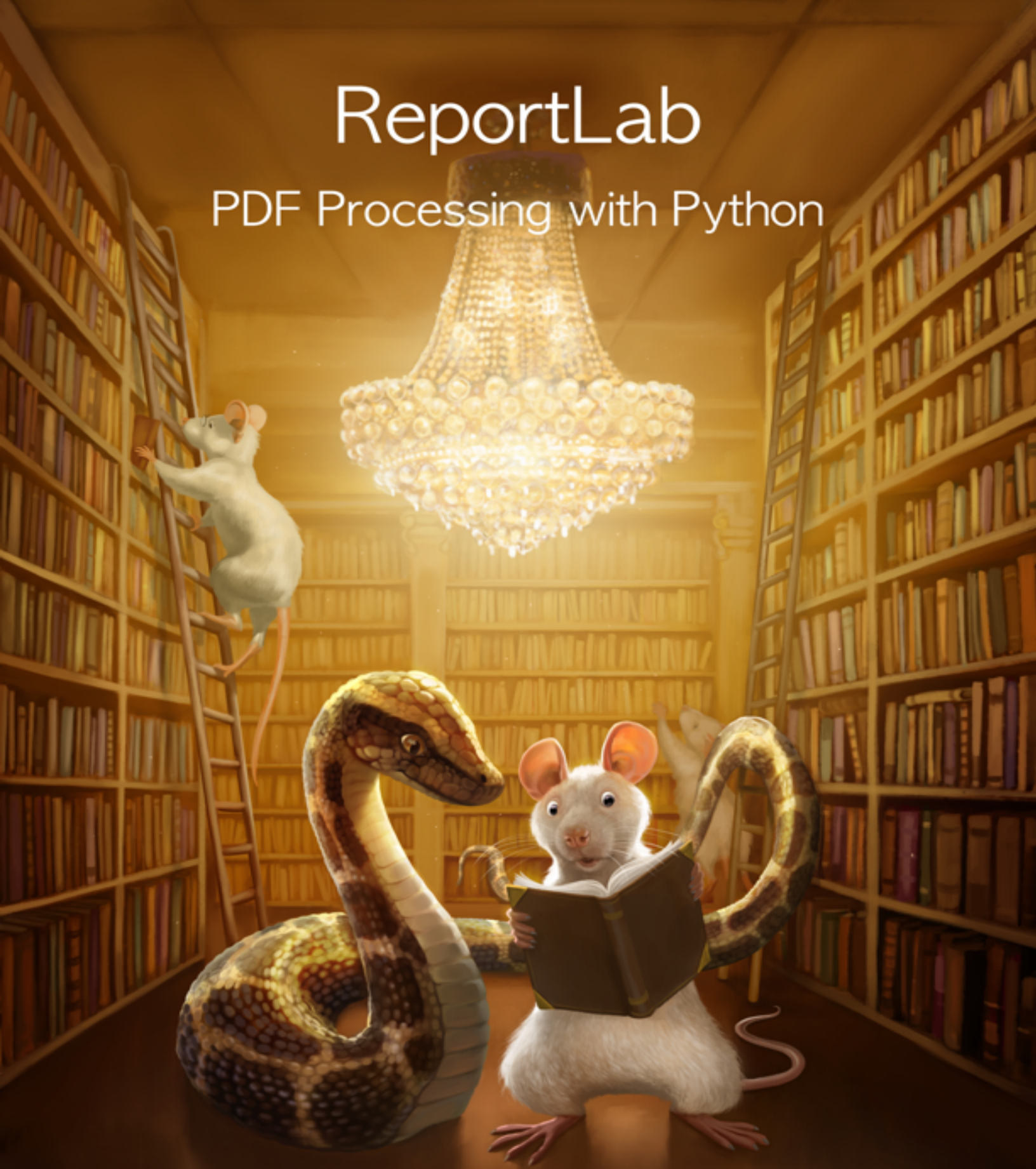
ReportLab

PDF Processing with Python



Michael Driscoll

ReportLab - PDF Processing with Python

Michael Driscoll

This book is for sale at http://leanpub.com/reportlab

This version was published on 2019-12-02

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction

The Reportlab PDF Toolkit started life in the year 2000 by a company called "Reportlab Inc.". Reportlab is now owned by "ReportLab Europe Ltd". They produce the open source version of Reportlab. The Reportlab toolkit is actually the foundation of their commercial product, **Report Markup Language** which is available in their **Reportlab PLUS** package. This book is focused on the open source version of Reportlab. The Reportlab PDF Toolkit allows you to create in Adobe's Portable Document Format (PDF) quickly and efficiently in the Python programming language. Reportlab is the defacto method of generating PDFs in Python. You can also use Reportlab to create charts and graphics in bimap and vector formats in addition to PDF. Reportlab is known for its ability to generate a PDF fast. In fact, Wikipedia chose Reportlab as their tool of choice for generating PDFs of their content. Anytime you click the "Download as PDF" link on the left side of a Wikipedia page, it uses Python and Reportlab to create the PDF!

In this book, you will learn how to use Reportlab to create PDFs too. This book will be split into three sections. We will be covering the following topics in the first section:

- The canvas
- Drawing
- Working with fonts
- PLATYPUS
- Paragraphs
- Tables
- Other Flowables
- Graphics
- and More!

In the second section, we will learn about data processing. The idea here is to take in several different data formats and turn them into PDFs. For example, it is quite common to receive data in XML or JSON. But learning how to take that information and turn it into a report is something that isn't covered very often. You will learn how to do that here. In the process we will discover how to make multipage documents with paragraphs and tables that flow across the pages correctly.

The last section of the book will cover some of the other libraries you might need when working with PDFs with Python. In this section we will learn about the following:

- PyPDF2
- pdfminer
- PyFPDF

About the Author

You may be wondering about who I am and why I might be knowledgeable enough about Python to write about it, so I thought I'd give you a little information about myself. I started programming in Python in the Spring of 2006 for a job. My first assignment was to port Windows login scripts from Kixtart to Python. My second project was to port VBA code (basically a GUI on top of Microsoft Office products) to Python, which is how I first got started in wxPython. I've been using Python ever since, doing a variation of backend programming and desktop front end user interfaces as well as automated tests.

I realized that one way for me to remember how to do certain things in Python was to write about them and that's how my Python blog came about: http://www.blog.pythonlibrary.org/. As I wrote, I would receive feedback from my readers and I ended up expanding the blog to include tips, tutorials, Python news, and Python book reviews. I work regularly with Packt Publishing as a technical reviewer, which means that I get to try to check for errors in the books before they're published. I also have written for the Developer Zone (DZone) and i-programmer websites as well as the Python Software Foundation. In November 2013, DZone published **The Essential Core Python Cheat Sheet** that I co-authored. I have also self-published the following books:

- **Python 101** - June 2014
- **Python 201: Intermediate Python** - Sept. 2016
- **wxPython Cookbook** - Dec. 2016

Conventions

As with most technical books, this one includes a few conventions that you need to be aware of. New topics and terminology will be in **bold**. You will also see some examples that look like the following:

```
>>> myString = "Welcome to Python!"
```

The >>> is a Python prompt symbol. You will see this in the Python **interpreter** and in **IDLE**. Other code examples will be shown in a similar manner, but without the >>>. Most of the book will be done creating examples in regular Python files, so you won't be seeing the Python prompt symbol all that often.

Setting up & Activating a Virtual Environment

If you don't want to add ReportLab into your system's Python installation, then you can use a virtual environment. In Python 2.x - 3.2, you would need to install a package called **virtualenv** to create a virtual environment for Python. The idea is that it will create a folder with a copy of Python and pip.

You activate the virtual environment, run the virtual pip and install whatever you need to. Python 3.3 added a module to Python called **venv** that does the same thing as the virtualenv package, for the most part.

Here are some links on how all that works:

- https://docs.python.org/3/library/venv.html (Python 3 only)
- https://pypi.python.org/pypi/virtualenv (Python 2 and 3)

When you are using a Python Virtual Environment, you will need to first activate it. Activation of a virtual environment is like starting a virtual machine up in VirtualBox or VMWare, except that in this case, it's just a Python Virtual Environment instead of an entire operating system.

Creating a virtual sandbox with the virtualenv package is quite easy. On Mac and Linux, all you need to do is the following in your terminal or command prompt:

```
virtualenv FOLDER_NAME
```

To activate a virtual environment on Linux or Mac, you just need to change directories to your newly created folder. Inside that folder should be another folder called **bin** along with a few other folders and a file or two. Now you can run the following command:

```
source bin/activate
```

On Windows, things are slightly different. To create a virtual environment, you will probably need to use the full path to virtualenv:

```
c:\Python27\Scripts\virtualenv.exe
```

You should still change directories into your new folder, but instead of **bin**, there will be a **Scripts** folder that can run **activate** out of:

```
Scripts\activate
```

Once activated, you can install any other 3rd party Python package.

Note: It is recommended that you install all 3rd party packages, such as ReportLab or Pillow, in a Python Virtual Environment or a user folder. This prevents you from installing a lot of cruft in your system Python installation.

I would also like to mention that **pip** supports a **–user** flag that tells it to install the package just for the current user if the platform supports it. There is also an **–update** flag (or just **-U**) that you an use to update a package. You can use this flag as follows:

```
python -m pip install PACKAGE_NAME --upgrade
```

While you can also use pip install PACKAGE_NAME, it is now becoming a recommended practice to use the python -m approach. What this does differently is that it uses whatever Python is on your path and installs to that Python version. The **-m** flag tells Python to load or run a module which in this case is **pip**. This can be important when you have multiple versions of Python installed and you don't know which version of Python pip itself will install to. Thus, by using the python -m pip approach, you know that it will install to the Python that is mapped to your "python" command.

Now let's learn what we need to install to get ReportLab working!

Dependencies

You will need the Python language installed on your maching to use ReportLab. Python is pre-installed on Mac OS and most Linux distribututions. Reportlab 3 works with both Python 2.7 and Python 3.3+. You can get Python at https://www.python.org/. They have detailed instructions for installing and configuring Python as well as building Python should you need to do so.

ReportLab depends on the Python Imaging Library for adding images to PDFs. The Python Imaging Library itself hasn't been maintained in years, but you can use the **Pillow** (https://pillow.readthedocs.io/en/latest/) package instead. **Pillow** is a fork of the Python Imaging Library that supports Python 2 and Python 3 and has lots of new enhancements that the original package didn't have. You can install it with pip as well:

```
python -m pip install pillow
```

You may need to run **pip** as root or Administer depending on where your Python is installed or if you are installing to a virtualenv. You may find that you enjoy Pillow so much that you want to install it in your system Python in addition to your virtual environment.

We are ready to move on and learn how to install ReportLab!

Installation

Reportlab 3 works with both Python 2.7 and Python 3.3+. This book will be focusing on using Python 3 and ReportLab 3.x, but you can install ReportLab 3 the same way in both versions of Python using pip:

```
python -m pip install reportlab
```

If you are using an older version of Python such as Python 2.6 or less, then you will need to use ReportLab 2.x. These older versions of ReportLab have *.exe installers for Windows or a tarball for other operating systems. If you happen to run a ReportLab exe installer, it will install to Python's system environment and not your virtual environment.

If you run into issues installing ReportLab, please go to their website and read the documentation on the subject at https://www.reportlab.com/

Now you should be ready to use ReportLab!

Configuration

ReportLab supports a few options that you can configure globally on your machine or server. This configuration file can be found in the following file: **reportlab/rl_settings.py** (ex. C:\PythonXX\Lib\site-packages\reportlab). There are a few dozen options that are commented in the source. Here's a sampling:

- **verbose** - A range of integer values that can be used to control diagnostic output
- **shapeChecking** - Defaults to 1. Set to 0 to turn off most error checking in ReportLab's graphics modules
- **defaultEncoding** - WinAnsiEncoding (default) or MacRomanEncoding
- **defaultPageSize** - A4 is the default, but you can change it to something else, such as letter or legal
- **pageCompression** - What compression level to use. The documentation doesn't say what values can be used though
- **showBoundary** - Defaults to 0, but can be set to 1 to get boundary lines drawn
- **T1SearchPath** - A Python list of strings that are paths to T1Font fonts
- **TTFSearchPath** - A Python list of strings that are paths to TrueType fonts

As I said, there are a lot of other settings that you can modify in that Python script. I highly recommend opening it up and reading through the various options to see if there's anything that you will need to modify for your environment. In fact, you can do so in your Python interpreter by doing the following:

```
>>> from reportlab import rl_settings
>>> rl_settings.verbose
0
>>> rl_settings.shapeChecking
1
```

You can now easily check out each of the settings in an interactive manner.

Reader Feedback

I welcome your feedback. If you'd like to let me know what you thought of this book, you can send comments to the following email address:

comments@pythonlibrary.org

Errata

I try my best not to publish errors in my writings, but it happens from time to time. If you happen to see an error in this book, feel free to let me know by emailing me at the following:

errata@pythonlibrary.org

Code Examples

Code from the book can be downloaded from Github at the following address:

- https://github.com/driscollis/reportlabbookcode

Here's an alternate shortlink to the above as well:

- http://bit.ly/2nc7sbP

Now, let's get started!

Chapter 1 - Getting Started with Reportlab

ReportLab is a very powerful library. With a little effort, you can make pretty much any layout that you can think of. I have used it to replicate many complex page layouts over the years. In this chapter we will be learning how to use ReportLab's **pdfgen** package. You will discover how to do the following:

- Draw text
- Learn about fonts and text colors
- Creating a text object
- Draw lines
- Draw various shapes

The pdfgen package is very low level. You will be drawing or "painting" on a canvas to create your PDF. The canvas gets imported from the pdfgen package. When you go to paint on your canvas, you will need to specify X/Y coordinates that tell ReportLab where to start painting. The default is (0,0) whose origin is at the lowest left corner of the page. Many desktop user interface kits, such as wxPython, Tkinter, etc, also have this concept. You can place buttons absolutely in many of these kits using X/Y coordinates as well. This allows for very precise placement of the elements that you are adding to the page.

The other item that I need to make mention of is that when you are positioning an item in a PDF, you are positioning by the number of **points** you are from the origin. It's points, not pixels or millimeters or inches. Points! Let's take a look at how many points are on a letter sized page:

```
>>> from reportlab.lib.pagesizes import letter
>>> letter
(612.0, 792.0)
```

Here we learn that a letter is 612 points wide and 792 points high. Let's find out how many points are in an inch and a millimeter, respectively:

```
>>> from reportlab.lib.units import inch
>>> inch
72.0
>>> from reportlab.lib.units import mm
>>> mm
2.834645669291339
```

This information will help us position our drawings on our painting. At this point, we're ready to create a PDF!

The Canvas Object

The canvas object lives in the pdfgen package. Let's import it and paint some text:

```
# hello_reportlab.py

from reportlab.pdfgen import canvas

c = canvas.Canvas("hello.pdf")
c.drawString(100, 100, "Welcome to Reportlab!")
c.showPage()
c.save()
```

In this example, we import the canvas object and then instantiate a Canvas object. You will note that the only requirement argument is a filename or path. Next we call **drawString()** on our canvas object and tell it to start drawing the string 100 points to the right of the origin and 100 points up. After that we call **showPage()** method. The **showPage()** method will save the current page of the canvas. It's actually not required, but it is recommended. The **showPage()** method also ends the current page. If you draw another string or some other element after calling showPage(), that object will be drawn to a new page. Finally we call the canvas object's **save()** method, which save the document to disk. Now we can open it up and see what our PDF looks like:

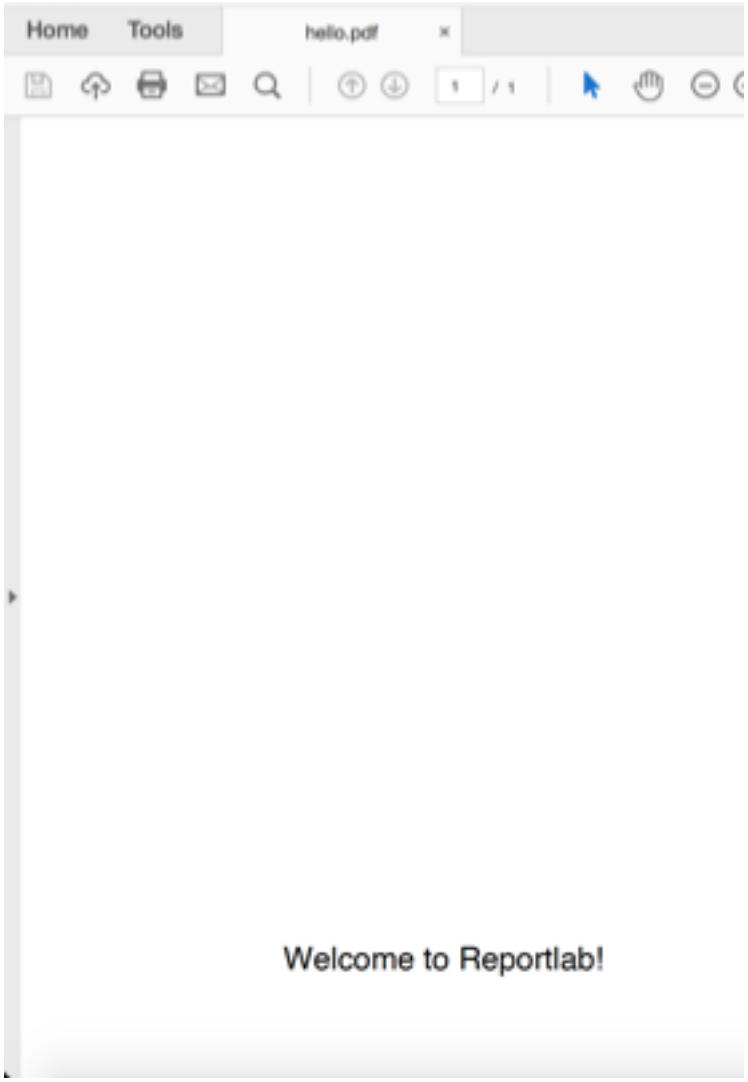


Fig. 1-1: Welcome to ReportLab

What you might notice is that our text is near the bottom of the document. The reason for this is that the origin, (0,0), is the bottom left corner of the document. So when we told ReportLab to paint our text, we were telling it to start painting 100 points from the left-hand side and 100 points from the bottom. This is in contrast to creating a user interface in Tkinter or wxPython where to origin is the top left.

Also note that since we didn't specify a page size, it defaults to whatever is in the ReportLab config, which is usually A4. There are some common page sizes that can be found in **reportlab.lib.pagesizes**.

Let's look at the Canvas's constructor to see what it takes for arguments:

```
def __init__(self,filename,
             pagesize=None,
             bottomup = 1,
             pageCompression=None,
             invariant = None,
             verbosity=0,
             encrypt=None,
             cropMarks=None,
             pdfVersion=None,
             enforceColorSpace=None,
             ):
```

Here we can see that we can pass in the **pagesize** as an argument. The **pagesize** is actually a tuple of width and height in points. If you want to change the origin from the default of bottom left, then you can set the **bottomup** argument to **0**, which will change the origin to the top left.

The **pageCompression** argument is defaulted to zero or off. Basically it will tell ReportLab whether or not to compress each page. When compression is enabled, the file generation process is slowed. If your work needs your PDFs to be generated as quickly as possible, then you'll want to keep the default of zero. However if speed isn't a concern and you'd like to use less disk space, then you can turn on page compression. Note that images in PDFs will always be compressed, so the primary use case for turning on page compression is when you have a huge amount of text or lots of vector graphics per page.

ReportLab's User Guide makes no mention of what the **invariant** argument is used for, so I took a look at the source code. According to the source, it *produces repeatable, identical PDFs with same timestamp info (for regression testing)*. I have never seen anyone use this argument in their code and since the source says it is for regression testing, I think we can safely ignore it.

The next argument is **verbosity**, which is used for logging levels. At zero (0), ReportLab will allow other applications to capture the PDF from standard output. If you set it to one (1), a confirmation message will be printed out every time a PDF is created. There may be additional levels added, but at the time of writing, these were the only two documented.

The **encrypt** argument is used to determine if the PDF should be encrypted as well as how it is encrypted. The default is obviously **None**, which means no encryption at all. If you pass a string to **encrypt**, that string will be the password for the PDF. If you want to encrypt the PDF, then you will need to create an instance of **reportlab.lib.pdfencrypt.StandardEncryption** and pass that to the **encrypt** argument.

The **cropMarks** argument can be set to True, False or to an object. Crop marks are used by printing houses to know where to crop a page. When you set cropMarks to True in ReportLab, the page will become 3 mm larger than what you set the page size to and add some crop marks to the corners. The object that you can pass to cropMarks contains the following parameters: borderWidth, markColor, markWidth and markLength. The object allows you to customize the crop marks.

The **pdfVersion** argument is used for ensuring that the PDF version is greater than or equal to what was passed in. Currently ReportLab supports versions 1-4.

Finally, the **enforceColorSpace** argument is used to enforce appropriate color settings within the PDF. You can set it to one of the following:

- cmyk
- rgb
- sep
- sep_black
- sep_cmyk

When one of these is set, a standard **_PDFColorSetter** callable will be used to do the color enforcement. You can also pass in a callable for color enforcement.

Let's go back to our original example and update it just a bit. Now as I mentioned earlier, in ReportLab you can position your elements (text, images, etc) using points. But thinking in points is kind of hard when we are used to using millimeters or inches. So I found a clever function we can use to help us on StackOverflow (http://stackoverflow.com/questions/4726011/wrap-text-in-a-table-reportlab):

```
def coord(x, y, height, unit=1):
    x, y = x * unit, height -  y * unit
    return x, y
```

This function requires your x and y coordinates as well as the height of the page. You can also pass in a unit size. This will allow you to do the following:

```
# canvas_coords.py

from reportlab.pdfgen import canvas
from reportlab.lib.pagesizes import letter
from reportlab.lib.units import mm

def coord(x, y, height, unit=1):
    x, y = x * unit, height -  y * unit
    return x, y

c = canvas.Canvas("hello.pdf", pagesize=letter)
width, height = letter

c.drawString(*coord(15, 20, height, mm), text="Welcome to Reportlab!")
c.showPage()
c.save()
```

In this example we pass the **coord** function the x and y coordinates, but we tell it to use millimeters as our unit. So instead of thinking in points, we are telling ReportLab that we want the text to start 15 mm from the left and 20 mm from the top of the page. Yes, you read that right. When we use the **coord** function, it uses the height to swap the origin's y from the bottom to the top. If you had set your Canvas's **bottomUp** parameter to zero, then this function wouldn't work as expected. In fact, we could simplify the coord function to just the following:

```
def coord(x, y, unit=1):
    x, y = x * unit, y * unit
    return x, y
```

Now we can update the previous example like this:

```
# canvas_coords2.py

from reportlab.pdfgen import canvas
from reportlab.lib.units import mm

def coord(x, y, unit=1):
    x, y = x * unit, y * unit
    return x, y

c = canvas.Canvas("hello.pdf", bottomup=0)

c.drawString(*coord(15, 20, mm), text="Welcome to Reportlab!")
c.showPage()
c.save()
```

That seems pretty straight-forward. You should take a minute or two and play around with both examples. Try changing the x and y coordinates that you pass in. Then try changing the text too and see what happens!

Canvas Methods

The **canvas** object has many methods. Let's learn how we can use some of them to make our PDF documents more interesting. One of the easiest methods to use **setFont**, which will let you use a PostScript font name to specify what font you want to use. Here is a simple example:

```
# font_demo.py

from reportlab.lib.pagesizes import letter
from reportlab.pdfgen import canvas

def font_demo(my_canvas, fonts):
    pos_y = 750
    for font in fonts:
        my_canvas.setFont(font, 12)
        my_canvas.drawString(30, pos_y, font)
        pos_y -= 10

if __name__ == '__main__':
    my_canvas = canvas.Canvas("fonts.pdf",
                              pagesize=letter)
    fonts = my_canvas.getAvailableFonts()
    font_demo(my_canvas, fonts)
    my_canvas.save()
```

To make things a bit more interesting, we will use the **getAvailableFonts** canvas method to grab all the available fonts that we can use on the system that the code is ran on. Then we will pass the canvas object and the list of font names to our **font_demo** function. Here we loop over the font names, set the font and call the **drawString** method to draw each font's name to the page. You will also note that we have set a variable for the starting Y position that we then decrement by 10 each time we loop through. This is to make each text string draw on a separate line. If we didn't do this, the strings would write on top of each other and you would end up with a mess.

Here is the result when you run the font demo:

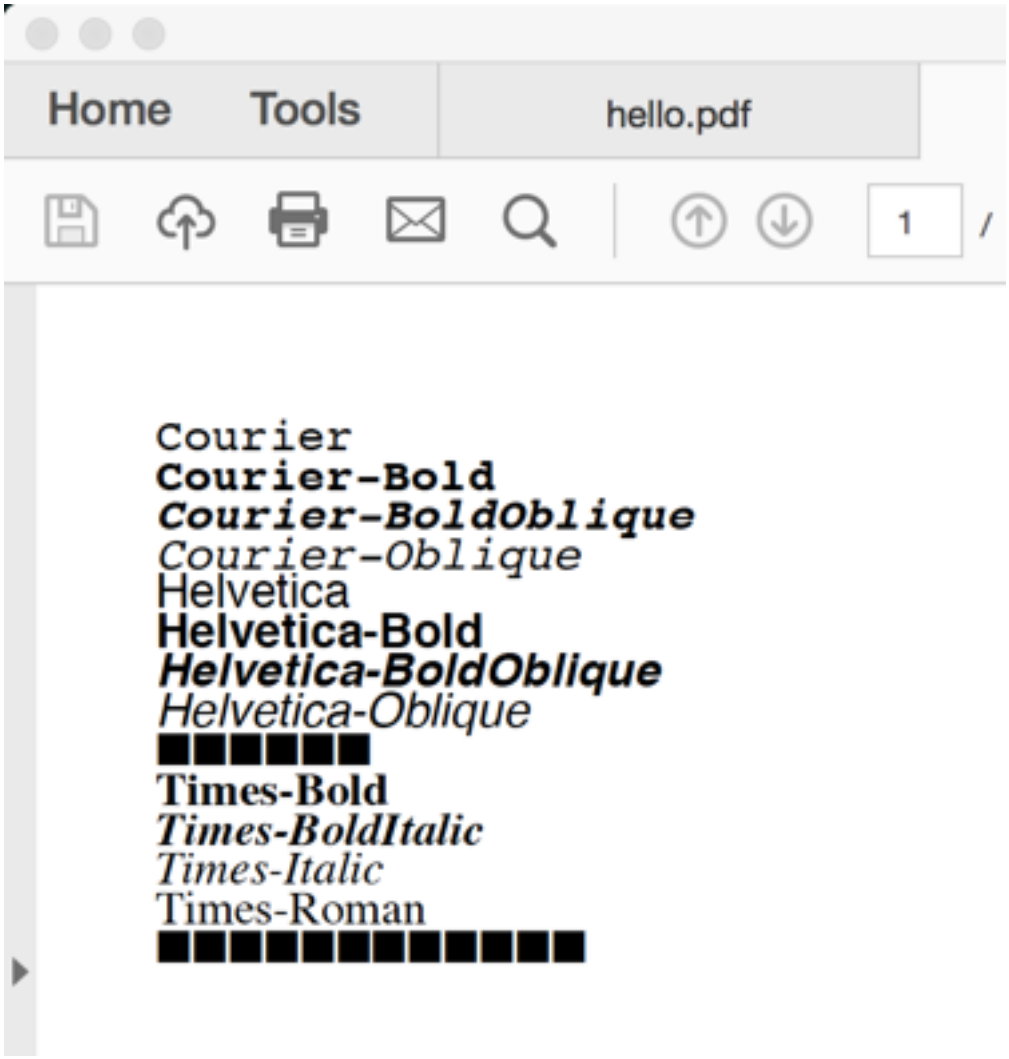


Fig. 1-2: Available fonts in ReportLab

If you want to change the font color using a canvas method, then you would want to look at **setFillColor** or one of its related methods. As long as you call that before you draw the string, the color of the text will change as well.

Another fun thing you can is use the canvas's **rotate** method to draw text at different angles. We will also learn how to use the **translate** method. Let's take a look at an example:

```
# rotating_demo.py

from reportlab.lib.pagesizes import letter
from reportlab.lib.units import inch
from reportlab.pdfgen import canvas

def rotate_demo():
    my_canvas = canvas.Canvas("rotated.pdf",
                              pagesize=letter)
    my_canvas.translate(inch, inch)
    my_canvas.setFont('Helvetica', 14)
    my_canvas.drawString(inch, inch, 'Normal')
    my_canvas.line(inch, inch, inch+100, inch)

    my_canvas.rotate(45)
    my_canvas.drawString(inch, -inch, '45 degrees')
    my_canvas.line(inch, inch, inch+100, inch)

    my_canvas.rotate(45)
    my_canvas.drawString(inch, -inch, '90 degrees')
    my_canvas.line(inch, inch, inch+100, inch)

    my_canvas.save()

if __name__ == '__main__':
    rotate_demo()
```

Here we use the **translate** method to set our origin from the bottom left to an inch from the bottom left and an inch up. Then we set out font face and font size. Next write out some text normally and then we rotate the coordinate system itself 45 degrees before we draw a string. According to the ReportLab user guide, you will want to specify the y coordinate in the negative since the coordinate system is now in a rotated state. If you don't do that, your string will be drawn outside the page's boundary and you won't see it. Finally we rotate the coordinate system another 45 degrees for a total of 90 degrees, write out one last string and draw the last line.

It is interesting to look at how the lines moved each time we rotated the coordinate system. You can see that the origin of the last line moved all the way to the very left-hand edge of the page.

Here is the result when I ran this code:

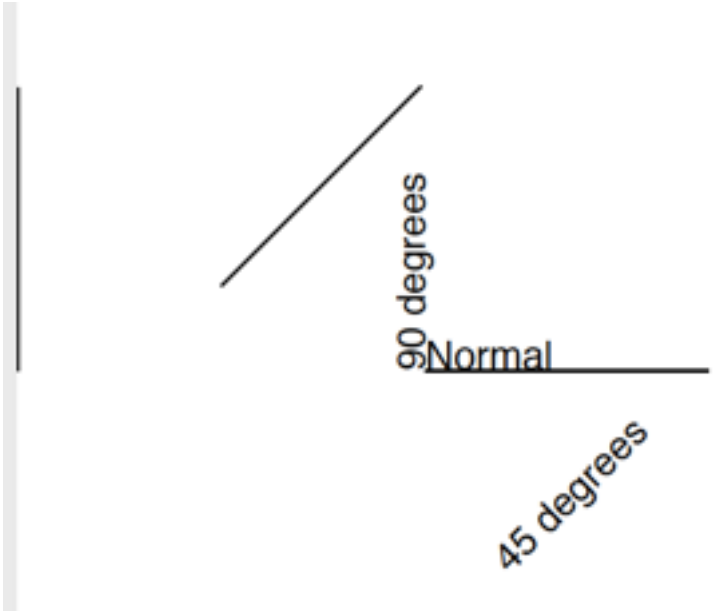


Fig. 1-3: Rotated text

Now let's take a moment learn about alignment.

String Alignment

The canvas supports more string methods than just the plain **drawString** method. You can also use **drawRightString**, which will draw your string right-aligned to the x-coordinate. You can also use **drawAlignedString**, which will draw a string aligned to the first pivot character, which defaults to the period. This is useful if you want to line up a series of floating point numbers on the page. Finally, there is the **drawCentredString** method, which will draw a string that is "centred" on the x-coordinate. Let's take a look:

```
# string_alignment.py

from reportlab.pdfgen import canvas
from reportlab.lib.pagesizes import letter


def string_alignment(my_canvas):
    width, height = letter

    my_canvas.drawString(80, 700, 'Standard String')
    my_canvas.drawRightString(80, 680, 'Right String')

    numbers = [987.15, 42, -1,234.56, (456.78)]
    y = 650
```

```
    for number in numbers:
        my_canvas.drawAlignedString(80, y, str(number))
        y -= 20

    my_canvas.drawCentredString(width / 2, 550, 'Centered String')

    my_canvas.showPage()


if __name__ == '__main__':
    my_canvas = canvas.Canvas("string_alignment.pdf")
    string_alignment(my_canvas)
    my_canvas.save()
```

When you run this code, you will quickly see how each f these strings get aligned. Personally I thought the **drawAlignedString** method was the most interesting, but the others are certainly handy in their own right. Here is the result of running the code:

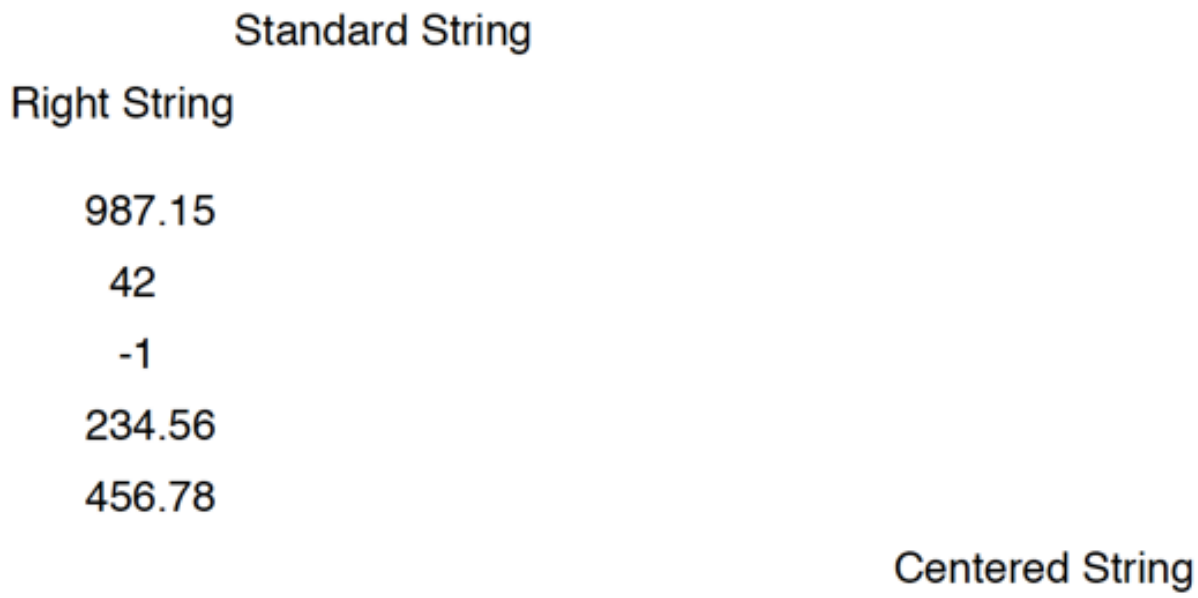


Fig. 1-4: String Alignment

The next canvas methods we will learn about are how to draw lines, rectangles and grids!

Drawing lines on the canvas

Drawing a line in ReportLab is actually quite easy. Once you get used to it, you can actually create very complex drawings in your documents, especially when you combine it with some of