Document Hacker

Writing Long Documents For Software Engineers using LibreOffice and Python UNO

Jamie Boyle
Issues for the Author
Introduction
Who’s This Book Aimed At

This book is aimed typically at software engineers or developers. Certainly it is aimed at people who know how to program. Having said that, you could probably copy and paste a lot of the Python code even if you’re a bit amateur at programming.

Beyond knowing about software development, it is aimed at people who are interested in writing long documents. However, the sections on processing documents with Python UNO will be of interest to anyone looking at automating LibreOffice, even if it is for mail merge or creating receipts. If you’re a cynical programmer, you could say that the section on how to write and structure long documents is a vanity piece from someone with now real knowledge about how to write good documents, and will choose to skip that section. You’re probably right, and I won’t begrudge you it.

Throughout the book, there is a focus on consistency and style of the document that the reader is creating. It’s not aimed at the professional writer, who probably knows more than me, but it is aimed at people who want to make something pretty close on professional standard. That means that not only is the content clear and readable, but also the document is designed to be attractive to look at. It’s trying to strike a balance for busy people who need to make better looking documents, but who aren’t writing for a living.

Different Thinking

The software engineer thinks and behaves in a different way to most other people. They are used to projects that need clear structure, but are also used to the concept that what they create initially may be wrong and need correcting. This book is aimed at people that have that background, and are writing a long document. By long, likely more 10 pages, although it may be relevant for shorter documents. We are talking about the stage when you move from being able to manually tweak all the formatting to the stage when it becomes too laborious and error prone.

Ideas are often presented with reference to programming concepts to enable the reader to better associate with why we use certain techniques.

Ability and Desire to Automate

Most software engineers have an unhealthy desire to automate wherever possible, rather than manually do a task. It’s not just a time based logic - often it’s not known how many times a task will be done, and it’s not known how much time it will take to set up automation; it’s just more fun to program a solution than to do boring task several times. Despite the dubious motivations for starting to automate, often it pays off. With editing now automated, we are better able to make design changes, and those design changes can be more subtle and detailed. For instance, using a different font for the first word of a paragraph can be easy when using programming, but prohibitively tedious when doing manually.

Often more important than simply saving time is increasing correctness. If you do a repetitive task hundreds of times, you will do it incorrectly many times. Automating generally allows you to create better consistency and have more confidence that a document is consistent.
It's Often Easier to Code Something than Look It Up

You know you're a true programmer, when you find it easier to use an API than a UI. There are some things that are really easy to say, pretty easy (and fun) to code, but damn hard to find out how to do with a one-size-fits-all user interface. The truth is that there's a learning curve for the user interface and to program LibreOffice. For most people it's easier to learn the UI, but for programmers, it's often as easy to learn the API.

When you get into programming LibreOffice, you start to find yourself opening documents with a python terminal so that you can quickly run those special searches or features. It gets addictive.

Mixing Elegant Solutions and Hacks

The programming part of this book is not about making your own next-generation word processor. It's about adding some hacks to existing software to make it do exactly what you want. No re-inventing the wheel here. It is about getting results quickly - the task is to create great documents; not to become awesome software developers.

Given we want quick results, it is likely that at the end of using all your new home-made super-tools, you will still want to make a few manual tweaks to make it just right. This is where re-using an existing word-processor becomes really powerful, compared to e.g. creating a markup language processor that outputs PDFs directly.

My Interest in Writing

So how did I start on this course, and why am I writing it? When starting my own business, I needed to write a business plan, which is used to attract investors. These business plans start at around 30 pages, and are not just a plan, but a marketing document. As a marketing document, they need to show clarity of thought, and look good - they're a hint at the standard of output that the business will have. Business plans also exist in a very competitive environment, and a very subjective environment - the reasons why people invest in certain businesses are not always clear cut or logical.

Historically I have also had a background interest in document design and structure. Writing documents for my company has been an excuse to explore this area more.

The final reason for writing it is that I want to give something back to the open source community, and Libreoffice in particular. It's not just about being a good citizen, or paying back for my extensive usage to date, it’s also about trying to move on Libreoffice. Doing this kind of automation of Libreoffice takes the reader one step closer to contributing features to the main distribution, and IMHO Libreoffice in particular needs more contributors - it's really good, but for such a major and mature project, it's not as good as it maybe should be.

A Book that Practises What It Preaches

This book is being compiled using the same techniques that it preaches - actually, you can use the source documents for this book to experiment with. See the licensing details for more information about what you are allowed to do with the sources, but in general, the motivation for writing this book is not to make money, but to share knowledge, encourage users of Libreoffice, and to get a bit of recognition. Therefore, so long as you give a nod of acknowledgement, you're probably fine republishing aspects of the book.

This Book is not Dogma

Much of this book should be considered like a book on design patterns - you are learning the theme
about what to do; but not necessarily looking to implement everything exactly as described here. This is part of the reason why it moves into a cookbook style - you should pick and choose what to do, and it is likely that for your situation you should move beyond what is documented here.
# Structure of the Book

## Sections

This book is divided into three sections:

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>How to Write and Design a Long Document</strong></td>
<td>A tutorial for creating better documents. This includes an introduction to the considerations of structure and design. It is an abstract discussion that is mostly relevant to any office package; not just Libreoffice.</td>
</tr>
<tr>
<td><strong>Automating Libreoffice with Python</strong></td>
<td>This introduces the reader to automating Libreoffice with Python. It goes through enough of the basics and some examples to take you to the stage when you can use the cookbook, and even go beyond the cookbook to look at the (not-so-great) other documentation online.</td>
</tr>
<tr>
<td><strong>The Python UNO Cookbook</strong></td>
<td>Self contained examples are provided for doing a variety of tasks. Jump in, and take what's needed.</td>
</tr>
</tbody>
</table>
Programming OpenOffice with Python Tutorial

Getting Started

Programming OpenOffice with Python Tutorial

Getting Started

LibreOffice vs OpenOffice from a Programmer's Perspective
Versions of Python and LibreOffice Used For this Book

Learning More

LibreOffice and OpenOffice Extensions
Converting Java Examples to Python Examples - a Lesson

LibreOffice Python UNO Cookbook

Introduction

Background
Prerequisites
Convention and Style
Code Style / Formatting

Opening Documents

Open a Document So That You Can Use Python's 'with ... as ...' Statements

Document Publishing

Document Publishing Overview and Use Cases
Converting To PDF
Saving a Document to HTML

Working With Styles

Set Styles Across Several Documents

Fields and References

Inserting Page Number Field
Inserting Page Count Field

Tables

Creating a Table
Get Existing Tables In a Document and Edit Them
Get a Cell by Name e.g. A1
Get a Cell by Position in the Table
Get the Text in a Table Cell
Change the Text in a Table Cell
Iterate over All Cells in a Table
Deleting Rows of a Table
Adding / Inserting Rows and Columns to a Table
Change the Background Colour of a Whole Table 49
Change the Background Colour of a Whole Table, With All Its Cells 49
Change the Background Colour of a Table Cell 50

Writing Code 51
Introduction 51

Headers and Footers 52
Adding a Header 52
Adding a Footer 52
Adding a Header / Footer with Page Numbers 52

Working With Headings 54
Getting the Outline Level of a Heading Style 54
Listing All Headings in Document Order 55
Demoting All Headings To a Lower Level 55

Managing Changing Documents 57
Removing Notes from Output 57

Processing the Raw ODT Document 58
Extracting the Contents of an ODT File 58
Re-compressing an ODT File 58
Debugging and Fixing Corrupted ODT / LibreOffice Writer Files 58

Tricks to Help Programming 61
Use iPython Console 61
List the Interfaces a Libreoffice Object Implements Nicely So You Can Look Up Its Functions 61
List the Interfaces an Object Implements with Links To the Documentation 62
List the Properties of an Object (for getPropertyValue, setPropertyValue) 63

Writing a Long Document
Motivation

When a Long Document Is Appropriate

Before you start writing, it's important to understand why you are writing the document, and to know whether it is the most appropriate means of communication. If you're writing it because your boss told you to, then you've got the wrong motivation and need to consider if your boss has thought it through. It's too easy for someone to say “write up how to use xyz” and for that to be interpreted simply as write a technical manual, and distribute it. You need to think more deeply about how the information will be created and distributed.

Checks on suitability of long-form documents

<table>
<thead>
<tr>
<th>Question</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Will anyone actually read it?</td>
<td>Long documents require someone to sit down and read it. The longer it is, the larger the barrier to getting started. Are the people you're writing it for interested enough to actually find the time to read it? Long documents (irrespective of what people say in the introduction) suggest reading cover-to-cover.</td>
</tr>
<tr>
<td>Will it be easy to find and accessible?</td>
<td>A PDF stored in a document server somewhere on the network won't be found. Initially you may e-mail it out, but if it has long term relevance, is publishing it as a long form document really the best way?</td>
</tr>
<tr>
<td>How much will it change over time?</td>
<td>If the document is continuously changing, it may be better to use a wiki, or similar pure web-publishing system so that everyone always has an up-to-date copy.</td>
</tr>
<tr>
<td>How many people will contribute?</td>
<td>You can break long documents down into several sub-documents, and collate them. However, the more people that contribute, the more active management of the process is required. For instance, you will continually be trying to get everyone to use the same formatting standards. Sometimes it is better to give up on the beautiful document in return for less time spent managing the process using a wiki.</td>
</tr>
<tr>
<td>Do you have time to write it?</td>
<td>It may be that the best way to communicate is in a detailed, long document, but that is no use if it will always sit half-done with you never having the time to complete it. Creating a document bit-by-bit, and using continuous deployment is an underlying theme of this book.</td>
</tr>
</tbody>
</table>

Long documents are however often the best option. It's one of the great surprises over the past couple of decades that the internet hasn't killed the writing process, and in particular it hasn't killed the book. Despite claims that videos should be 1 minute 45 seconds or we won't bother to watch them, product reviews now go into extreme multi-page detail and books are flourishing. Most interestingly is that books on how to program still exist, despite the wealth of information available free on the internet, and despite the audience being the most tech-savvy an therefore most likely to move beyond the printed book.

So why does the long document, and even book still exist nowadays? Long documents provide consistently presented, and complete coverage of a topic that shorter documents don’t. They are concise and accessible. Consider buying a new digital SLR camera, having already owned one. I not only know something about SLRs already, but from my experience of my current model, I know...
certain features that I really want, such as fast focusing, a responsive user interface and easily accessible buttons. It’s time to do some research on the new models in the market today:

**Researching a new Digital SLR**

<table>
<thead>
<tr>
<th>Newspaper reviews, such as The Times</th>
<th>Dumbed-down to the lowest common denominator, their short article stating that “SLRs take great photographs... SLRs are however larger than a compact camera...” doesn’t tell me anything useful.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon Reviews</td>
<td>Inevitably 5 stars all round as everyone likes their fancy new toy, but to get useful information on how responsive the user interface is takes reading 20 reviews; most of which contain little of interest.</td>
</tr>
<tr>
<td><a href="http://www.dpreview.com">www.dpreview.com</a></td>
<td>Monster reviews, often spanning 16 sections of dense text, graphs, examples and diagrams. I won't read all the text for all the cameras, but might for the camera I eventually buy. Being well structured documents, I can skip to the parts that interest me. I read the introduction, the section on user interface and the conclusion.</td>
</tr>
</tbody>
</table>

So why do I use DPReview? Why does it often come top of google searches for an SLR review? Because SLRs are cameras for enthusiasts and enthusiasts care about the detail. If I bought an SLR today that took even a second longer to turn on than my current SLR, I would regret the purchase for the rest of time. However, if I bought a cheap-ish compact camera, I wouldn’t care about many features - I’d just buy something cheap-ish, and probably be pleasantly surprised with the photographs that come out of it.

It’s simply a case of considering who you are targeting with the document. Are you targeting the compact camera market that just wants to know that the camera is good value, and if it has 4 or 5 stars, they will buy it? Or is it targeted at the SLR market that wants all the detail?

In the camera market, we had clear differentiation based on the product between those who are interested in the extra detail, and those who want a simple summary. It’s not always that clear. Take this book for instance. Lots of people use LibreOffice. Lots of people want know how to use it a bit better. However, few want to know enough to justify reading a book about it. By writing a book, I am targeting a super-interested niche. It would have been equally valid to write a smaller, more accessible document, or a series of blog posts. If I wrote a smaller article, it is likely that more people would read it, but not necessarily my target market. For this reason, it is common for authors of books to write short articles summarising their books, or to appear on TV giving summaries of some of their book contents - you are making the information more accessible and therefore you attract a larger number of people, some of which may buy they book.

**When Long Documents Work**

<table>
<thead>
<tr>
<th>Targeting an audience that demands detail</th>
<th>Often your reader really does want to understand your subject in depth. Maybe the document is for a big decision - which company would feel comfortable spending £100M without a rational and detailed argument for it? Alternatively it may be targeted at that highly interested or expert niche that wants to move beyond what can fit on one page.</th>
</tr>
</thead>
<tbody>
<tr>
<td>There is a logical flow through the document</td>
<td>Long documents work when there is a logical flow between different essays within it. For instance, a biography of a politician might provide the story of their working class youth as context for their policies described later in the book. Both</td>
</tr>
</tbody>
</table>

Writing a Long Document | 12
parts could be interesting published separately, but together they help form a logical argument.

Adding gravitas

Writing a blog post gets information out there. A published academic paper can be put on your CV. Writing “The Book” gives you far-reaching credibility, and will likely lead to people actively searching you out for your expertise.

If you have written at length, you have necessarily looked into your subject in depth, and therefore have well informed views on the subject. You can therefore be considered not just an opinionated observer, but an expert; maybe even the expert.

This can be important as much in the workplace as in the wider world. It’s a lot harder for a manager to contradict someone who has demonstrated their depth of knowledge on a subject so strongly and publicly.

Personal Motivation for Writing a Long Document

Decisions are never, and should never, be made without consideration of your personal motivation and what you can get out of them. That doesn’t mean you should be selfish, but it does mean that you shouldn’t be embarrassed by cynically evaluating the benefits for you, rather than your audience, of approaching a project in certain ways.

Everyone is on a constant path of self-betterment and self-promotion. In the software engineering world, it is more typical to be interested in becoming better at what you do from a technical perspective, rather than improving your wider reputation. Wider reputation can often be ignored, but it really does matter. It affects not just your pay packet, but also your ability to do what you want.

Value to the Author of Writing a Long Document

| Reputation and credibility | Documents get distributed and read, often by people outside your day-to-day contact group. This can give you a positive reputation within a wider audience, that will pay off in the long term.
If it is a public document, then people can easily verify your credibility by discovering the document online. Often they won’t read it, but the fact that you have written on a subject is sufficient. |
| Demonstrating wider skills | In any job, it is easy to become pigeon-holed. “He’s just a software engineer”. Here is an opportunity to demonstrate not just an appreciation of commercial considerations or other teams, but also to demonstrate your ability to communicate and implicitly take more senior roles. |
| Asserting your position as the position to be challenged or supported | Often documents, whether explicitly, or implicitly, are there to build an argument. “Evaluate different options for …” often really means “provide an argument for your choice for …”. If you write the document outlining the arguments, then people will not only better understand your point of view, but will be lead into challenging your viewpoint and the |
Improving writing and communication skills

Writing a long document gives a lot of great experience. Not only are you simply writing more, and so practising more, but the document needs more careful planning and structuring. This teaches a lot of good habits that can then be used everyday, even in simple, short emails.

The value extends beyond written documents. When you write, you think more carefully about the content and have the ability to change and improve it, which is hard to do when speaking. The benefits will show in the way you speak as well as the way you write.

I experienced the need for better communication and stronger reputation in my early career in particular, and I continue to experience it now. I saw opportunities that would be awesome for the company, however, I struggled to convince management of its value early on. If I was to paraphrase the attitude that managers had, it was “He’s smart and has been going on about it, so he’s clearly convinced. I don’t understand it however. We’ll let him do it and see what happens. We can always cancel the project if it’s not going anywhere.” This has resulted in some pretty awkward moments weeks or months later when they look at my project and still can’t see the value. I’ve been lucky and have convinced them just enough to stay alive, but had I been able to communicate properly at the start, or even just have full trust at the start, things would have not only been smoother, but I might have got more resources to get it done.

Consider what a good, detailed document would have done. There is now a clear explanation of the benefits, risks and estimated time for a project. It is clear that the problem has been properly thought through. Most importantly, the value has been laid out explicitly and the manager has agreed that the value is there. When the manager re-evaluates the project weeks or months later, the conversation can be along the lines of “Since I gave you the document explaining the project, a few things have changed, but overall we’re still on track. The value of the project has not changed, but we are 3 weeks behind with development having found that xyz was harder than expected.” Underlying the conversation is the message that the manager signed up to the value of the project, and a small hiccup doesn’t change that. If the manager wants to challenge the continuation of the project, then they must challenge their own original decision to support the project.

Alternatives to Long Documents

Communicating information doesn’t need to be done through traditional essays or books any longer. It is worth considering if it is appropriate to use a different medium to communicate to your audience; even if the traditional document is the most obvious option.

Long documents have limitations - the most important of which are:

- Long documents are less inviting to read
- Other systems can be better for collaborative and more rapidly changing documents

Therefore, you should be open to considering other forms of distributing information. Think widely, and consider how other people and companies do it.

Alternatives to Traditional Documents

| Video | Videos only work if they are short, and increasingly standards need to be high (blame Kickstarter). However, they are more interesting |
and inviting than a plain document.

**Slide Show**

Slideshare has been increasing the amount of presentations on the internet. Few are good to read, but often they give a basic overview of a topic and if you’re making a presentation anyway, a few small changes can make it understandable as just the slides.

**Website**

Web sites work well for less structured information. On a blog, we don’t expect the posts to have any strong link between them. Similarly, technical documentation works well online as it’s easy to find and can be accessed easily in a random order.

The ability to leave comments and to collaborate is also a major advantage.

**Wiki**

A great way to publish documents that need to be updated continually. Ask yourself if you are actually going to keep a document up to date, or whether it will rapidly become low priority. If it will become low priority, publish a wiki so that other people can contribute and you don’t become the bottleneck to progress.

**Facing up to a Long Project**

Just like developing software is never as quick or simple as you expect, similarly, writing a good long document is never as quick or simple as you might hope. A good document needs researching, planning, writing, formatting and then it needs to be done all over again when you or your helper reads through it and realises it isn’t quite right.

With software, often you are creating it for someone who doesn’t really know what they want. Generally documents have less risk here - the task is pretty clear, and any uncertainty can be removed in the design stage. I think that documents can be worse in the small tweaks. More people understand documents, so more people can have input. This can vary from “it needs a semi-colon there” to “it’s too long”. The latter comment being the most scary. Reducing the size of documents can be an energy sapping experience - you thought you’d finished it, and now you have to essentially re-write it, but all the time trying to fit everything into a smaller space. This is another reason why taking time to perfect your writing style early on can be extremely valuable - getting it right first time will save a lot of time.

Estimating the time that a document takes to write is very hard to write. I have written two page marketing documents that have taken weeks of tweaking and feedback to make them just right. Then there are other documents, where the content and style doesn’t need to be as precise, which can be distributed after a quick read through to check for obvious mistakes. We can state some very approximate rules of thumb to help, but the main message is that it will take a long time, be prepared for this, and don’t worry if it happens.

**Contributors to Writing Time**

<table>
<thead>
<tr>
<th>Fitting within a fixed word or page count</th>
<th>Getting, and responding to feedback</th>
</tr>
</thead>
<tbody>
<tr>
<td>If a document turns out to be too long, you are faced with tough decisions about whether to leave out whole sections, remove unnecessary words, or rewrite more concisely. This can take as long as it took to write the document originally.</td>
<td>Getting someone to proof-read a document is near-essential, but as soon as you get into a cycle of making changes and getting feedback, the time taken will increase. This is particularly true as you helper gets less excited by re-reading</td>
</tr>
</tbody>
</table>
your document.

**Re-formatting the document**

It is rare for the writing part to be completed and the formatting not need adjusting. Maybe the titles don’t stand out enough, maybe the text looks too dense, maybe it looks too informal... whatever the reason, something will need changing. This part can be sped up significantly if you have used styles, rather than direct formatting (e.g. wrongly using bold, rather than an emphasis style).

**Charts, diagrams and graphics**

It can be quicker to write a whole page of text than to insert a small chart, even with data that you have easily available. Any form of graphics needs tweaking and matching to the rest of the document contents which takes a lot of time.

**Research**

When you start writing a document, often it will be clear that you believe something, but you need to be more certain about it, and maybe provide independent evidence. Hours of internet browsing may follow.

**Bibliography and References**

It’s a lot faster to write casual opinion than citing sources.

Often it is worth considering how you can avoid some of the things that will take a lot of time. If you are told to write a 30 page document, ask if it’s okay if you go a bit over. If you don’t want to spend the time to really perfect the wording, consider setting expectations low by stating that it’s just an informal discussion document. Have a strategy from the start about how to minimise the time taken while achieving what you want to - and it’s valid to set a target of making a document with an amazing design, just see if you can then skimp on another area such as how well perfected the text is or whether a bibliography is included.

So how long will it take? Estimate as 3-5 hours per page for most documents, but as much as 9-10 hours per page for more complex technical documents or for short documents that need to be just-right such as feature articles or newsletters.

Consider what this could mean - if you’re writing a 30 page technical document, with 9 hours per page, and with 7hrs per day dedicated to the task, it will take almost 8 working weeks.

**How to Enjoy the Writing Process**

Creating a great document takes time, and effort, but it can also be a hugely enjoyable and satisfying task when done right. The key to doing any long task is to find enjoyment in the process, and not just the result. Many of the fun parts of programming are present in the writing process. Actively seek out these parts, and you will better understand why writing is enjoyable.

---

**Why Writing is Fun For Software Engineers**

| Complex interactions between a large, complex system | One of the most challenging parts of software engineering is keeping in your head all the different parts of the system, and understanding the interactions between them. If you change one part, often it will affect another part. Similarly, a document can be considered a collection of mutually dependent parts that when altered have knock on effects. For example, if I change an argument about target market for a product, then it will also affect the argument for revenues. |
| Open ended problem with | When you set out to write a document, there are often few |
many possible solutions constraints. Maybe you have been set the title, or been told what you are trying to communicate, but there is a huge amount of freedom about how you do it. Explore these options and their implications.

It's as much about what you leave out as what you leave in Finite time is available to write software and documents alike - you need to decide not just what is important, but what can be implemented with the available resources.

Elegant algorithms In software, there are often many ways of implementing a function, but some are more beautiful than others. Within a few lines, they state everything perfectly, and in a readable manner that other implementations take ten times as many lines to do. Similarly, in writing documents, we look for deep insights and concise arguments that make you stand out.

Space for constant improvement The more you write, and think about your writing, the better you become and the more you appreciate the writing around you. Soon you will automatically analyse and evaluate not just the content, but the style and design of other documents that you read. Then trying to imitate and elaborate on what you see becomes a game in itself.

Writing documents becomes enjoyable when you stop viewing it as a task to get done, but as an art form and a challenge. It is not about creating a dumb, pre-prescribed output, but about creating a solution with creativity and insight.
Overview

There are a number of different options for creating really good documents. Ask a pro, and I bet they won't suggest using LibreOffice. They might suggest Microsoft Office, because it's what everyone's got. Alternatively they might suggest a desktop publishing package (DTP) such as Adobe InDesign for more professional results. If they were a bit more geeky, they might point you to Latex. If you are printing a lot of repetitive documents such as invoices, they might suggest using a PDF generating library. Still, LibreOffice wouldn't come up.

That doesn't matter. You came here because you know, use and love LibreOffice. It's not perfect, but it's pretty damn good, it's free and it works consistently across Linux, Mac and Windows. Learning a DTP package is too much effort, and you're never going to use it again. LibreOffice is that trusty tool that's always there, and that's why we're going to use it.

I'll go over a few of the other options mostly just so you've got some comparisons - by looking them up, you might understand how you could use LibreOffice better.

LibreOffice and OpenOffice

They're free, open source and cross platform. Great.

For the rest of this document, you can assume that referring to one is the same as referring to the other. If you're interested, you can google the big battle between the nasty Oracle and the happy loving open source community that have lead to the split into LibreOffice and OpenOffice. As far as I can tell, at the time of writing, LibreOffice has the community and momentum behind it, but they are essentially the same.

I think the major benefit is that it provides a familiar interface that you use for all your documents. There might be better tools for the job, but it's a jack of all trades office suite, and for our needs, that's good enough. LibreOffice has traditionally been quite good with long documents, and actually used to be IMHO better than Microsoft Word, but Word has caught up.

If I was to critique it, I see a couple of problems:

• Many features that we are implementing using Python should already be in the Writer

• Its compatibility with Microsoft Office is a lot worse than would have expected

• The documentation for automating it with Python is abysmal (but we're changing that here, so don't worry).

Microsoft Office

Probably better than LibreOffice, but it's commercial and doesn't work on Linux. If you were sensible, you'd probably use it as the programming skills you're going to learn will be more broadly applicable.

I've used it. It works. For whatever irrational reason, I prefer LibreOffice.
LaTeX (and e.g. LyX)

Apparently used a lot in academia. LaTeX is a markup language, which you compile into a document (in whatever format), but there are a number of GUIs to make the editing process more like using an office suite, often referred to as WYSIWYM (What you see is what you mean). It's great for things like writing mathematical and chemical formulae as well as handling large documents.

In general it behaves a bit like HTML in that you use tags such as (in HTML) `<H1>` for the heading. Then you tell the program how to turn those tags into nicely rendered text. The advantage of this system (which is actually similar to how we will use LibreOffice) is that you can easily change the formatting for the whole document by just changing the style of the tags.

If you're reading this book, you should probably use LaTeX, but in my experience, LaTeX is one of those things that people say “I really want to learn how to use it”, but never get round to it. I never have.

Commercial Desktop Publishing (e.g. InDesign)

If you talk to a hipster who does this for a living, they're almost certainly using something made by Adobe, in this case InDesign. Again, I've never used it. It costs over £600 on Amazon. I'm sure it's features for making interactive eBooks are great, beyond its traditional use for Typesetting.

If you want to spend a long time making a really nice looking brochure, with text that wraps around an image just how you want it, then you should probably consider this. For our needs, it's overkill.

Wiki

Many long documents are successfully created using wikis, such as the wikibooks collection. Even if it is just you creating the document, there are a number of advantages from creating a wiki such as the ability to trace changes, and edit it online. They really come into their own when you create a collaborative document, or a document where the contents are likely to change.

Personally, I don't like being tied to an internet connection, and I would rather use a GUI that shows me something approximating the end output first. I also don't feel that wiki books look that great online. If you do use a wiki, spend some time customising it - make it your own. However, much I love Wikipedia, using their style for your own content looks dull and unoriginal.

I have also generally found the PDF renderings to be mediocre. I'm sure this is something that you could work on however.

Custom PDF Generation

There are libraries that allow you to create PDF documents straight from Python. These documents would then either be printed or published online.

If I was creating a tool to print nicely formatted receipts from a database, I would probably use this method. It's great for documents that you're not going to tweak later, and lower level control means that you can get everything consistently just right (with enough effort). For many simple tasks, it shouldn't take long to pick up this technique.
Design

What Makes a Professional Looking Document

When you pick up a document, you immediately judge it, and its author by its aesthetic. Does it look professional, or knocked together? Does it look like a dense academic paper, or approachable and well structured? The first impressions - when we look at its cover; when we look at the first page of text; and when we flick through the document to get a feel for it - they really matter. Often they make the difference between reading and not reading the document, or more subtly, great design can encourage us to keep reading after the first chapter.

The key message is that design should not be dismissed as pointless aesthetic. The purpose of writing a document is to convey a message and good design helps you to convey that message.

Most elements of design take a bit of time to learn, but once you start learning about it, you start experimenting and you soon find that the basic level of design used in all your documents has become significantly better. As I have become better, so I have started to set up all my documents so that they look better from the start - it takes me five minutes, but makes a big difference to the output.

When you consider the important parts of great design, it is worth noting that nowhere does it mention fancy graphics, or the use of colour. This is a more traditional and timeless form of design - we're talking about the fundamentals, not the gloss. If you get the fundamentals right, your documents will stand out, the content will be clear, and you won't need fancy graphics.

Avoiding Common Pitfalls

Good design is not just about what you do, but also about what you don't do. With a better base design to your document, you shouldn't feel the need to force in desperate measures such as big graphs to break up your document (and because professional documents have graphs right?).

Large Tables of Data

Tables of data are often inserted without enough consideration of their content - is it easy to read, and is it really relevant? The best use of tables is when they're super-small, and often put in the margins as an aside, rather than breaking up the flow of the document.

If large tables of data are needed, then they should be placed in the appendix.

Graphs

Graphs might have looked good in the 90s, but generally they are used poorly. Does your graph really show the data better than putting the numbers in a table? Is it really easier to read? Is there any insight that you can only get from looking at a graph?

Graphs can be appropriate to show trends, typically over time. In this case, the shape is of interest, and the shape should be obvious, so the graph should be made small - the reader shouldn't be expected to read values off a graph.
Graphs are not only over-used, but they also take a long time to construct and format correctly. Consider therefore that often it would be a lot easier to just not include them.

**Bullets**

It took me a long time to realise this, but bullet points should almost always be avoided. For anything more than about three key points, they are hard to read, or browse, and the points don't stand out. Generally a table like this one (which we will call Words in Table, or WiT later) allows the key points to stand out properly and encourages the writer to properly justify the contents.

**White Space**

The key with white space is to always use more of it, much more of it than you ever think. Part of the problem with white space is that many office suites, and LibreOffice in particular, don't by default include enough. For instance, the default padding around tables is minimal and titles are cramped in next to the text that precedes and follows them.

**Variety**

If you follow the rest of the tips, for creating white space and making a document easy to browse, you will find that there is little need to add extra variety to your document. Not only will you have enough variety, but you will have included it in a way that retains the consistency of the document.

Should the document not appear to have enough variety, then careful consideration should be made about how to increase the variety without expending much effort. Adding pictures is hard to do well and quickly - you don't want to turn your document into a modern day clipart nightmare and ruin all the other careful work you have done so far.

Here are some simple and quick ways to increase variety:

---

**Quick Ways to Increase Variety**

| Split long blocks of text into sections | If a you have a long block of text, which isn't broken up, consider dividing it up into sections with their own heading. |
| Replace paragraphs with Words in Tables (WiT) | Often within a paragraph, or a block of text, we make several points and explain them in some detail. We fail to separate the bullet-point style key messages from the explanation. This is much better to do through a WiT. It not only makes it easier to browse the contents, but also breaks up dense blocks of text. The general rule is that you can never have too many WiT. |
| Add stand-out quotes | Stand-out quotes, where you separate out a key message from a paragraph with a large font and possibly even in a separate colour works well to break up the dense text and help the reader skim to the important parts. |
| Divide into chapters, with differently formatted chapter start pages | The problem might not be that a particular page seems to lack variety, but that the document as a whole doesn't have sufficient variety. In this case, divide the document into chapters with a luxurious start page to the chapter. Consider having just the chapter title on the start page, maybe with a small table of contents, but without any of the actual chapter contents. |
Other means of increasing variety such as including images, graphs or tables should be very carefully considered before being used. Doing them well takes a lot of time and effort and they will likely distract from the purpose of the document. They should be only used when they also help the reader understand information. Your aim probably isn’t to create an amazingly glossy brochure, but to create a document that’s great at communicating a message, so focus on what’s important.

**Browsability**

Long documents should be structured in a such a way as to allow the reader to flick through them and pick out the interesting parts. This can be for a number of reasons - to decide whether to read the document, to decide to only read part of the document, or to refer back to parts that they have already read.

Beyond adding a table of contents, the primary way to make a document browsable is to have text that stands out from the main body and which quickly informs the reader of the content around that stand-out text. This can be done through headings, or through devices such as standout quotes that highlight key points from within the paragraph. The key here is to make sure that you are highlighting important details, and not over-using it. If you highlight too much text, then nothing really stands out, and the text will likely be less important and so the reader will be turned off to the relevance of the text that stands out.

**Making a Document Easier to Browse**

| **Use Words In Table (WiT)** | Rather than using paragraphs with several points inside, or bullet points with too much descriptive text hiding the important point; use WiT to make the key points stand out to the side. |
| **Give charts and tables headings** | Give tables and charts headings that allow the reader to understand not just what the chart is about, but why it is included. |
| **Add Standout Quotes** | These are over-sized quotes that really stand out from the text to make a point. You often see them used in magazine articles. Typically they repeat elements from within the body of the text. |
| **Use (more) Headings** | Divide the document into sections, and subsections. Have a heading for each point being made or area covered - don’t be afraid to have a heading for a short body of text. |
| **Include a table of contents** | If you have correctly used headings in the document, this will be trivial to create, so it’s a no-brainer. |
| **Include an index** | An index where you can look up a word or phrase from an alphabetical list can be useful for certain reference documents, but requires diligence and careful tweaking to ensure completeness and relevance. Often it will not be worth the effort. |
| **Includes Page Numbers** | Easy to forget, but super-important. |
| **Include Line or Paragraph Numbers** | More often used for academic or legal documents when precise referencing to the text is required. It looks out of place in a regular document, and should only be used in special areas such as code samples. |

A lot of documents are structured around areas of interest, rather than conclusions, when the conclusions are at least as important. For instance, the heading may be “Factory Outputs for the Middle East”, which may or may not be interesting to the reader according to the conclusions, so you
might want to highlight the fact that there are interesting conclusions with a stand-out quote such as “50% of factories will need to close”. Alternatively, you can put this in the title of the section “Middle East Factory Outputs - 50% of Factories Will Need to Close”. Putting the conclusion in the heading is particularly effective for tables and charts, where you want to quickly guide the reader as to why the table or chart is important. It is less effective for section titles in part because it makes a confusing index to browse, so a sub-title can be an effective tool here.

Evaluating Your Design

Once you have decided on a design, you need to evaluate it to determine if it actually works. The first step is to look at the document as if it's in the distance. The best way to do this is to zoom out so that you have several pages on the screen at the same time. As you zoom out, the body of the text will become harder to read, and you will only be able to see the titles and the structure of the document. Here are some checks:

<table>
<thead>
<tr>
<th>Design Checks when Zoomed Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can you see the headings and sections clearly?</td>
</tr>
<tr>
<td>When you have zoomed out to the extent that you can't read the body text, you should be able still to read the headings, and you should see clear separation between blocks within the page - here's a table, there's a graphic, there's a new paragraph.</td>
</tr>
<tr>
<td>Does it look like a big block of text?</td>
</tr>
<tr>
<td>It should never look like a big continuous school essay. If you have a page full of paragraphs of text, consider breaking it up with WiT or headings. Big blocks of text are not just a sign of bad design, but of unclear thinking - you are either writing too much text to justify one point, or you're not separating out the points you are trying to make.</td>
</tr>
</tbody>
</table>
Design Details

Font

Technically a font is the combination of a typeface (such as Arial), and a size (12pt). However, colloquially it is often used to refer simply to the typeface. Typefaces can be separated into two categories, serif and sans serif. Serifs are little marks that are added to letters to, in theory, make them easier to read.

Typeface Styles

<table>
<thead>
<tr>
<th>Serif</th>
<th>These have little marks that supposedly help when reading the text. A common example is Times New Roman:</th>
</tr>
</thead>
</table>
| Lorum Ipsum | Look at the 'L' and see how the top of the letter has a horizontal line, and similarly at the bottom, there is an extension to the left and an upturn of the base line on the right. These little marks are called “serifs”.
For instance, the ‘o’ is thin on top and thicker on the sides. |

<table>
<thead>
<tr>
<th>Sans Serif</th>
<th>Sans Serif fonts, don’t have serifs (obvious if you know a bit of French). The theory goes that they are harder to read. A common example is Arial:</th>
</tr>
</thead>
</table>
| Lorum Ipsum | Again, look at the 'L' character and see that it is simply two perpendicular lines. This makes it seem cleaner and less cluttered.
In particular it makes it more modern. |

The traditional belief is that sans serif typefaces are harder to read, especially for large blocks of text. This is why novels traditionally have been written with serifs. However, the current fashion is to use sans serif fonts on websites in particular, and increasingly in all printed text as well. Now serifs look old fashioned, and long form documents written with serifs look like they were written by academics, rather than professionals. Typefaces with serifs can be used effectively however for document titles.

When you get into the world of document design, you start to find that people get obsessed with typefaces. It’s very similar to product designers’ obsession with chairs - all designers want to make their signature chair to show off to other designers. Product designers’ obsession with chairs and graphic designers’ obsession with typeface is to the outsider nothing more than internal sparring; don’t feel like you need to create your own typeface, or spend ages trawling the internet for the perfect font. A slightly dull truth is that Arial looks perfectly good most of the time. The one exception can be with logos and company / product branding. Here it can be good to use something a bit more special to stand out, and then use that typeface consistently between the logo and the headings within documents.
English Grammar

What You Failed to Learn at School

There's a stage in everyone's life when they realise that they weren't really taught how to write English that well at school (and not just because you grew up in America), and you certainly weren't really taught how to write a decent document.

The kind of questions I will attempt to answer here are:

• When to use a semi-colon?

• How do you capitalise titles?

• Should I say indexes or indices? Matrixes or matrices?

Now I'm not an English teacher, but what I have done is some research to try to uncover the truth. Often there is some debate about how to do things “right”, but I'll try to guide you through the chatter of contradicting voices.

While many people are happily lazy about a lot of these areas, incorrect usage tends to really annoy people who actually learnt them at school. Therefore, if you want the maximum positive impact from your document, it's good to get them right.
Further Reading

Here are some other books that I have read that might be of interest. I've read them, and they've typically got good reviews on Amazon, but you might find better books yourself.

**How to Make an IMPACT**

How to Make an IMPACT: Influence, Inform and Impress with Your Reports, Presentations, Business Documents, Charts and Graphs: Influence, Inform and ... Business Documents (by Jon Moon, Financial Times Series)

Focuses on the reasons behind the design of your document, than on just the aesthetic. Really good to think about, and helps clarify you're thinking. Despite being well written, it's a bit of a strain to read cover-to-cover. A Significant influence on this book.
Preparing for Processing

LibreOffice and Styles
Programming OpenOffice with Python Tutorial
Getting Started

LibreOffice vs OpenOffice from a Programmer’s Perspective

Currently, there doesn’t appear to be a need to worry about the differences between OpenOffice and LibreOffice. Much of the code base hasn’t changed for years. When researching this book, many of the examples were from nearly 10 years ago and still work fine. A reasonable person would assume that unless knowingly using a new feature, which may or may not be included in both LibreOffice and OpenOffice, you can safely ignore the differences.

Versions of Python and LibreOffice Used For this Book

The current setup used for writing this book is:

- LibreOffice 3.6.2
- Python 2.7
- Ubuntu 12.10

At the time of writing, the author doesn’t have the resources to test with different versions, and in particular not with Python3. However, it is thought that most code examples will work out-of-the box with most versions of Python, LibreOffice and OpenOffice.
Learning More

LibreOffice and OpenOffice Extensions

There are a lot of extensions available on:

- http://extensions.libreoffice.org/
- http://extensions.openoffice.org/

Extensions tend to be fairly simple additions and so can provide good example source code. Typically they will also have been used by a variety of users and so will have better quality code due to errors having been identified.

Extensions are simply zip files. When you extract them, generally the source file is available. Beyond the actual “software”, it also includes details of how it will be added to OpenOffice, icons etc. Therefore many files you can often ignore.

Tips for various formats

OpenOffice Basic (.xba)

These files use HTML / XML style conversion from characters such as turning the quotation mark into &quot. If you open in a browser, often it will convert these, but it might not format the document nicely.

In Python3 it’s easy to convert:

```python
import html.parser
f = open(my_xba_file_path)
text = f.read()
f.close()
clean_text = html.parser.HTMLParser().unescape(text)
print(clean_text)
```

This failed for me with Python2 (from HTMLParser import HTMLParser), almost certainly due to the input file not being plain ASCII, but including unicode characters.

When you have converted them, your favourite text editor will provide much cleaner formatting.

Java

Converting Java Examples to Python Examples - a Lesson
LibreOffice Python UNO Cookbook
Introduction

Background

This cookbook is designed to provide quick and dirty solutions to everyday problems encountered when using Python with OpenOffice. Sometimes it will not only suggest a programming solution, but also suggest a means of achieving the same result using features built into LibreOffice.

As a cookbook, all the examples are designed to be self-contained. It is up to the reader to combine these solutions appropriately. Compared to a classic manual, you will get less insight into the deep workings of LibreOffice, or best practice, however, you should be able to get most things working much quicker.

Prerequisites

You should only use this document if you have already learnt how to use Python with LibreOffice. This is not a tutorial, and doesn't include information about how to set up LibreOffice.

Convention and Style

Throughout the recipes, the following variables will be used, and it will be assumed that the reader already knows how to initialise them:

<table>
<thead>
<tr>
<th>Variables by Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>document</td>
</tr>
<tr>
<td>cursor</td>
</tr>
</tbody>
</table>

Code Style / Formatting

If the code isn't nicely formatted at the time of writing, I apologise, and acknowledge the irony of writing a book about coding and style, which is poorly formatted.
Opening Documents

Open a Document So That You Can Use Python's 'with ... as ...' Statements

It's easy to leave open references to documents. Here we add a feature that allows using the python 'with ... as ....' statements. This means you never need to call .dispose() again.

This isn't necessarily the behaviour you want if experimenting in an interactive terminal, but it is what you should do when you are writing the final program, and you can use the same openDocument() function across situations.

Although there's quite a lot of code here, you can just save it in a separate file and import the openDocument() function.

```python
import uno
import os
from com.sun.star.beans import PropertyValue
from unohelper import systemPathToFileUrl

#Create a wrapper for the document object so it works with 'with' statement
#Most calls to the object it just forwards to the object it wraps.
class DisposeToExitWrapper(object):
    '''
    This means that you can use with ... as ...
    '''
    def __init__(self, obj):
        self._wrapped_obj = obj

    def __getattr__(self, attr):
        if attr in self.__dict__:
            return getattr(self, attr)
        else:
            return getattr(self._wrapped_obj, attr)

    def __exit__(self, type, value, tb):
        self._wrapped_obj.dispose()

    def __enter__(self):
        return self._wrapped_obj

def dictToProperties(dictionary):
    '''
    Utility to convert a dictionary to properties
    '''
    props = []
    for key in dictionary:
        prop = PropertyValue()
        prop.Name = key
        prop.Value = dictionary[key]
        props.append(prop)
    return tuple(props)
```
```python
def convertPathToOOPath(document_path):
    # This adds e.g. file:/// to the start of the path, and makes it
    # absolute (has to be done)
    return systemPathToFileUrl(os.path.abspath(document_path))

# Create a function to quickly and easily open the document
def openDocument(document_path):
    '''document_path can be relative'''
    # Connect to OO
    local = uno.getComponentContext()  #
    resolver = local.ServiceManager.createInstanceWithContext("com.sun.star.bridge.UnoUrlResolver", local)
    context = resolver.resolve("uno:socket,host=localhost,port=2002;urp;StarOffice.ComponentContext")

    # Load services
    desktop = context.ServiceManager.createInstanceWithContext("com.sun.star.frame.Desktop", context)
    document_path_full = convertPathToOOPath(document_path)
    document = desktop.loadComponentFromURL(document_path_full, "_blank", 0, dictToProperties({"Hidden": True}))  # "Hidden" -> Doesn't show up in the GUI
    return DisposeToExitWrapper(document)

# open a document and use
document_path = './mydoc.odt'
with openDocument as safe_document:
    print('Document title is ' + document.Title)

# Now the document has been properly disposed of.
```

This version isn't as complete as it could be - if you fail to start the OpenOffice server, then it will fail to run.
Document Publishing

Document Publishing Overview and Use Cases

Word processors were originally designed largely to allow you to create printed documents more effectively, and this is still where their strength lies. There are better tools for collaborative working, such as wikis and forums, but when you want to quickly and easily create documents to be treated as printed documents, word processors are still the winners.

Therefore, it is a base assumption that you will want to convert your nice ODT document that you have created in Writer into another format, typically PDF for emailing or printing to paper for that retro-cool physical output. OpenOffice has a lot of abilities to convert to and from different formats, but the quality of this conversion process can be variable - even for tasks that you would expect to work flawlessly in a mature office suite, such as printing. My preferred technique is to always create a PDF - everyone can read them, and they can be printed reliably by you or an external printer. PDFs can be used as a form of print preview.

One use case that can be particularly useful is to automatically convert all your documents to PDF. This gives you an easy-to-browse (even on tablets) copy of your documents, which you won't accidentally edit when all you wanted to do was read them.

Documents are not necessarily static and final, and providing a mechanism for feedback can be important. Typically we're not talking about fully collaborative documents that would be suitable for Wikis, but more curated documents where the author is in control of the whole process, but wants to enable people to leave targeted feedback about particular areas of the document. In the case of this book, a typical use case would be to allow people to make comments that code examples don't work in certain situations.

Converting To PDF

Save the current document as a PDF file:

```python
from com.sun.star.beans import PropertyValue
property = (
    PropertyValue( "FilterName" , 0, "writer_pdf_Export" , 0 ),
)
document.storeToURL("file:///home/my_username/output.pdf", property)
```

Remember to use the full file url, not just relative paths.

Saving a Document to HTML

This is very similar to the method to save in a number of formats, such as PDF as well as HTML.

There are different options for even exporting as HTML. In brief:

- "HTML (StarWriter)" - seems to format elements such as bullets well, but doesn't restrict the width of the text on the screen, so making it fill the browser window (a bit awkward with modern screens). The underlying HTML seemed fairly neat, which could make it easier to manually manipulate the HTML or CSS style sheets.

- "XHTML Writer File" - restricts the width of the text in the browser, to make it more like
reading a document, but some of the formatting for things like bullets seemed off in my experiment. The underlying HTML seemed a lot more complicated than the "HTML (StarWriter)".

- "HTML" - seems similar to "HTML (StarWriter)"

There is a dated, but maybe useful list of output formats at

http://wiki.openoffice.org/wiki/Framework/Article/Filter/FilterList_OOo_2_1

```python
from com.sun.star.beans import PropertyValue
prop = (
    PropertyValue( "FilterName" , 0, "XHTML Writer File" , 0 ), #zeroes
    should be None or False??????
    #PropertyValue( "FilterName" , 0, "HTML (StarWriter)" , 0 ), #zeroes
    should be None or False??????
    PropertyValue( "FilterName" , 0, "HTML" , 0 ), #zeroes should be None
    or False??????
)
document.storeToURL(HTML_OUTPUT_FILE_PATH,prop) #fully qualified with
file:///
```
Working With Styles

Set Styles Across Several Documents

Here we use one document as a template, and use it to set the styles across a number of documents. If the style doesn't exist, then it will copy it across, if it does, it will (optionally) override the settings. This can be very useful when you change the default style that you use.

The code is in part based on the Template Changer extension, which you can download to LibreOffice (untested), and which is coded in basic. By the end, the code is somewhat different. What I hope is that it means that it's a reliable system.

http://extensions.libreoffice.org/extension-center/template-changer

There are a few things of note. Firstly, it doesn't iterate over all the style properties to copy them across, but uses a built-in function to do all the heavy lifting. Secondly, it uses this slightly funny line

uno.invoke( to_rules, "replaceByIndex" , (i,

This is a hack to get around limitations of the type system for OpenOffice 3.x. Essentially, what it is doing is converting rules[i], a list of PropertyValues into a strongly typed list, and then passing it to the to_rules.replaceByIndex function. If it worked properly, the code would be:

#Code if there weren't type conversion issues that need a hack
to_rules.replaceByIndex(i, rules[i])

For the interested, there is more on this problem at

- https://issues.apache.org/ooo/show_bug.cgi?id=12504
- http://www.openoffice.org/udk/python/python-bridge.html

I've complicated this example a little by including folder walking code to help get a list of files automatically. It also sets both the standard styles (e.g. Paragraph), and the Chapter Numbering Rules, in line with the Template Changer plugin.

I use my own openDocument() function here to open the documents when given a file path. I haven't listed it in this example to save space, but it returns an object that you can use with “with... as...” and not need to call document.Dispose().

During testing, this lead to a number of crashes of the entire of LibreOffice, for an unknown reason. It is still quicker than doing manually however. Adding a delay between documents might help.

I also don't know if the linking of the file to the template is working as it should. I couldn't find its effect, but I've left it in for reference.

#----------Settings--------------
TEMPLATE_FILE = "/home/jamie/TemplateFile.odt"
FILES_TO_APPLY_TEMPLATE_TO = None #List of paths if want to use, else None.
FOLDER_TO_APPLY_TEMPLATE_TO = "/home/jamie/ToApplyTo/" #path, or None if you want to use an explicit list
APPLY_TO_FOLDER_RECURSIVELY = True #i.e. should it apply to documents within subfolders of the given folder?
OVERWRITE_EXISTING_STYLES = True #e.g. if Heading 1 exists, set its properties to those of the template.
LINK_TEMPLATE_WITH_FILES = False #May not work properly.

#----------Imports-------------------
import logging
import os
import uno

#my standard function that takes a file path and returns a document object.
from CoreDocumentProcessingImports import openDocument

#----------Code-----------------
def dictToProperties(dictionary): #normally I'd just import this
    """
    Utility to convert a dictionary to properties
    """
    props = []
    for key in dictionary:
        prop = PropertyValue()
        prop.Name = key
        prop.Value = dictionary[key]
        props.append(prop)
    return tuple(props)

def getAllODTFilesInDirectory(directory, recursively):
    """
    Returns the files fully qualified
    """
    file_extensions = [".odt"]
    all_files = []
    for root, sub_folders, files in os.walk(directory):
        all_files+=[os.path.join(root, z) for z in files if any([z.endswith(y) for y in file_extensions])]
        if not recursively:
            break
    return all_files

def setOutlineNumbering(apply_to_document, template_document):
    """
    Copies the outline numbering rules across.
    Method from the TemplateChanger.xba
    """
    logging.info("Copying across the chapter numbering rules")
    #Get the rules
    logging.debug("Extracting the rules")
in_rules = template_document.getChapterNumberingRules()
num_rules = in_rules.getCount()
rules = []
for i in range(num_rules):
    rules.append( in_rules.getByIndex(i))

to_rules = apply_to_document.getChapterNumberingRules()

    #Set the rules
logging.debug("Setting the rules")

    #TODO: This method of using the index is based on the TemplateChanger example.
    #I haven't verified it is designed to work like this (i.e. they're always in the same
    #order.
    for i in range(num_rules):
        uno.invoke( to_rules, "replaceByIndex" , (i,
uno.Any("[]com.sun.star.beans.PropertyValue", rules[i])))

logging.info("Chapter numbering rules copied")

def setAllStylesFromTemplateBuiltInMethod(apply_to_document, template_document, override_existing_styles, link_template_with_file = False):
    """
    Method has been taken from the TemplateChanger LibreOffice extension.
    if link_template_with_file, then it will create a permanent reference and AFAIK it will update the main
document when the template is updated.
    """
    propertiesDict = {"OverwriteStyles":override_existing_styles}
properties = dictToProperties(propertiesDict)
template_url = template_document.URL
apply_to_document.StyleFamilies.loadStylesFromURL( template_url, properties)

setOutlineNumbering(apply_to_document, template_document)

if link_template_with_file:
    if template_document.Title != ":
        apply_to_document.DocumentProperties.TemplateName = template_document.Title
    else:
        apply_to_document.DocumentProperties.TemplateName = template_document.URL.rsplit(os.sep, 1)[-1].split(".")[0]


doc_settings =
apply_to_document.createInstance("com.sun.star.document.Settings")
doc_settings.UpdateFromTemplate = True
#---------Running--------------

if __name__ == "__main__":

    logging.getLogger("").setLevel("DEBUG")

    logging.info("Using the template "+TEMPLATE_FILE)

    logging.info("Getting files to update template for")

    # Sense check the settings
    if (FILES_TO_APPLY_TEMPLATE_TO and FOLDER_TO_APPLY_TEMPLATE_TO) or not (FILES_TO_APPLY_TEMPLATE_TO or FOLDER_TO_APPLY_TEMPLATE_TO):
        message = "You must either set a list of files to set the styles in, or set a folder to set all files within"
        logging.error(message)
        raise Exception(message)

    files_to_apply_to = FILES_TO_APPLY_TEMPLATE_TO if FILES_TO_APPLY TEMPLATE TO else getAllODTFilesInDirectory(FOLDER TO APPLY TEMPLATE TO, APPLY TO FOLDER RECURSIVELY)

    logging.info("Will set for "+str(len(files_to_apply_to))" files")

    with openDocument(TEMPLATE_FILE) as template_document:
        for i, file_path in enumerate(files_to_apply_to):
            logging.info("Applying to "+str(i+1)+"/"+str(len(files_to_apply_to))+")"+file_path)
            with openDocument(file_path) as apply_to_document:

                setAllStylesFromTemplateBuiltInMethod(apply_to_document, template_document, override_existing_styles = OVERRIDE_EXISTING_STYLES, link_template_with_file = LINK TEMPLATE WITH FILES)

                logging.debug("Saving")
                apply_to_document.store()

    logging.info("finished")
Fields and References

Inserting Page Number Field

```python
from com.sun.star.style.NumberingType import ARABIC

# Create the field
pageNumberField = document.createInstance("com.sun.star.text.TextField.PageNumber")
pageNumberField.setPropertyValue("NumberingType", ARABIC) # use numbers

# Insert into the document near the current cursor.
document.Text.insertTextContent(cursor, pageNumberField, False)
```

Inserting Page Count Field

```python
from com.sun.star.style.NumberingType import ARABIC

# Create the field
pageCountField = document.createInstance("com.sun.star.text.TextField.PageCount")
pageCountField.setPropertyValue("NumberingType", ARABIC) # use numbers

# Insert into the document near the current cursor.
document.Text.insertTextContent(cursor, pageCountField, False)
```
Creating a Table

Insert / add a table into a writer document, and fill the table cells with text.

```python
import string

# Create a table with 6 rows and 2 columns
num_rows = 6
num_columns = 2
new_table = document.createInstance("com.sun.star.text.TextTable")
new_table.initialize(num_rows, num_columns)
# Actually insert the created table into the document.
document.Text.insertTextContent(cursor, new_table, 0)

# Insert some text into the table.
new_table.getCellByName("A1").setString("This is first column and first row")
new_table.getCellByName("B1").setString("Second column, first row")

# Now let's fill the rest of the cells with their names.
for col in string.uppercase[:num_columns]:  # "A", "B" etc works up to 26 columns
    for row in [str(z) for z in range(1,7)]:  # "1", "2" etc
        cell_name = col+row
        if cell_name not in ["A1", "B1"]:  # Exclude the cells we have set already
            new_table.getCellByName(cell_name).setString(cell_name)
```

Get Existing Tables In a Document and Edit Them

You can iterate over all the tables in the document, and modify their contents, but you need to be aware that the index of the table is not generally the first table in the document (maybe the first table to be created?).

```python
# Get an object to access all the tables
all_tables = document.getTextTables()
print("There are %i tables" % all_tables.getCount())

# Get the first table - useful if you know there is only one table
first_table = all_tables.getByIndex(0)
print("The first table by index is %s" % first_table.getName())

# Get a table by name. You can set this by right clicking on the table and setting its properties, but generally people probably don't bother
first_table = all_tables.getByName(table_name)
print("The first table by name is %s" % first_table.getName())

# Iterate over all the tables, printing their names
for i in range(all_tables.getCount()):
    table = all_tables.getByIndex(i)
    print("Table %i is called %s" % (i, table.getName()))
```
Get a Cell by Name e.g. A1

```python
# Set the text in the first cell of the first table
table = document.getTextTables().getByIndex(0)
table.getCellByName("A1").setString("Text I set")
```

```python
# Get the first table in the document
table = document.getTextTables().getByIndex(0)

# Get the cell in the first column and second row and print the contents
cell_reference = 'A2'
cell = table.getCellByName(cell_reference)
print(cell.getString())
```

Get a Cell by Position in the Table

```python
# Get the first table in the document
table = document.getTextTables().getByIndex(0)

# Get the cell in the first column and second row and print the contents
column_index = 0
row_index = 1
cell = table.getCellByPosition(column_index, row_index)
print(cell.getString())
```

Get the Text in a Table Cell

```python
# Get the first table in the document
table = document.getTextTables().getByIndex(0)

cell = table.getCellByName('A1')
print( cell.getString())
```

Change the Text in a Table Cell

```python
# Get the first table in the document
table = document.getTextTables().getByIndex(0)

cell = table.getCellByName('A1')
cell.setString('new cell text')
```

Iterate over All Cells in a Table

**Warning:** iterating over a lot of cells can be very slow.

This method iterates over all the indexes and checks that the cell exists before printing the cell contents.

```python
# Get the first table in the document
table = document.getTextTables().getByIndex(0)

num_rows = table.getRows().getCount()
num_cols = table.getColumns().getCount()

# Iterate over the rows, then the columns
for row_num in range(num_rows):
    for col_num in range(num_cols):
        try:
            cell = table.getCellByPosition(col_num, row_num)
        ```
The following method is good if you don't really care about the order or contents of the cells. It simply gets all the names of the cells and iterates over them. It is quicker as you know that the cell will exist in advance.

```python
#Get the first table in the document
table = document.getTextTables().getByIndex(0)

for cell_name in table.getCellNames():
    cell = table.getCellByName(cell_name)
    print('Cell %s = %s' % (cell_name, cell.getString()))
```

### Deleting Rows of a Table

```python
#Get the first table in the document
table = document.getTextTables().getByIndex(0)

#delete the first two rows of the table
rows = table.getRows()
start_row_index = 0  # first row
num_rows_to_delete = 2
rows.removeByIndex(start_row_index, num_rows_to_delete)
```

### Adding / Inserting Rows and Columns to a Table

First, let's add two rows to the table

```python
#Get the first table in the document
table = document.getTextTables().getByIndex(0)

#insert 2 new rows at the end
rows = table.getRows()
num_rows_to_insert = 2
index_of_first_new_row = rows.getCount()
rows.insertByIndex(index_of_first_new_row, num_rows_to_insert)

#Fill the rows with some text
num_cols = table.getColumns().getCount()
for row_num in range(index_of_first_new_row, index_of_first_new_row + num_rows_to_insert):
    for col_num in range(num_cols):
        cell = table.getCellByPosition(col_num, row_num)
        cell.setString('New Cell (%s, %s)' % (col_num, row_num))
```

This takes advantage of the com.sun.star.table.XTableRows interface at See [http://www.openoffice.org/api/docs/common/ref/com/sun/star/table/XTableRows.html](http://www.openoffice.org/api/docs/common/ref/com/sun/star/table/XTableRows.html).

Almost identical is to use the com.sun.star.table.XTableColumns interface to add columns, available at [http://www.openoffice.org/api/docs/common/ref/com/sun/star/table/XTableColumns.html](http://www.openoffice.org/api/docs/common/ref/com/sun/star/table/XTableColumns.html), however, in this case you need to be careful as the index of the last column within a row will vary according to whether cells have been merged across the row - you can't simply use a fixed index, you must work out what the last column's index is.

```python
#Get the first table in the document
table = document.getTextTables().getByIndex(0)

#insert one new column at the end
columns = table.getColumns()
num_columns_to_insert = 1
index_of_first_new_column = columns.getCount()
columns.insertByIndex(index_of_first_new_column, num_columns_to_insert)

#Fill the cells in the new column with some text
num_cols = table.getColumns().getCount()
for col_num in range(index_of_first_new_column, index_of_first_new_column + num_columns_to_insert):
    for row_num in range(num_rows):
        cell = table.getCellByPosition(col_num, row_num)
        cell.setString('New Cell (%s, %s)' % (col_num, row_num))

#THIS FAILS IF THERE VARIABLE NUMBERS OF COLUMNS

Change the Background Colour of a Whole Table

When you set the colour for the table, be aware that if a cell has a background colour set, it will override the colour for the whole table. Beyond that, the text within the cell may have a background colour associated with it.

There is a difference between the default colour, which is selected using -1, and making the table background transparent.

#Get the first table in the document
table = document.getTextTables().getByIndex(0)

#Print the colour of the table background
#If -1, then it is set to the default, otherwise it is an RGB value expressed as a Long
existing_colour = table.getPropertyValue("BackColor")

rgb_value = 0x0077FF #No red = 00, green = 77, full blue = FF
table.setPropertyValue('BackColor', rgb_value)
#table.setPropertyValue('BackTransparent', True) #have no background color

Change the Background Colour of a Whole Table, With All Its Cells

To ensure that the background colour for all of the table, iterate over the table cells setting the background colour to -1 (default) as well as setting the background colour for the table. This is a lot slower than just setting the background for the whole table.

This won't however override the background colour to the text that is within the cell.

#get the first table in the document
table = document.getTextTables().getByIndex(0)

#Set the table background colour
rgb_value = 0x0077FF #No red = 00, green = 77, full blue = FF
table.set PropertyValue('BackColor', rgb_value)

#Iterate over all the cells, setting:
#-background color to -1 (default)
#-transparency on (so that the colour doesn't matter anyway)
for cell_name in table.getCellNames():
    cell = table.getCellByName(cell_name)
    cell.setPropertyValue('BackColor', -1)
    cell.setPropertyValue('BackTransparent', True)
Change the Background Colour of a Table Cell

This is very similar to changing the background colour of the whole table, but it overrides the value for the table.

```python
#Get the first table in the document
table = document.getTextTables().getByIndex(0)

#Print the colour of the table background
#If -1, then it is set to the default, otherwise it is an RGB value
#expressed as a Long
existing_colour = table.getPropertyValue("BackColor")

rgb_value = 0x0077FF #No red = 00, green = 77, full blue = FF
table.setPropertyValue("BackColor", rgb_value)
```
Writing Code

Introduction

When writing code in a document, you should separate it out from the body of the text by using a different style. This allows you to search for the code in a document and then process it more intelligently later. One easy way of doing this is to use a special code paragraph style, however, you may choose to create a special frame, and maybe also a special character style.

The sort of tasks that you might want to do include extracting the code samples and testing them, to ensure that your reader can run them, or re-formatting the code to make it easier to read e.g. with special code highlighting.
Adding a Header

The header is controlled by the PageStyle in use for a given page. Here we will set the header to be on for the “Standard” page style, which is the default page style generally used.

```python
#Get the page style object
standard_style =
document.getStyleFamilies().getByName("PageStyles").getByName("Standard")

#Turn on the header
standard_style.setPropertyValue("HeaderIsOn", True)

#Set some text in the header
header_text = standard_style.getPropertyValue("HeaderText")
header_text.setString("this is header text")
```

Adding a Footer

```python
#Get the page style object
standard_style =
document.getStyleFamilies().getByName("PageStyles").getByName("Standard")

#Turn on the footer
standard_style.setPropertyValue("FooterIsOn", True)

#Set some text in the footer
footer_text = standard_style.getPropertyValue("FooterText")
footer_text.setString("this is footer text")
```

Adding a Header / Footer with Page Numbers

Here we will only deal with the header, but modifying to work with the footer should be as simple as replacing “Header” with “Footer” in the code. To do this, we will also create a cursor object to help edit text within the footer.

```python
#Get the page style object
standard_style =
document.getStyleFamilies().getByName("PageStyles").getByName("Standard")

#Turn on the footer
standard_style.setPropertyValue(’FooterIsOn’, True)

from com.sun.star.style.NumberingType import ARABIC

#Create the page number field
pageNumberField =
document.createInstance(’com.sun.star.text.TextField.PageNumber’)
pageNumberField.setPropertyValue(’NumberingType’, ARABIC) #use numbers

#Create the page count field
pageCountField =
document.createInstance("com.sun.star.text.TextField.PageCount")"
getPageCountField.setPropertyValue('NumberingType', ARABIC) # use numbers

# Insert into the document near the current cursor.
# document.Text.insertTextContent(cursor, pageCountField, False)
# document.Text.insertTextContent(cursor, pageNumberField, False)

# Get the text object for the footer
footer_text = standard_style.getPropertyValue('FooterText')

# Clear any existing footer content
footer_text.setString('')

# Create a cursor in the footer to manipulate its contents
footer_cursor = footer_text.createTextCursor()

footer_text.insertString(footer_cursor, '\t', False) # Tab to put it in the middle
footer_text.insertString(footer_cursor, 'Page Number ', False)
footer_text.insertTextContent(footer_cursor, pageNumberField, False)
footer_text.insertString(footer_cursor, ' / ', False)
footer_text.insertTextContent(footer_cursor, pageCountField, False)
Getting the Outline Level of a Heading Style

The outline level is used to create a hierarchy, for instance to create a table of contents at the start of the document, or for the Navigator panel in the OpenOffice user interface. The outline level is normally 1 for Heading 1, 2 for Heading 2 etc. However, you need to be careful that the default level is 0, used by for example the Text Body. Therefore you can’t simply order by this index.

```python
heading_style_name = 'Heading 2'
heading_style =
document.getStyleFamilies().getByName('ParagraphStyles').getByName(heading_style_name)
heading_level = heading_style.getPropertyValue('OutlineLevel')
print(heading_style_name + ' has level ' + str(heading_level))
```

This method is fairly slow, you so might want to cache the levels into a dictionary to make it faster if calling repeatedly (and not changing the OutlineLevel for the paragraph styles).

```python
#Make dictionary of all paragraph styles to outline_level for a cache
styles = document.getStyleFamilies().getByName('ParagraphStyles')
styles_level_dictionary = {}
for i in range(styles.getCount()):
    style = styles.getByIndex(i)
    styles_level_dictionary[style.Name] = style.getPropertyValue('OutlineLevel')

#Look up an outline level
heading_style_name = 'Heading 2'
heading_level = styles_level_dictionary[heading_style_name]
print(heading_style_name + ' has level ' + str(heading_level))
```

It is quite slow to create this cache as well (it takes a few seconds), so you may be better off restricting to known names, or sets of names. It is quick to get the styles.ElementNames property, which lists all the style names.

```python
#Make dictionary of paragraph styles to outline_level for a cache
#But only do for Headings
styles = document.getStyleFamilies().getByName('ParagraphStyles')
styles_level_dictionary = {}
for style_name in styles.ElementNames:
    if style_name.startswith('Heading'):
        style = styles.getByName(style_name)
        styles_level_dictionary[style.Name] = style.getPropertyValue('OutlineLevel')

#Look up an outline level
heading_style_name = 'Heading 2'
heading_level = styles_level_dictionary[heading_style_name]
print(heading_style_name + ' has level ' + str(heading_level))
```

A more advanced system might look up a style that it doesn’t already have in the cache, and add it to the cache i.e. lazy initialisation.

LibreOffice Python UNO Cookbook | 50
Listing All Headings in Document Order

This method uses the text cursor to iterate over all the paragraphs, and detects whether or not they are in a list of known heading styles.

```python
import string

# Define a function to get all the heading in text order
def getAllHeadingsInTextOrder():
    heading_styles = ['Heading 1', 'Heading 2']
    headings_in_order = [] # list of heading_text, heading style
    cursor.gotoStart(False)
    while True:
        current_heading_name = cursor.getPropertyValue('ParaStyleName')
        if current_heading_name in heading_styles:
            # get the text used in the heading
            cursor.gotoEndOfParagraph(True) # select the text
            heading_text = cursor.getString()
            headings_in_order.append((heading_text, current_heading_name))
        if not cursor.gotoNextParagraph(False): # have reached the end
            break
    return headings_in_order

# Create a function to give the level of a heading
# This level is used for creating hierarchical tables of contents
def getOutlineLevel(heading_style_name):
    '''This will return 0 for default level, 1 for Heading 1 etc.''
    heading_style = document.getStyleFamilies().getByName('ParagraphStyles').getByName(heading_style_name)
    outline_level = heading_style.getPropertyValue('OutlineLevel')
    return outline_level

# Actually print out the headings, but with some pretty indenting according to level.
headings = getAllHeadingsInTextOrder()
print('_'*4+'Headings for '+document.getTitle()+'_'*4)
print('')
for heading_text, heading_style_name in headings:
    level = getOutlineLevel(heading_style_name)
    print((level-1)*'\t'+heading_text+ '  '+'('+heading_style_name+')')

Getting the OutlineLevel can be slow using this method and caching the level would be a good idea if you have a long document in particular.

Demoting All Headings To a Lower Level

It often happens that you start with a document and think that you have everything in the right hierarchy of headings, but then decide that it's not clear and decide to add in a level above your headings to represent chapter-like divisions. This means demoting the level of the headings so that e.g. Heading 1 goes to Heading 2.

```python
heading_styles_in_order = ['Heading '+str(z) for z in range(9)]

# Start from the beginning of the document
cursor.gotoStart(False)
```
while True:
    current_style_name = cursor.getPropertyValue('ParaStyleName')

    if current_style_name in heading_styles_in_order:
        index = heading_styles_in_order.index(current_style_name)
        change_to_style = heading_styles_in_order[index + 1]

        # select the whole paragraph
        cursor.gotoEndOfParagraph(True)
        cursor.setPropertyValue('ParaStyleName', change_to_style)

    if not cursor.gotoNextParagraph(False):  # move to next paragraph
        break
Managing Changing Documents

Removing Notes from Output

There are several ways that you can add notes to a document, you can add a comment that is associated with the text (Insert -> Comment; or CTRL + Alt + C). These show up in the side margins.

The way that I favour is to use a special style for the notes, which has a different background colour. I then put the notes in the main document, which save screen real estate and makes them easier to print. However, they need to be removed for distributing the document.

So the method here is to remove text that is of a certain style.

```python
#Remove notes based on them being a particular style.
style_names_to_remove = ['Notes_Body']  # Names of styles of notes
cursor.gotoStart(False)
while cursor.gotoNextParagraph(False):
    style_name = cursor.getPropertyValue("ParaStyleName")
    if style_name in style_names_to_remove:
        cursor.goLeft(1, False)  # Do this to remove the newline character between paragraphs.
        # Select all the contents of the paragraph
        cursor.gotoNextParagraph(True)
        cursor.gotoEndOfParagraph(True)

        cursor.setString(""")  # Delete the text contents.
```
Extracting the Contents of an ODT File

.odt files are a special form of zip file. You need to be careful when compressing them, but when uncompressing them, it is as simple as using a regular zip tool.

Depending on your operating system and zip tool of choice, you may find it easier to change the extension of the ODT file to .zip and then extract it.

After extraction, you should have a folder with a number of .xml files and some sub-folders. The .xml files will include content.xml, style.xml, metal.xml, settings.xml. These files can be opened with a regular text editor. In the sub-folders, you may have items such as images that were embedded in the file.

Re-compressing an ODT File

ODT files can be uncompressed as if they were simple zip files, but in practice they are slightly more complex. The mimetype file is stored first and uncompressed. This is to allow the operating system to detect the true type of file, without relying on the .odt file extension - it's a security feature as the file extension can easily be changed.

First step is to create the zip file with an uncompressed mimetype. From within the folder where content.xml is stored, run:

```
$zip -0 -X ../*.odt mimetype
```

The -0 means that it won't be compressed and the -X means that no extra file attributes will be stored. The resulting file will be stored in the folder one level above and called my-document.odt.

Then you need to add the rest of the files to the zip, but this time actually compressing them:

```
zip -r ../*.odt * -x mimetype
```

The -r is to recursively add files in sub-folders (i.e. compress all the contents of the folder. The -x is to exclude the mimetype file, as we have already added it.

The resulting file should now open with LibreOffice.

More details, and the source for this section at:

http://www.jejik.com/articles/2010/03/how_to_correctly_create_odf_documents_using_zip/

Debugging and Fixing Corrupted ODT / LibreOffice Writer Files

This can happen for a number of reasons such as bugs in the software used to open ODTs (LibreOffice, Abiword, Calligra Words, OpenOffice, MS Office etc); it could be due to an error when manually editing them; or it could be due to some form of filesystem corruption.

The first thing is to make a backup of the file that you are going to try to fix!

When you extract an ODT file (using a zip decompression tool - it's just a special .zip file with a different extension), the most likely source of problems is in the content.xml file. This represents
the main body and structure of the document, and you will be able to see the text in the file surrounded by various XML tags, like `<text:p text:style-name="P27" >`. This is just a standard XML file, and so problems with the XML structure can be debugged as with any standard XML file. If you struggle to extract the ODT, then the problem is with the zip file, and you may struggle to ever recover it. You need to search online for methods to recover corrupted zip files.

A typical error message when you open the ODT file in LibreOffice may be

```
Read-Error.
Format error discovered in the file in sub-document content.xml at 914,129(row,col)
```

This means that the problem is something to do with the content.xml file from the extracted ODT and LibreOffice's best guess is that it is most likely on line 914, and the 129th character on that line. It's important to note that this is simply a best guess, and often the problem will be at some point above that line (the files are read and interpreted down the page). It also doesn't give many hints as to what the problem might be.

On Linux you can use the command line tool xmllint to try to get better debugging information:

```
$xmllint -noout content.xml
```

This should provide some more information about where the problem is, but remember that it is only guessing what the problem is and often it will be misleading.

Alternatives to xmllint include web browsers, such as Chrome, which can show XML files (you may need to do 'show source' on the page), and which may give useful debugging information as well. There is also XML Copy Editor, http://xml-copy-editor.sourceforge.net/, which is available on Windows and Linux which not only formats the XML nicely, but it will also red underline the errors, and you can make changes in-situ.

The next stage is to use your favourite text editor, and go to the lines suggested by xmllint. Typical problems might be tags that haven't been opened / closed properly, quotes that haven't been opened and closed properly, and duplicated data. Problems can span several lines, and you will have to work through it. XML is read from the top down, so any problem should have started in the line highlighted or in a line above. In my experience there is no silver bullet for solving these problems. If your XML file is all on one line, you might want to use the Linux command line tool xmlindent to format it better, or your text editor may do this.

A crude, but effective method of narrowing down the problem is to take out a large chunk of text within one set of tags, and see if it produces a valid XML file. Then gradually reduce the size of the chunk taken out until you narrow down the error to a small section. Hopefully you will be able to spot the error, but failing that you can at least remove the problematic code and only lose a small amount of information. You have to do this while understanding the XML structure however, or you will just create new problems.

If you have a backup file, which does open correctly, then another strategy can be to compare the two files content.xml files. Most programmers' text editors will have a tool for comparing text files and highlighting the difference. Often this process is called “diff” (as it’s showing the difference between two files).

Once you have created XML content that passes the xmllint checks (or loads in Chrome without problems), you should re-form the ODT file and try opening in LibreOffice. On Linux:

```
$cd thefolderwithcontent.xml
$zip -o -X ../my-document.odt mimetype
$zip -r ../my-document.odt * -x mimetype
```

The resulting odt file will be called my-document.odt and be in the folder above the content.xml
file.

Given all OpenDocument format files (.odt, .ods, .odp etc) have broadly the same format of being zipped files of content in XML and some sub-folders, you can use the same methods to fix corrupted spreadsheet and presentation files as well.

More help can be found in the forum posts at http://forum.openoffice.org/en/forum/viewtopic.php?f=7&t=1532, with a particular acknowledgement to user Acknak.
Tricks to Help Programming

Use iPython Console

If you hit TAB in the iPython console, it will (often) successfully list the functions that can be run on an object, when doing help(object) doesn’t show anything useful.

List the Interfaces a Libreoffice Object Implements Nicely So You Can Look Up Its Functions

This may well be a crude hack, but it works. In our example, we will print a list of the interfaces implemented by the document object. These interfaces can be looked up online in the Java API documentation.

This is useful as currently you can’t do help(libreoffice_object) and get a useful output.

```python
# Make a function to print just the interfaces of any object
def printInterfaces(my_object):
    text = str(my_object)
    interfaces_block = [z for z in text.split(' ') if z.startswith('supportedInterfaces=')][0]
    interface_names = interfaces_block[interfaces_block.find('{')+1:interfaces_block.find('}')].split(',
    for interface_name in interface_names:
        print(interface_name)

# actually print the interfaces the document object implements
print('The interfaces implemented by the document object are:')
printInterfaces(document)
```

This produces a list like

```
The interfaces implemented by the document object are:
com.sun.star.container.XChild
com.sun.star.document.XDocumentInfoSupplier
com.sun.star.document.XDocumentPropertiesSupplier
com.sun.star.rdf.XDocumentMetadataAccess
com.sun.star.document.XDocumentRecovery
com.sun.star.document.XUndoManagerSupplier
....
```

The next version removes the fully qualified name and the “X” character at the start of the interface, then it sorts them alphabetically. This IMHO makes it a lot easier to skim through.

```python
# Make a function to print just the interfaces of any object
def printInterfacesClear(my_object):
    text = str(my_object)
    interfaces_block = [z for z in text.split(' ') if z.startswith('supportedInterfaces=')][0]
    interface_names = []
    for long_name in interfaces_block[interfaces_block.find('{')+1:interfaces_block.find('}')].split(',
        interface_name = long_name[long_name.rfind('.')+1:]
        if interface_name[0]=='X':
            interface_names.append(interface_name[1:]).lower()
    interface_names.sort()
    print('The interfaces implemented by the document object are:
    print('
    '.join(interface_names))
```

```python
# actually print the interfaces the document object implements
print('The interfaces implemented by the document object are:
printInterfacesClear(document)
```

This produces a list like

```
The interfaces implemented by the document object are:
Child
DocumentInfoSupplier
DocumentPropertiesSupplier
DocumentMetadataAccess
DocumentRecovery
UndoManagerSupplier
....
```

LibreOffice Python UNO Cookbook | 57
interface_name = interface_name[1:]
interface_names.append(interface_name)
interface_names.sort() #Sort all the names alphabetically
for interface_name in interface_names:
    if interface_name[0]=='X':
        print interface_name[1:]
    else:
        print interface_name

#actually print the interfaces the document object implements
print('The interfaces implemented by the document object are:')
printInterfacesClear(document)

It is a small modification of this technique to also print out the services it exposes - just replace the “SupportedInterfaces” with “SupportedServices”.

**List the Interfaces an Object Implements with Links To the Documentation**

This works well if you are coding in the console and the console highlights links that you can click on to open in the browser (typically using CTRL+LClick).

This version gives you the option of using the OpenOffice or LibreOffice documentation, which are similar, but one may be of personal preference. For example, the interface com.sun.star.sheet.XCellRangeData is available at both:


So the implementation using the OpenOffice.org link is:

```python
#Function to convert an interface name into a link to #the openoffice documentation
def convertInterfaceToDocumentationUrl(interface_name):
    parts = interface_name.split('\\.\\')
    url = 'http://api.openoffice.org/docs/common/ref/\\'.join(parts) + '\.html'
    url = 'http://www.openoffice.org/api/docs/common/ref/\\'.join(parts) + '\.html'
    return url

def printInterfacesWithLinks(my_object):
    text = str(my_object)
    interfaces_block = [z for z in text.split('\\n') if
        z.startswith('\\nSupportedInterfaces=')] [0]
    long_names = interfaces_block[interfaces_block.find('{') +1:interfaces_block.find('}')] .split('\\n')
    short_names = [] #interface names in easy-read format
    for long_name in long_names:
        interface_name = long_name [long_name .rfind('\\.\\')+1:]
        if interface_name[0]=='X':
            interface_name = interface_name[1:]
        short_names.append(interface_name)
    short_long_names = [(short_names[i], long_names[i]) for i in
        range(len(long_names))] #create a combined list
    short_long_names.sort(key=lambda x:x[0]) #Sort all the names
```
alphabetically by the short name

```python
longest_short_name_length = max([len(z[0]) for z in 
short_long_names])
for short_name, long_name in short_long_names:
    url = convertInterfaceToDocumentationUrl(long_name)
    print(short_name.ljust(longest_short_name_length+2)+'
'+url)
```

#actually print the interfaces the document object implements
print('The interfaces implemented by the document object are:')
printInterfacesWithLinks(document)

List the Properties of an Object (for `getPropertyValue`, `setPropertyValue`)

Objects that have `PropertyValues` that can be set, can be investigated in program. These can typically be set with `object.setPropertyValue(property_name, value)`.

```python
#Define a method to make a nice printed list of properties
def printObjectProperties(obj):
    #Get the properties
    properties = list(obj.getPropertySetInfo().getProperties())
    #Sort alphabetically by name
    properties.sort(key = lambda x:x.Name)
    longest_len = max([len(z.Name) for z in properties])
    for property in properties:
        print(property.Name.ljust(longest_len)+'  -  '+str(property.Type))

    #Print the properties of the document object
    printObjectProperties(document)
```

see [http://www.openoffice.org/api/docs/common/ref/com/sun/star/beans/Property.html](http://www.openoffice.org/api/docs/common/ref/com/sun/star/beans/Property.html) for more fields of the Property object.