiformat. Seine Spezifikation (aktuelle Version ist 1.4) ist auch insofern offen, als jeder eigene Software zum Erzeugen oder lesen/darstellen von PDF schreiben kann, ohne Lizenzgebühren entrichten zu müssen.
Aber PDF hat sich nicht nur als Nachfolger von Postscript (auch von Adobe) am Markt durchgesetzt und wurde von Leuten entwickelt, von denen man annehmen muss, dass sie wissen, was sie tun. Es hat ganz einfach eine Reihe von praktischen Eigenschaften. Weil es plattformübergreifend und „kompakt“ ist und es die Software zum Betrachten und Drucken gratis gibt (nicht nur den Acrobat Reader von Adobe), ist es aus der Benutzersicht sehr einfach zu handhaben. Solche Dateien enthalten keine Viren, können auf allen Plattformen betrachtet und ausgedruckt werden, sind leicht zu handhaben, per E-Mail zu verschicken und zu archivieren, und sie bieten beliebige Freiheit beim Layout. Inhaltlich sind PDF-Dokumente nur sehr schwer direkt zu manipulieren und enthalten keine verräterische Information bzgl. Versionierung, da sie nicht mit einem einfachen Editor verändert werden können.
Die weniger praktischen Eigenschaften treten dann in Erscheinung, wenn es darum geht, PDF zu erzeugen und zu „verarbeiten“. In diesem Bereich ist eine ganze Plugin- und Filter-Industrie mit allen Begleiterscheinungen drumherum entstanden. Erst in letzter Zeit gibt es hierzu einige, oft eingeschränkte, Open Source-Ansätze.
Nebenbei sei bemerkt, dass man mit einigen Tricks aus ReportLab-Dokumenten auch QuickTime-Filme erzeugen kann, was aber noch nirgendwo beschrieben wurde.

Die Leinwand als Grundlage

Historisch betrachtet stand am Anfang ein von Andy Robinson (Gründer und CEO von ReportLab) entwickeltes, eigenständiges Modul namens »pdfgen«, das im wesentlichen auch heute als Teil der Gesamtbibliothek weiterbesteht. Es

implementiert ein simples Konzept einer, auch mehrseitigen, „Leinwand“ (Canvas). Auf dieser wird mit einer einfachen Schnittstelle Text und Graphik platziert, wobei zwischendurch immer wieder einmal die „Werkzeuge“ d.h. Farben, Formen, Strichbreiten, Schriftarten und -größen gewechselt werden.
Andere Eigenschaften von PDF werden ebenfalls unterstützt, zum Beipspiel Kompression oder Verschlüsselung (optional), Metainformation zum Dokument selbst (Titel, Verfasser, Stichwörter), oder gewisse Optionen dafür, wie es unmittelbar nach dem Öffnen dargestellt wird oder welche Effekte beim Wechsel zwischen den Seiten verwendet werden.
Listing X zeigt ein minimales Beispiel, das obligate „Hello World“, in dem mit nur fünf Zeilen eine einzelne PDF-Seite erzeugt wird (per Voreinstellung A4, Hochkant), auf der an einer bestimmten x/y-Position die Worte „Hello World“ stehen. Einfacher geht es kaum. Neben den hier verwendeten typographischen Punkten können auch andere Einheiten wie »cm« oder »inch« benutzt werden. Interessant ist, dass (wie im Publishing üblich) der Urprung des Koordinatensystem unten links ist und die y-Achse nach oben zeigt.

Listing X: Hello World

```
# hello.py

from reportlab.pdfgen.canvas import Canvas

c = Canvas('hello.pdf')
c.setFont('Helvetica-Bold', 36)
c.drawString(100, 100, 'Hello World')
c.save()
```

Die Leinwand verfügt über viele weitere Methoden, mit denen sich einfache Grapfikformen direkt dargestellen und die genannten speziellen Eigenschaften verwenden lassen.

Schriften verwenden

ReportLab unterstützt die 14 Standard-Schriften, die in Postscript definiert werden (Varianten von Helvetica, Times, Courier sowie Symbol und ZapfDingbats). Außerdem kann jede weitere Schriftart verwendet werden, sofern eine Beschreibung davon in Postscript (Type-1) als AFM- und PFB-Dateien vorliegt.
TrueType wird dank einer Erweiterung von Marius Gedminas ebenfalls implementiert, mit der TTF-Dateien direkt geladen werden können. Allerdings funktioniert dies noch nicht im Kontext von Graphiken und speziell Bitmap-Formaten. TTF-Dateien sind dafür geringfügig einfacher zu verwenden als Type-1.

Listing X: Schrift-Import (TrueType)

```
 1 # ttfonts.py
 2
 3 from reportlab.pdfgen.canvas import Canvas
 4 from reportlab.pdfbase import pdfmetrics as pdfm
 5 from reportlab.pdfbase.ttfonts import TTFont
 6
 7 c = Canvas('helloRina.pdf')
 8 pdfm.registerFont(TTFont('Rina', 'rina.ttf'))
 9 c.setFont('Rina', 36)
10 c.drawString(100, 700, "Text in Rina")
11 c.save()
```

Dynamische Graphiken erzeugen

Eine der Stärken der ReportLab-Bibliothek ist es, dass Graphiken zur „Laufzeit“, also in dem Moment, wo das Dokument generiert wird, erzeugt werden können, zum Beispiel mit Hilfe von aktuellen Daten aus einer Datenbank oder von einer entfernten Website.
Diese Graphiken möchte man in der Regel auch einzeln erzeugen und abgespeichern, also ohne ein „umgebendes“ Dokument. Dazu definiert das Toolkit mit »ReportLab Graphics« (RLG) eine abstrakte API, die unabhängig von der bisherigen PDF-Leinwand ist. Zu den so unterstützten Formaten gehören JPEG, GIF, PNG, BMP, etc. (im Prinzip alle Pixel-Formate), sowie die Vektorformate PDF, EPS und SVG.
Die Graphik-Primitive umfassen die üblichen als »Shapes« bezeichneten Formen: nämlich Line, Rect, Circle, Ellipse, Path, Polygon, PolyLine, String, etc. Hinzu kommt die logische Form »Group«, mit der man Formgruppen als Einheit behandeln kann. Üblicherweise bedeutet das Transformationen wie Skalierung, Verschiebung, Spiegelung und Verzerrung darauf anzuwenden.
Da das API abstrakt ist, wird der Graphikinhalt programmatisch und völlig plattformneutral aufgebaut. Erst am Ende muss man sich entscheiden, in welchem Format er abgespeichert werden soll. Dazu wählt man einen der möglichen »Renderer« aus, nämlich

- »renderPM« („Pixelmap“),
- »renderPS«,
- »renderPDF« und
- »renderSVG«.

Bei den letzten dreien wird Vektorgraphik erzeugt, die oftmals nicht nur eine fast beliebige Skalierbarkeit im Ausgabeformat bedeutet, sondern auch eine wesentlich reduzierte Dateigröße. Der SVG-Renderer ist der jüngste Zugang in dieser Bibliothek und erzeugt brauchbaren, wenn auch noch wenig optimierten SVG-Code.
Im folgenden kurzen Beispiel wird eine einfache Graphik aufgebaut und in den Formaten EPS und GIF abgespeichert. Man sieht wie einfach es ist, aus dem gleichen Code (und den gleichen Eingangsdaten) konsistente Graphiken sowohl für den Print- wie auch den Online-Bereich herzustellen.

Listing X: Ein simples Graphik-Beispiel

```
 1 # hellorect.py
 2
 3 from reportlab.lib import colors
 4 from reportlab.graphics import shapes,\
 5      renderPS, renderPM
 6
 7 rect = shapes.Rect(0, 0, 100, 50,
 8     strokeColor=colors.blue,
 9     fillColor=colors.aqua)
10
11 text = shapes.String(10, 10, "Hello",
12     fontName="Times-Bold", fontSize=24)
13
14 d = shapes.Drawing(100, 50)
15 d.add(rect)
16 d.add(text)
17
18 renderPS.drawToFile(d, "hellorect.eps")
19 renderPM.drawToFile(d, "hellorect.gif")
```

Ein Modul namens »svglib« zum Lesen und Wiederverwenden von SVG-Dateien ist auf den Webseiten des Autors frei verfügbar. Es konvertiert SVG in gewissen Grenzen mit dem ReportLab-Graphics API (RLG) in eine interne Darstellung, mit der man dann genauso arbeiten kann, als ob man sie selbst erstellt hätte. Das kann zum Beispiel heißen, dass man sie in einem anderen Format abspeichern möchte. Folgendes Listing zeigt exemplarisch, wie man »svglib« verwenden kann, um SVG in PNG zu konvertieren. Der eigentliche Code ist prinzipiell nicht viel mehr als ein Einzeiler.

Listing X: Konverter, SVG nach PNG

```
# svg2png.py

import sys
from os.path import splitext

from reportlab.graphics import renderPM
from svglib import svg2rlg

inPath = sys.argv[1]
drawing = svg2rlg(inPath)
outPath = splitext(inPath)[0] + '.png'
renderPM.drawToFile(drawing, outPath, 'PNG')
```

Statische Graphiken einbinden

Bilder, d.h. „Bitmaps“, in eigene Dokumente einzubinden geht ohne weitere Hilfsmittel, wenn diese im JPEG-Format vorliegen. Für alle anderen Formate ist es notwendig, die frei verfügbare Standardbibliothek PIL, »Python Imaging Library« der Firma PythonWare zu installieren. Danach bleiben keine (Bitmap-)Wünsche mehr offen.

RLG bedient sich für Bitmap-Graphiken nicht nur bei PIL, sondern auch bei »libart«, dessen C-Code in einer Zusatzbibliothek namens »_renderPM« eingegangen ist, die von ReportLab.com heruntergeladen werden kann (für Windows-Benutzer auch als übersetzte DLL).

Eigene Graphikbibliotheken

Aufbauend auf den einfachen Grundformen in RLG wird eine Klasse namens »Widget« definiert, die dazu gedacht ist, eigene wiederverwendbare Graphikbibliotheken für spezifische Anwendungen zu definieren. Ein konkretes Beispiel sind Business-Charts (Linien, Kuchen- und Balkendiagramme), von denen einige Varianten in der Bibliothek enthalten sind.

Das folgende Beispiel zeigt, wie man sehr einfach Aktienkurse von einer Website wie Yahoo.com herunterladen und so bearbeiten kann, dass man ein Diagramm des zeitlichen Verlaufs des Tagesendstandes zweier Aktienwerte erhält. Dazu wird die existierende Klasse »HorizontalLineChart« wiederverwendet.

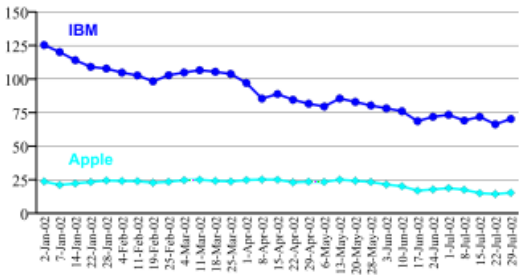
Dies ist jedoch nur eine Spielerei, da ein „richtiges“ Diagramm einer Zeitreihe auch die unterschiedlichen Zeitabstände berücksichtigen müsste. Dennoch gibt es einen Eindruck davon, wie einfach das Grundproblem, nämlich das Besorgen und Darstellen entfernter Daten prinzipiell ist.

Man sieht wie in der Funktion »makeChart« eine Standardversion eines Balkendiagrams erzeugt und dann nach Bedarf parametrisiert wird. Um das Diagramm ohne Qualitätsverlust über den Umweg als externe Bitmap in einem „richtigen“ PDF-Dokument als Vektorgraphik zu verwenden, kann man die »Drawing«-Instanz direkt als Platypus-Komponente verwenden (siehe nächster Abschnitt).

Listing X: Yahoo-Chart

```
# yahoochart.py

import string, copy, urllib

from reportlab.lib import colors
from reportlab.graphics import shapes
from reportlab.graphics import renderPDF, renderPM
from reportlab.graphics.charts import linecharts
from reportlab.graphics.widgets import markers

mm = markers.makeMarker


def makeChart(titles, data, cats):
    x, y, w, h = 50, 50, 300, 125

    # make chart
    lc = linecharts.HorizontalLineChart()
    lc.x, lc.y = x, y
    lc.width, lc.height = w, h
    lc.data = data
    lc.lines.strokeWidth = 1.5
    lc.lines[0].strokeColor = colors.blue
    lc.lines[0].symbol = mm('Circle')
    lc.lines[1].strokeColor = colors.aqua
    lc.lines[1].symbol = mm('Diamond')
    lc.valueAxis.valueMin = 0
    lc.valueAxis.valueStep = 25
    lc.valueAxis.valueMax = 1.2*max(map(max, data))
    lc.valueAxis.visibleGrid = 1
    lc.valueAxis.gridStrokeColor = colors.black
    lc.valueAxis.gridEnd = w
    lc.categoryAxis.tickDown = 0
    lc.categoryAxis.categoryNames = cats
    lc.categoryAxis.labels.boxAnchor = 'e'
    lc.categoryAxis.labels.dy = -2
    lc.categoryAxis.labels.angle = 90
    lc.categoryAxis.labels.fontSize = 8

    drawing = shapes.Drawing(400, 200)
    drawing.add(lc)

    # make titles
    tdata = [(70, 160, titles[0], colors.blue),
             (70, 80, titles[1], colors.aqua)]
    for x, y, str, col in tdata:
        t = apply(shapes.String, [x, y, str])
        t.fontName = "Helvetica-Bold"
        t.fontSize = 10
        t.fillColor = col
        drawing.add(t)

    return drawing


close = lambda l: float(l[:-1].split(',')[-2])
date = lambda l: l[:-1].split(',')[0]

base = "http://table.finance.yahoo.com/"
format = "table.csv?a=1&b=1&c=2002&d=7&e=31&f=2002"
format = format + "&s=%s&y=0&g=w&ignore=.csv"

values = []
for val in ['ibm', 'aapl']:
    url = base + format % val
    lines = urllib.urlopen(url).readlines()
    cats = map(date, lines[1:])
    cats.reverse()
    closeVals = map(close, lines[1:])
    closeVals.reverse()
    values.append(closeVals)

d = makeChart(['IBM', 'Apple'], values, cats)
renderPM.drawToFile(d, "yahoo.png", "PNG")
```

Abbildung X: Ausgabe



Layouten mit Platypus

Bislang haben wir eine Schnittstelle zu einer abstrakten „Leinwand“, auf der wir mit einfachen Mitteln „malen“ können, sowie erweiterte Eigenschaften für separate Graphik gesehen. Was noch fehlt ist eine Möglichkeit auf einer höheren Ebene speziell Text nicht nur als einzelne Zeilen zu setzen, sondern in ganzen Absätzen korrekt umzubrechen und dabei die Struktur eines Dokumentes beizubehalten. Das ist genau, was »Platypus« leistet, ein Layout-Mechanismus, bei dem der Textfluss eines Dokumentes (die von ihm erzählte „Story“) einzelne Rahmen durchläuft. Diese Rahmen sind ihrerseits in Vorlagen für einzelne Seiten und das gesamte Dokument definiert.

Die Grundidee ist ganz einfach, nämlich das Dokument als Folge von „fließenden“ Komponenten zu definieren, die von Platypus beim Rendering auf die einzelnen Seiten platziert werden. Diese »Flowables« können etwa einfache Überschriften, Paragraphen, Bilder oder Tabellen, aber auch Abstände und derlei mehr sein. Ebensogut können es auch Kuchendiagramme sein, die erst in dem Moment gerendert werden, in dem sie Platypus anfasst, um sie ins Dokument einzusetzen.

Einzelne eigene Komponenten müssen oder können möglicherweise umgebrochen werden, zum Beispiel seitlich am Rand oder unten am Ende einer Seite. In dem Fall kann man die Methoden »wrap« und »split« eines Flowables entsprechend implementieren. Folgendes Beispiel definiert eine Unterklasse von »Paragraph«, die in einem Projekt in Tabellen verwendet wurde und die als ganzes eine blaue, klickbare URL darstellen sollten.

Listing X: URL-Paragraph

```
# urlpara.py

from reportlab.platypus import Paragraph

class UrlParagraph(Paragraph):
    "A clickable non-splittable paragraph."

    def __init__(self, text, style, url,
                 bulletText=None, frags=None):
        "Set text color to blue and memorize URL."
        self.url = url
        format = '<font color="blue">%s</font>'
        text = format % text
        # Now, business as usual.
        apply(Paragraph.__init__,
              (self, text, style, bulletText, frags))

    def split(self):
        "Don't allow vertical splitting."
        return []

    def draw(self):
        "Make the consumed area a clickable URL."
        Paragraph.draw(self)
        rect = (0, 0, self.width, self.height)
        self.canv.linkURL(self.url, rect,
            relative=1, thickness=0)
```

Mittlerweile umfasst Platypus bereits eine ganze Reihe nützlicher Komponenten, darunter »PythonPreformatted«, die Python-Quellcode mit farblich hervorgehobener Syntax darstellt und die im nächsten Beispiel verwendet wird. Manchmal möchte man solche Komponenten gerne benutzen, ohne ein Dokument von Platypus drumherum zu setzen. Auch dies ist problemlos möglich, da alle Flowables über die Methode »drawOn()« verfügen, mit er man sie selbst an einer bestimmten Stelle auf eine Leinwand platzieren kann.

Möchte man Platypus mit anderer freier Layout-Software vergleichen, so bietet sich als härtester Vergleich der mit »TeX« von Donald Knuth und den »LaTeX«-Makros von Leslie Lamport an. Aber der Vergleich hinkt, da die Zielsetzung jeweils eine völlig andere ist. TeX/LaTeX sollte eine typografisch perfekte Ausgabe speziell für anspruchsvolle wissenschaftliche Texte und Bücher garantieren, für die es vorher nichts qualitativ Gleichwertiges gab.

Die ReportLab-Bibliothek hingegen ist für die Erstellung von Dokumenten entwickelt worden, die meist in hoher Stückzahl mit leichten Änderungen immer wieder erzeugt werden. Daher liegt der Schwerpunkt nicht auf typographischer Perfektion und Formelsatz, sondern auf Flexibilität und Konsistenz in der Handhabung, einfache Programmierbarkeit und schnelle Ergebnisse.

Ein ganz reales Beispiel

Der Code im Anhang gibt ein „echtes“ Beispiel für den einfachen Einsatz der ReportLab-Bibliothek bei einem „realen Problem“ an, nämlich der Umwandlung des Quelltextes eines Artikels für das Linux-Magazin nach PDF. Dessen Ursache ist darin zu sehen, dass der Autor eigene Texte zwar gerne mit einfachen Editoren schreibt, sie aber ebenso gerne in einer augenfreundlicheren und übersichtlicheren Darstellung (nicht selten auf altmodischem Papier) liest und korrigiert.

Das vom Linux-Magazin gewünschte Format ist ein leicht angereichtertes ASCII, in dem einige Marken die notwendige Strukturinformation beschreiben. So zeichnet zum Beispiel »@ZT:« am Anfang einer Zeile den Rest als Überschrift aus. Innerhalb von Absätzen können auch HTML-ähnliche Tags vorkommen, „ähnlich“ nur deswegen, weil öffnende und schließende Tags identisch sind, letztere also keinen Schrägstrich enthalten, aber das nur am Rande.

Das Programm »lmascii2pdf.py« geht ganz klassisch vor. Nach einer Reihe von Import-Anweisungen wird in der Klasse »LinuxMagDocTemplate« die Dokumentvorlage beschrieben, im Wesentlichen die Rahmen für eine zweispaltige Darstellung, wobei für jede Seite die Funktion »eachPage« aufgerufen wird, die hier lediglich die aktuelle Seitennummer am unteren Ende der Seite platziert.

Danach kommen ein paar Hilfsfunktionen, in denen diverse Ersetzungen vorgenommen werden, zum Beispiel das URL-Quasielement »U« in ein »font«-Element in Platypus, in dem die Farbe auf blau gesetzt wird. Außerdem werden hier Listings aus dem ASCII-Code speziell herausgefiltert. Dabei

wird die Funktion »handleCode« auch von der zusätzlichen neuen Markierung »@CO:« verwendet, die sehr praktisch ist, weil mit ihr auch externe Listings eingebunden werden können.
In der Funktion »main« spielt sich dann die eigentliche Konvertierung ab. Nachdem verschiedene Standardstile erst geladen und dann leicht variiert werden, iteriert die lange while-Schleife über die Zeilen des Quelltextes wobei für jede Marke der relevante Dokumentteil extrahiert und auf eine Platypus-Komponente abgebildet wird. Diese Komponenten werden fortwährend einer Liste namens »story« hinzugefügt, die am Ende einer Instanz von »LinuxMagDocTemplate« übergeben wird, die schließlich das Dokument erzeugt.

Faul sein dank XML

Einige kommerzielle ReportLab-Produkte, die aber nicht Gegenstand dieses Artikel sind, setzen auf der beschriebenen Open Source-Bibliothek auf. Eines davon, »RML2PDF«, sei nur deswegen kurz angesprochen, weil es die logische Fortsetzung von Platypus mit Hilfe von XML ist.
Dabei wurde ein XML-Dialekt namens »RML« (Report Markup Language) definiert, in dessen DTD das Layout von sehr allgemeinen Dokumenten beschrieben wird. »RML2PDF« stellt dann eine „black box“ dar, die solche Beschreibungen direkt und schnell in PDF konvertiert. Das ist natürlich noch eine Stufe bequemer, weil man sich dann auch diese (obwohl schon leichte) Programmierung mit Platypus sparen kann. Die DTD ist sehr einfach und in weiten Teilen ähnlich zu HTML, etwa was Tabellen angeht. Daher ist es mit nur sehr wenig Aufwand leicht möglich, Daten direkt in RML zu erzeugen oder sie dahin zu transformieren. Auch dann behält man alle Möglichkeiten, externen Plugin-Code für eigene Graphiken und Flowables aufzurufen.
Verglichen mit »DocBook« und »XSL-FO« wurde und wird RML sehr praxisorientiert entwickelt, was es beim Vergleich der DTDs als weniger mächtig erscheinen läßt. Gleichzeitig werden aber sehr unterschiedliche Dokumente mit RML erstellt, das aber mit sehr viel weniger Technologie im Gepäck. Weitere Informationen findet man auf der Website von ReportLab.com.

Aus der Praxis

Die ReportLab-Toolkit wird vor allem zur automatisierten dynamischen Erstellung von hochgradig personalisierten, professionellen Dokumenten in Echtzeit eingesetzt. Die Praxis hat gezeigt, dass sie nicht nur zum Beispiel in Projekten mit Fidelity und AIG (beides internationale Finanzdienstleister) bei einem sehr großen Durchsatz an erzeugten Dokumenten zu wesentlichen Zeiteinsparungen und höherer Flexibilität führt. Ähnliche Effekte sind auch in kleineren Projekten zu beobachten, etwa bei der verstärkt automatisierten Herstellung von Druckvorlagen für Dokumente im Bereich »Corporate Identity«.
Weiteres Potential liegt insbesondere darin, dass neue personalisierte Dienstleistungen angeboten werden können, etwa in der Informations-, Kommunikations- und Finanzbranche, aber auch in der Verwaltung. Überall dort türmen sich die Datenberge auf, während keiner so recht weiß, was er damit anfangen soll. Geradezu revolutionär mutet hier das Beispiel einer englischen Fluggesellschaft an, die ihren Vielfliegern maßgeschneiderte Information zu deren Destinationen per elektronischem Newsletter verschickt.
Hierzulande kann sich jeder sein eigenes Negativbild der nicht nur sogenannten „Service-Wüste Deutschland“ machen. Man muss sich nur einmal mit dem eigenen Finanz- oder Telekom-Dienstleister über elektronische Kontoauszüge oder aufbereitete Rechnungen (etwa mit einer graphischen Statistik) unterhalten. So kann man nur darüber staunen, dass bis vor kurzem noch die »Rechnung-Online« als dünne PDF-Hülle um eine schwarz-weiße Bitmap daher kam. Wundern sollte es einen nicht. Heute ist immerhin das »T« magenta, wie es sich gehört, und im Text kann sogar gesucht werden.
Eine Software kann weder die Einstellung in den Firmen ändern noch sie daran erinnern, was ihr eigentliches Kerngeschäft ist, aber sie kann gewisse Projekte ermöglichen, deren beteiligte Partner einen Vorteil vor anderen gewinnen. Die vorgestellte Open Source-Bibliothek ist mit Python als Hauptimplementierungssprache nicht nur sehr leicht einzusetzen und weiterzuentwickeln. Sie kann dank der FreeBSD-Lizenz auch ohne Einschränkungen in eigenen kommerziellen Projekten benutzt werden.

Der Autor

Dinu Gherman ist freier IT-Entwickler, -Berater, -Autor und Übersetzer. Er arbeitet seit ca. drei Jahren mit und an ReportLab-Technologie und hat vier Bücher zum Thema Python übersetzt. Auf seinen Python-Webseiten (siehe unten) findet sich eine Reihe weiterer Beispielprojekte, in denen ReportLab-Technologie eingesetzt wird.

Infos

[1] ReportLab: http://www.reportlab.com
[2] PythonWare: http://www.pythonware.com
[3] Dinu Gherman: http://python.net/~gherman
[4] Python: http://www.python.org
[5] Jython: http://www.jython.org
[6] Adobe: http://www.adobe.com

Listing X: lmascii2pdf.py

```
#! /usr/bin/env python

import os, sys, re

import reportlab.rl_config as rlc
rlc.defaultEncoding = 'MacRomanEncoding'

from reportlab.lib import colors
from reportlab.lib.enums import TA_JUSTIFY
from reportlab.lib.units import cm
from reportlab.lib.pagesizes import A4
from reportlab.lib.styles import getSampleStyleSheet
from reportlab.pdfgen import canvas
from reportlab.platypus.flowables \
    import Spacer, Preformatted, Image, KeepTogether
from reportlab.platypus.paragraph import Paragraph
from reportlab.platypus.xpreformatted \
    import PythonPreformatted
from reportlab.platypus.frames import Frame
from reportlab.platypus.doctemplate \
    import PageTemplate, BaseDocTemplate


def eachPage(canvas, doc):
    "Adornments for each page."

```

```
    canvas.saveState()
    canvas.setFont('Helvetica', 8)
    canvas.setFillColor(colors.black)
    # add page number
    num = "%d" % doc.page
    canvas.drawCentredString(10.5*cm, 1*cm, num)
    canvas.restoreState()


class LinuxMagDocTemplate(BaseDocTemplate):
    "The document template used for all pages."

    def __init__(self, filename, **kw):
        fw, fh = 8.5*cm, 26.5*cm
        f1 = Frame(2*cm, 1.5*cm, fw, fh, id='c1')
        f2 = Frame(10.5*cm, 1.5*cm, fw, fh, id='c2')
        self.allowSplitting = 0
        BDT, PT = BaseDocTemplate, PageTemplate
        apply(BDT.__init__, (self, filename), kw)
        pt = PT('TwoCol', [f1, f2], eachPage)
        self.addPageTemplates(pt)


def handlePara(para):
    "Convert intra-paragraph markup."

    # quotes
    while '"' in para:
        for j in chr(227), chr(210):
            i = para.find('"')
            para = para[:i] + j + para[i+1:]

    # guillemets and URLs
    data = [
        ('<C>', chr(200), chr(199)),
        ('<U>', '<font color="blue">', '</font>')]
    for (tag, before, after) in data:
        pat = '%s.*?%s' % (tag, tag)
        while para.find(tag) >= 0:
            m = re.search(pat, para)
            if not m:
                continue
            start, end = m.start()+3, m.end()-3
            word = para[start:end]
            word = before + word + after
            para = re.sub(pat, word, para, 1)

    return para


def handleCode(lines, i):
    "Handle external code listing."

    listing = lines

    # strip off heading and trailing empty lines
    if listing:
        while not listing[0].strip():
            del listing[0]
        while not listing[-1].strip():
            del listing[-1]

    # number lines if we have more than 10 lines
    listLen = len(listing)
    if listLen > 10:
        format = "%%%dd %%s"
        format = format % len(str(listLen))
        for k in xrange(len(listing)):
            listing[k] = format % (k+1, listing[k])

    # assemble and add to story
    listing = ''.join(listing)

    return listing, i


def handleInlineListing(lines, i):
    "Handle inline listing."

    # extract lines belonging to the listing
    listing = []
    j = 0
    while 1:
        j = j + 1
        try:
            line = lines[i + j]
        except IndexError:
            break
        if line and line[0] == '@':
            break
        else:
            listing.append(line)
    i = i + j - 1

    return handleCode(listing, i)


def main(path):
    "The main meat."

    lines = open(path).readlines()
    print len(lines)

    # Platypus variables.
    story = []
    styleSheet = getSampleStyleSheet()
    h1 = styleSheet['Heading1']
    h2 = styleSheet['Heading2']
    h3 = styleSheet['Heading3']
    code = styleSheet['Code']
    bt = styleSheet['BodyText']
    for h in (h1, h2, h3):
        h.fontName = 'Helvetica'
    bt.spaceBefore = 0
    bt.spaceAfter = 0
    bt.fontSize = 9
    bt.alignment = TA_JUSTIFY
    bt.bulletFontName = 'ZapfDingbats'
    bt.bulletFontSize = 7
    code.leftIndent = 0
    code.firstLineIndent = 0
    code.fontSize = 7
    code.spaceBefore = 3
    code.spaceAfter = 9

    P = Paragraph
    bold = '<b>%s</b>'
    author = ''
    allowedKeys = 'T V ZT IT L IL KT LI B Bi CO'
    allowedKeys = allowedKeys.split()

    i = -1
    while i < len(lines)-1:
        i = i + 1
        line = lines[i].strip()

        key = ''
        m = re.match('^@([A-Za-z]+):', line)
        if m:
            key = m.groups()[0]
            keyLen = len(key) + 2
        elif line[:2] == '* ':
            key, keyLen = '* ', 2
        else:
            continue
        rline = line[keyLen:].strip()

        if key == 'A':
            author = line[keyLen:]

        elif key == 'Bi':
            p = Image(rline,
                      width=8.5*cm, height=200,
                      kind='proportional')
            print "img: %s " % rline

        elif key == 'LI':
            listing, i = handleInlineListing(lines, i)
            p = PythonPreformatted(listing, code)

        # Added to LinuxMag format
        elif key == 'CO':
            clines = open(rline).readlines()
            print "%s (%d)" % (rline, len(clines))
            listing, i = handleCode(clines, i)
            p = PythonPreformatted(listing, code)
```

```
193
194         else:
195             para = handlePara(rline)
196             bpara = bold % para
197             vformat = "<i>%s (%s)</i>"
198             dict = {
199                 'L':  P(para, bt),
200                 'T':  P(bpara, h1),
201                 'ZT': P(bpara, h2),
202                 'IT': P(bpara, h3),
203                 'IL': P(bpara, bt),
204                 'B':  P(bpara, h3),
205                 'KT': P(bpara, h3),
206                 '* ': P(para, bt, bulletText=chr(110)),
207                 'V':  KeepTogether([
208                       P(vformat % (para, author), bt),
209                       Spacer(0, 12)])
210             }
211             p = dict.get(key, None)
212 
213         if key in allowedKeys + ['* ']:
214             story.append(p)
215 
216     # Build the PDF document.
217     path = os.path.splitext(path)[0] + '.pdf'
218     doc = LinuxMagDocTemplate(path)
219     doc.build(story)
220 
221 
222 if __name__ == '__main__':
223     main(sys.argv[1])
```