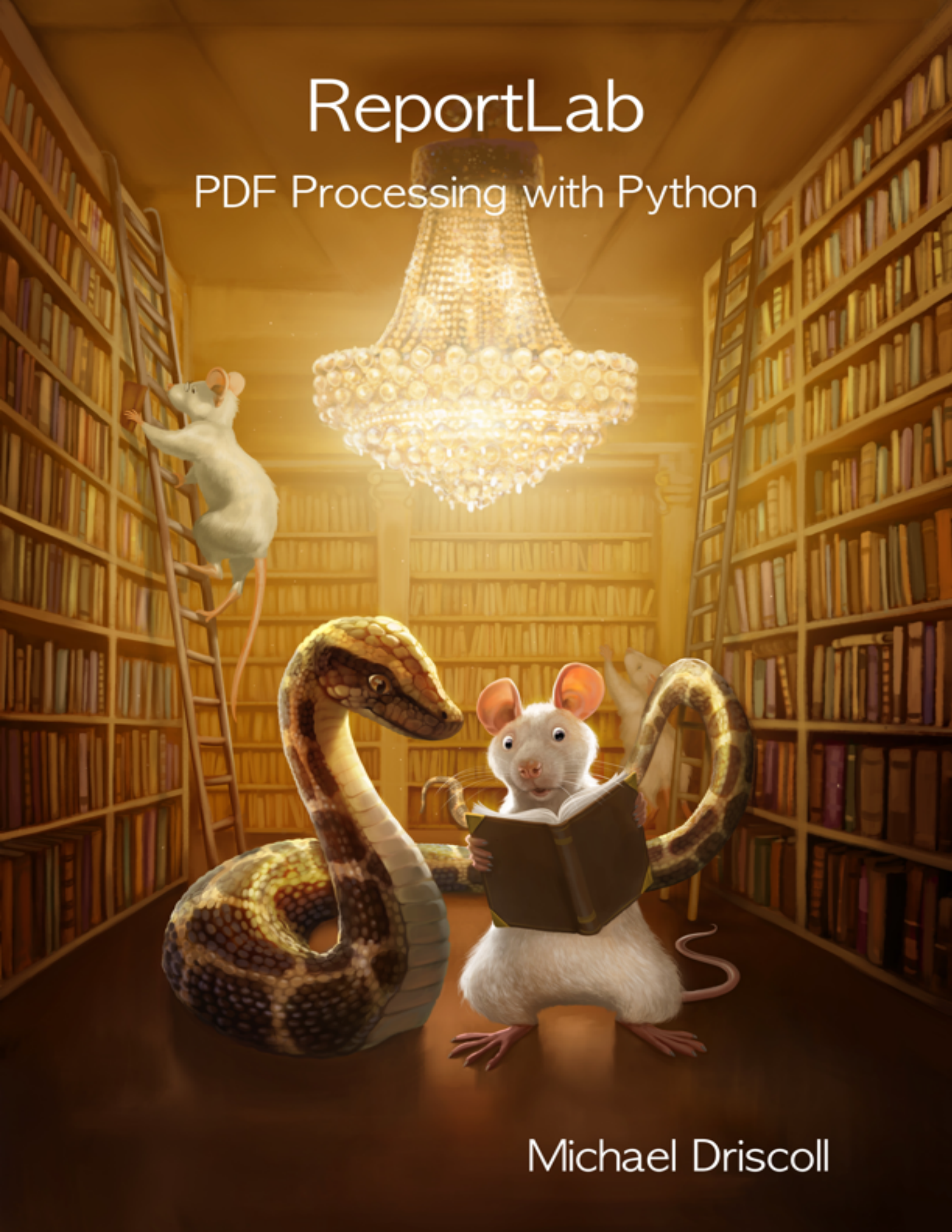# ReportLab

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

Leanpub

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 distibututions. 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:
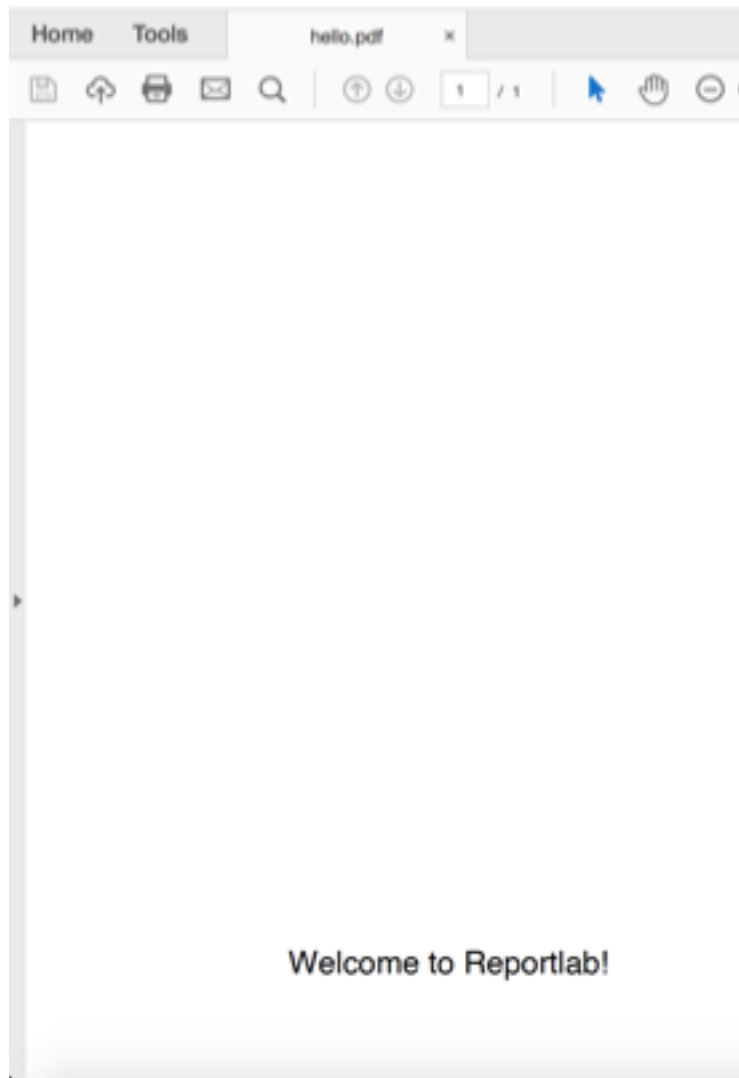
Fig. 1-1: Welcome to ReportLab

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:

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

Now we can update the previous example like this:

```python
# 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:

```python
# 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.

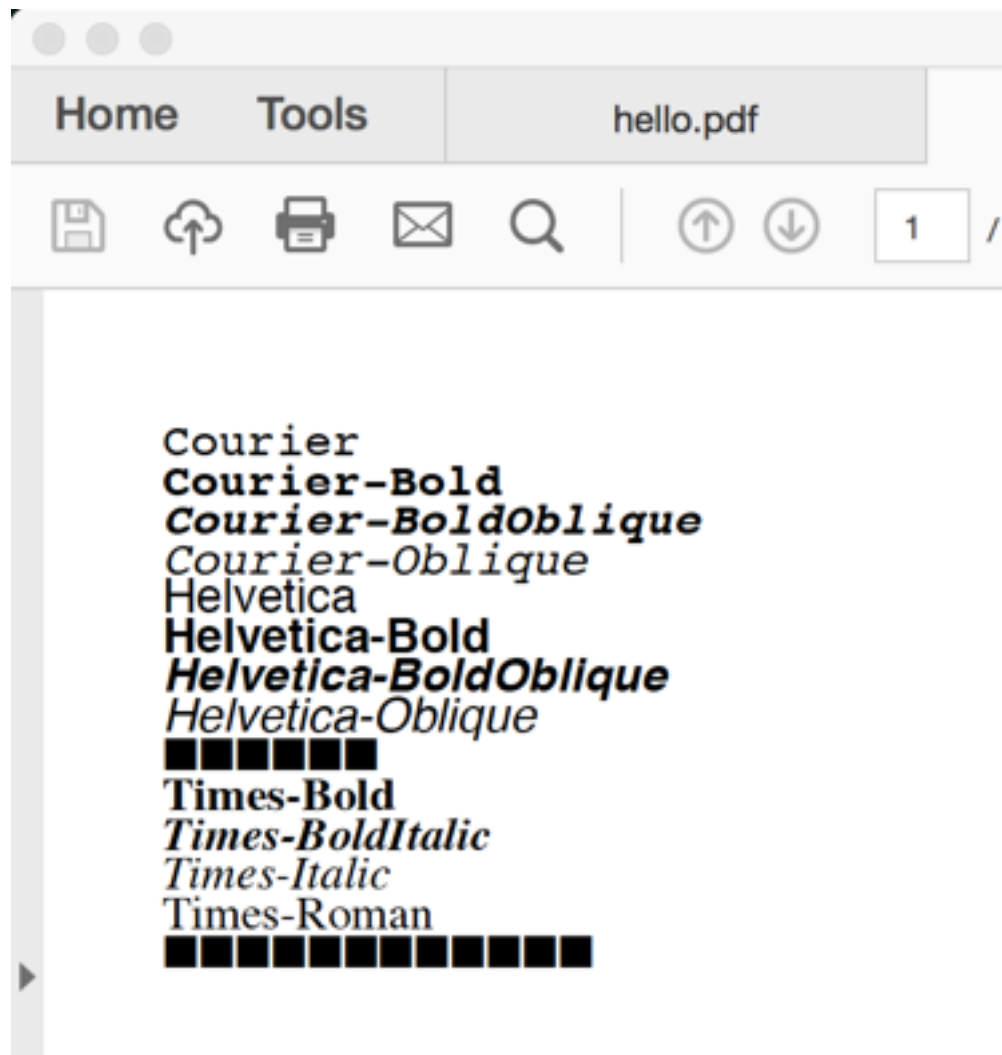Here is the result when you run the font demo:

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:

```python
# 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.

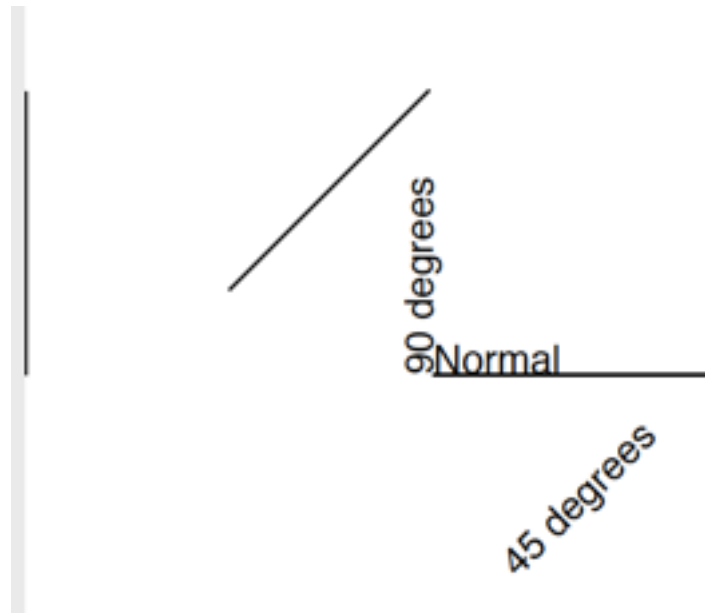Here is the result when I ran this code:

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:

```python
# 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
```

```python
    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

ReportLab's other features. The method to draw a straight line is simply **line**. Let's take a look at a simple example:

```python
# drawing_lines.py

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


def draw_lines(my_canvas):
    my_canvas.setLineWidth(.3)

    start_y = 710
    my_canvas.line(30, start_y, 580, start_y)

    for x in range(10):
        start_y -= 10
        my_canvas.line(30, start_y, 580, start_y)


if __name__ == '__main__':
    my_canvas = canvas.Canvas("lines.pdf", pagesize=letter)
    draw_lines(my_canvas)
    my_canvas.save()
```

Here we create a simple **draw_lines** function that accepts a canvas object as its sole parameter. Then we set the line's width via the **setLineWidth** method. Finally we create a single line. You will notice that the **line** method accepts four arguments: x1, y1, x2, y2. These are the beginning x and y coordinates as well as the ending x and y coordinates. We add another 10 lines by using a **for** loop. If you run this code, your output will look something like this:
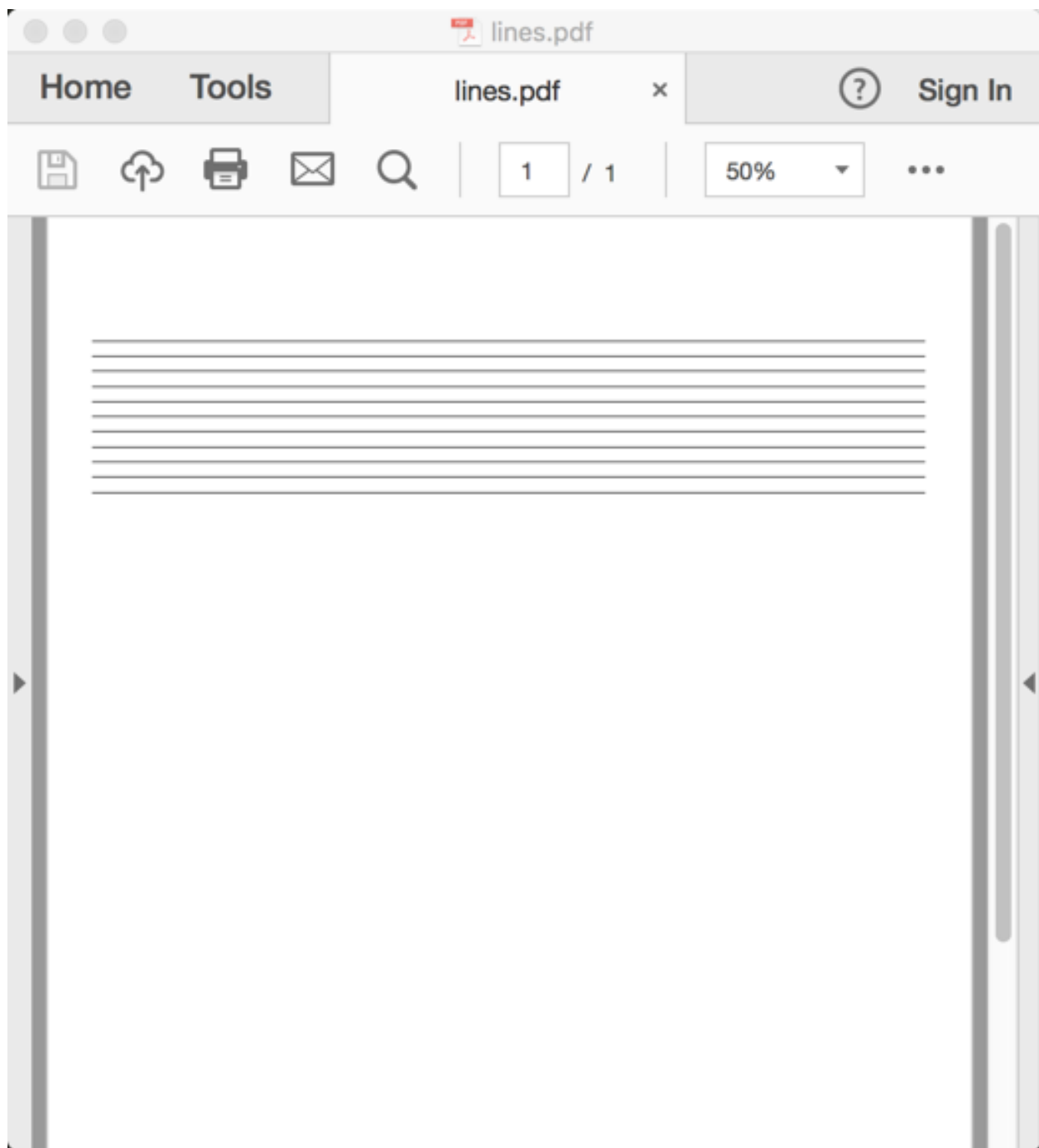
Fig. 1-5: Drawing lines on the canvas

The canvas supports several other drawing operations. For example, you can also draw rectangles, wedges and circles. Here's a simple demo:

```python
# drawing_polygons.py

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


def draw_shapes():
    c = canvas.Canvas("draw_other.pdf")
    c.setStrokeColorRGB(0.2, 0.5, 0.3)
    c.rect(10, 740, 100, 80, stroke=1, fill=0)
    c.ellipse(10, 680, 100, 630, stroke=1, fill=1)
    c.wedge(10, 600, 100, 550, 45, 90, stroke=1, fill=0)
    c.circle(300, 600, 50)
    c.save()


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

When you run this code, you should end up with a document that draws something like this:
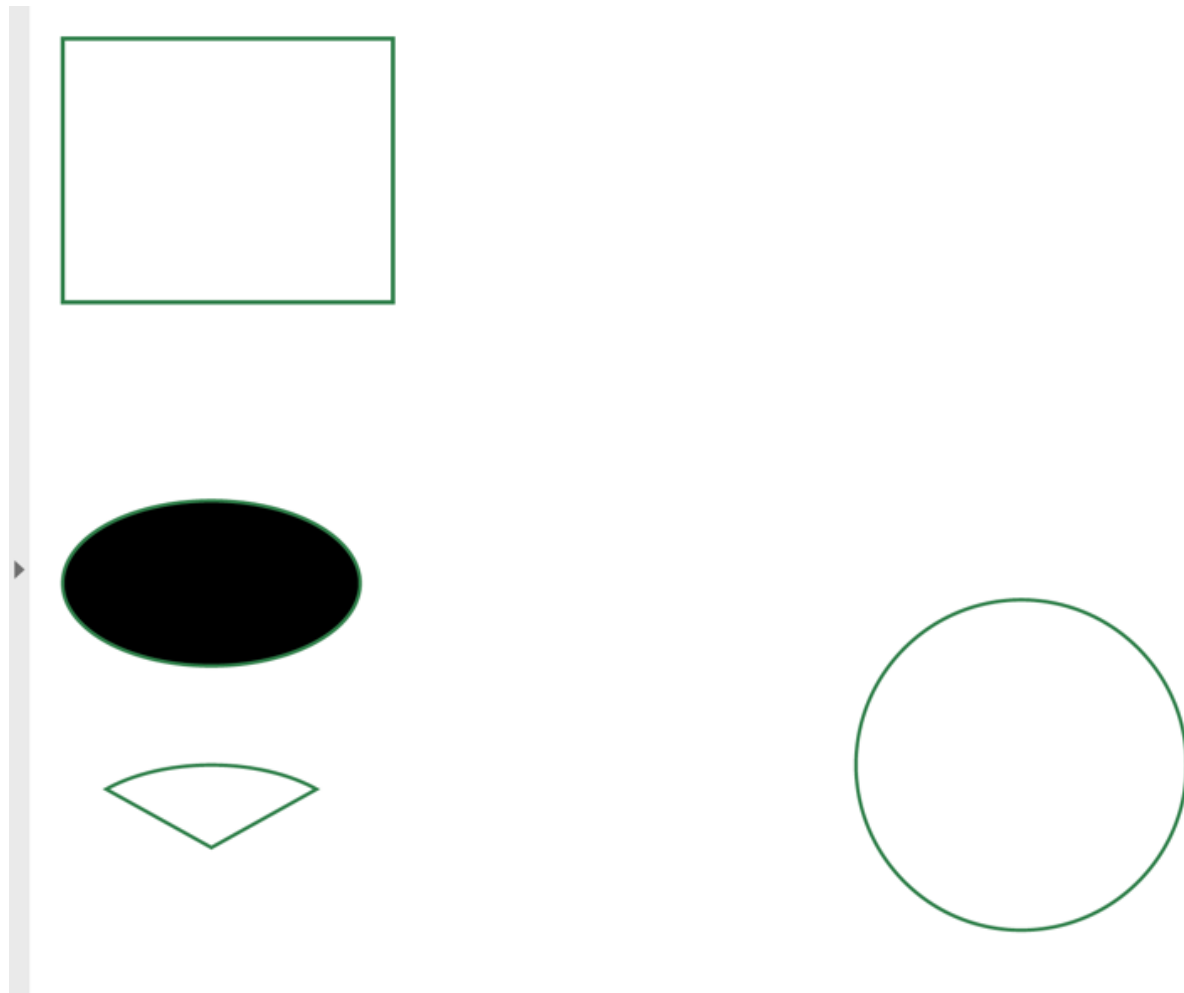
Fig. 1-6: Drawing polygons on the canvas

Let's take a few moments to go over the various arguments that each of these polygon methods accept. The **rect**'s code signature looks like this:

```
def rect(self, x, y, width, height, stroke=1, fill=0):
```

That means that you set the lower left-hand corner of the rectangle's position via its x/y parameters. Then you set its width and height. The stroke parameter tells ReportLab if it should draw the lines, so in the demo code I set **stroke=1**, or True. The fill parameter tells ReportLab to fill the interior of the polygon that I drew with a color.

Now let's look at the **ellipse**'s definition:

```
def ellipse(self, x1, y1, x2, y2, stroke=1, fill=0):
```

This one is very similar to the rect. According to method's docstring, the x1, y1, x2, y2 parameters are the corner points of the enclosing rectangle. The stroke and fill parameters operate the same way as the rect's. Just for fun, we went ahead and set the ellipse's fill to 1.

Next we have the wedge:

```
def wedge(self, x1,y1, x2,y2, startAng, extent, stroke=1, fill=0):
```

The x1,y1, x2,y2 parameters for the wedge actually correspond to the coordinates of an invisible enclosing rectangle that goes around a full 360 degree circle version of the wedge. So you will need to imagine that the full circle with a rectangle around it to help you position a wedge correctly. It also has a starting angle parameter (**startAng**) and the **extent** parameter, which basically tells the wedge how far out to arc to. The other parameters have already been explained.

Finally we reach the **circle** polygon. It's method looks like this:

```
def circle(self, x_cen, y_cen, r, stroke=1, fill=0):
```

The circle's arguments are probably the most self-explanatory of all of the polygons we have looked at. The **x_cen** and **y_cen** arguments are the x/y coordinates of the center of the circle. The **r** argument is the radius. The stroke and fill arguments are pretty obvious.

All the poloygons have the ability to set the stroke (or line) color via the **setStrokeColorRGB** method. It accepts Red, Green, Blue values for its parameters. You can also set the stroke color by using the **setStrokeColor** or the **setStrokeColorCMYK** method.

There are corresponding fill color setters too (i.e. setFillColor, setFillColorRGB, setFillColorCMYK), although I didn't show those in the demo code. The reason that wasn't covered above is that we are going to cover it in the very next section!

Using Colors in ReportLab

ReportLab has support for applying colors in several different ways. You can add a color to a drawing using one of two methods: RGB or CMYK. In the case of RGB, there are actually three different methods:

- By specifying red/green/blue values (i.e. values must be between zero and one)
- By name or
- By gray level

Frankly I think the gray level specification is a bit misleading in that you're really only specifying what level of gray you want, not any other color. Let's start with gray levels though and work our way up through the other methods!

```python
# gray_color_demo.py

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


def gray_color_demo():
    my_canvas = canvas.Canvas("grays.pdf",
                                pagesize=letter)
    my_canvas.setFont('Helvetica', 10)
    x = 30

    grays = [0.0, 0.25, 0.50, 0.75, 1.0]

    for gray in grays:
        my_canvas.setFillGray(gray)
        my_canvas.circle(x, 730, 20, fill=1)
        gray_str = "Gray={gray}".format(gray=gray)
        my_canvas.setFillGray(0.0)
        my_canvas.drawString(x-10, 700, gray_str)
        x += 75

    my_canvas.save()

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

This code should be pretty self-explanatory, but let's break it down anyway. First off, we create a list of different gray values. Then we set the fill color using **setFillGray**. After that we draw a circle and tell it to fill. Finally we create a string and draw it underneath each circle so we have the circles labeled with their gray value. When you run this code, you should see something like this:
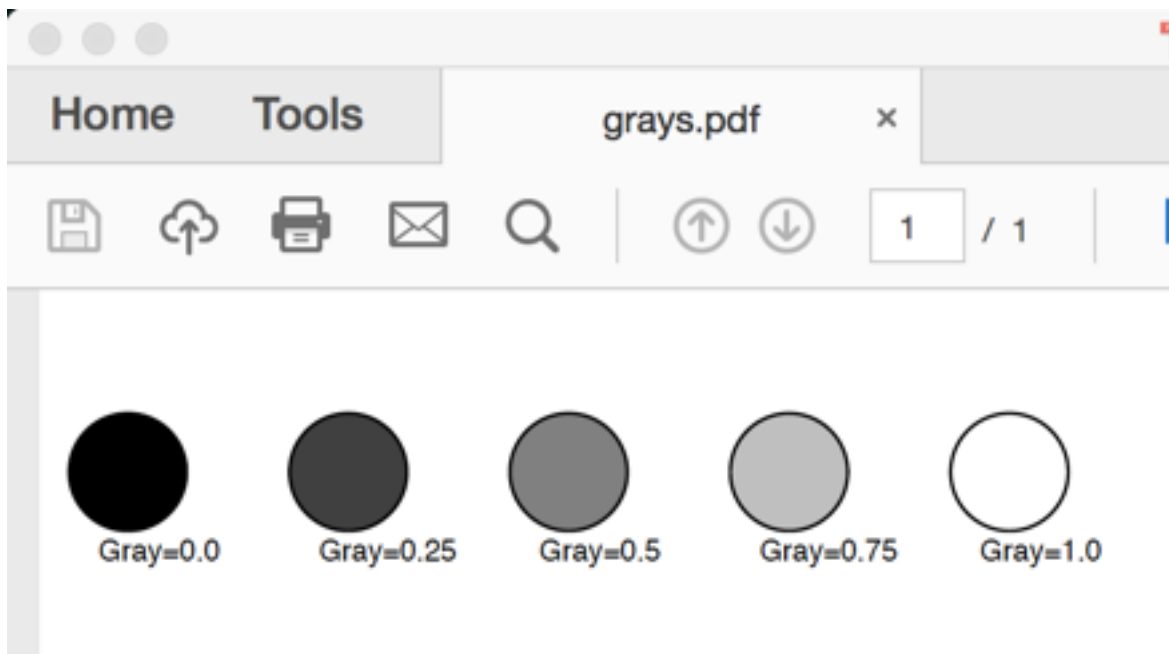
Fig. 1-7: ReportLab's shades of gray

Let's move on to learn how to add some color. The first method we will look at is setting the fill color by name:

```python
# colors_demo.py

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


def color_demo():
    my_canvas = canvas.Canvas("colors.pdf",
                              pagesize=letter)
    my_canvas.setFont('Helvetica', 10)
    x = 30

    sample_colors = [colors.aliceblue,
                     colors.aquamarine,
                     colors.lavender,
                     colors.beige,
                     colors.chocolate]

    for color in sample_colors:
        my_canvas.setFillColor(color)
```

```
        my_canvas.circle(x, 730, 20, fill=1)
        color_str = "{color}".format(color=color._lookupName())
        my_canvas.setFillColor(colors.black)
        my_canvas.drawString(x-10, 700, color_str)
        x += 75

    my_canvas.save()

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

Here we import the **colors** sub-module from ReportLab. Then we create a list of sample colors to iterate over like we did with the gray demo earlier. Then we do the loop and call **setFillColor** with the color's name. Now if you actually were to print the color to standard out, so you would see something like this:

```
>>> print(colors.aliceblue)
Color(.941176,.972549,1,1)
```

So these aren't exactly just names. In fact, they are ReportLab Color objects with RGB values and an intensity level between 0 (dark) and 1 (full intensity). Anyway, the next piece of code of note is where we grab the color's name via the **_lookupName()** method. The rest of the code is pretty easy to figure out.
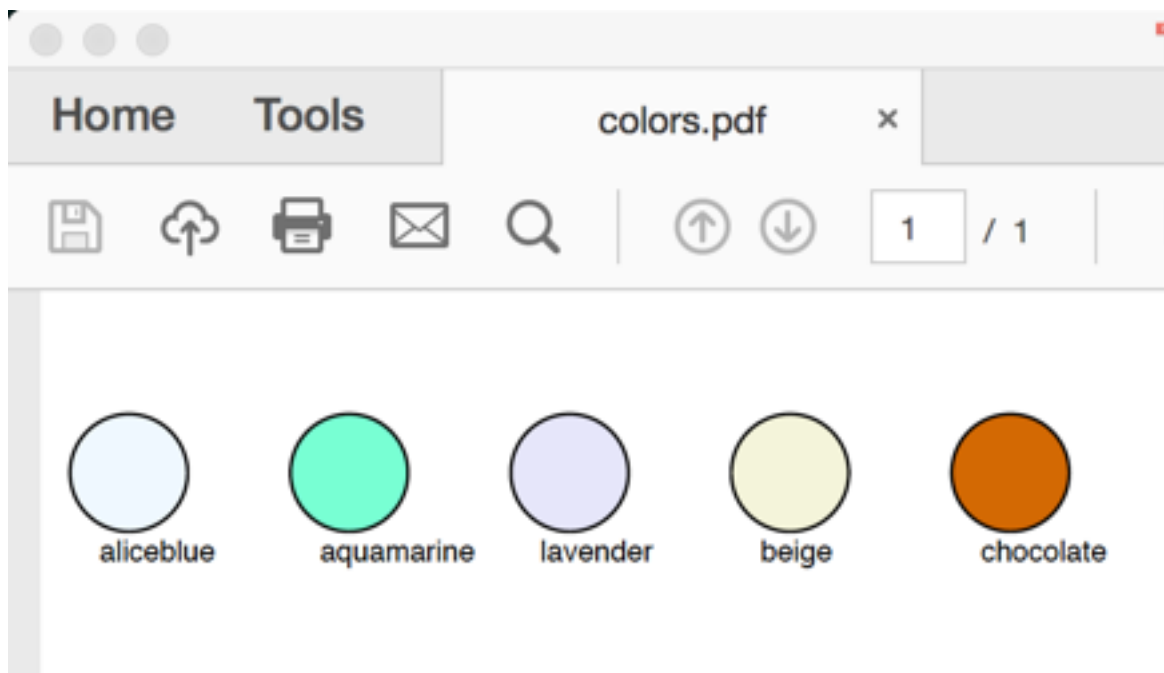

Fig. 1-8: Example colors

If you want to use straight-up RGB or CMYK values, then you can edit the example above to call **setFillColorRGB** or **setFillColorCMYK** respectively. They accept RGB or CMYK colors plus an alpha parameter respectively. The primary reason to use CMYK is for when you want more control over how the ink in your printer is applied to the printer. Of course you will need a printer that support CMYK for this to be really useful.

Adding a Photo

ReportLab supports adding images to your PDFs via the **Python Imaging Library (PIL)** package. Note that PIL is no longer supported and it is recommended that you download the **Pillow** project, a fork of PIL that works with both Python 2 and Python 3, something that the original PIL didn't do. To install Pillow, you just need to issue the pip command in your terminal, although you will only need to run this if it didn't automatically install when you installed ReportLab itself:

```
pip install pillow
```

Now that we have Pillow installed, let's talk about how to insert a photo into your PDF. The Report-Lab canvas object supports two methods: **drawInlineImage** and **drawImage**. It is recommended that you use the newer **drawImage** method as it will cache the image and allow you to draw it many times while only being stored once in the PDF. If you use **drawInlineImage**, it will embed the image into the page stream itself, which makes it much less efficient as the image will be added multiple times to the document if you draw it more than once. While the documentation doesn't mention this, I would assume that this can also make the PDF larger in file size.

In this book, we will be using the **drawImage** method. Here's an example:

```python
# image_on_canvas.py

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


def add_image(image_path):
    my_canvas = canvas.Canvas("canvas_image.pdf",
                              pagesize=letter)
    my_canvas.drawImage(image_path, 30, 600,
                        width=100, height=100)
    my_canvas.save()

if __name__ == '__main__':
    image_path = 'snakehead.jpg'
    add_image(image_path)
```

As you can see, the **drawImage** method accepts the image's file path and its x / y position. These arguments are required. You can also specify the width and height of the image. Note that this will not automatically scale the image or keep its aspect ratio, so you may end up stretching the image if you don't know what you are doing. Finally you can also supply a **mask** parameter which will allow you to create a transparent image. This parameter is a list of 6 numbers which allows to define a range of RGB values which will be masked.

In ReportLab version 2 and newer, there is a **preserveAspectRatio** parameter that you can set as well as an **anchor** parameter. See the docstring for the **drawImage** method for more information on their proper usage.

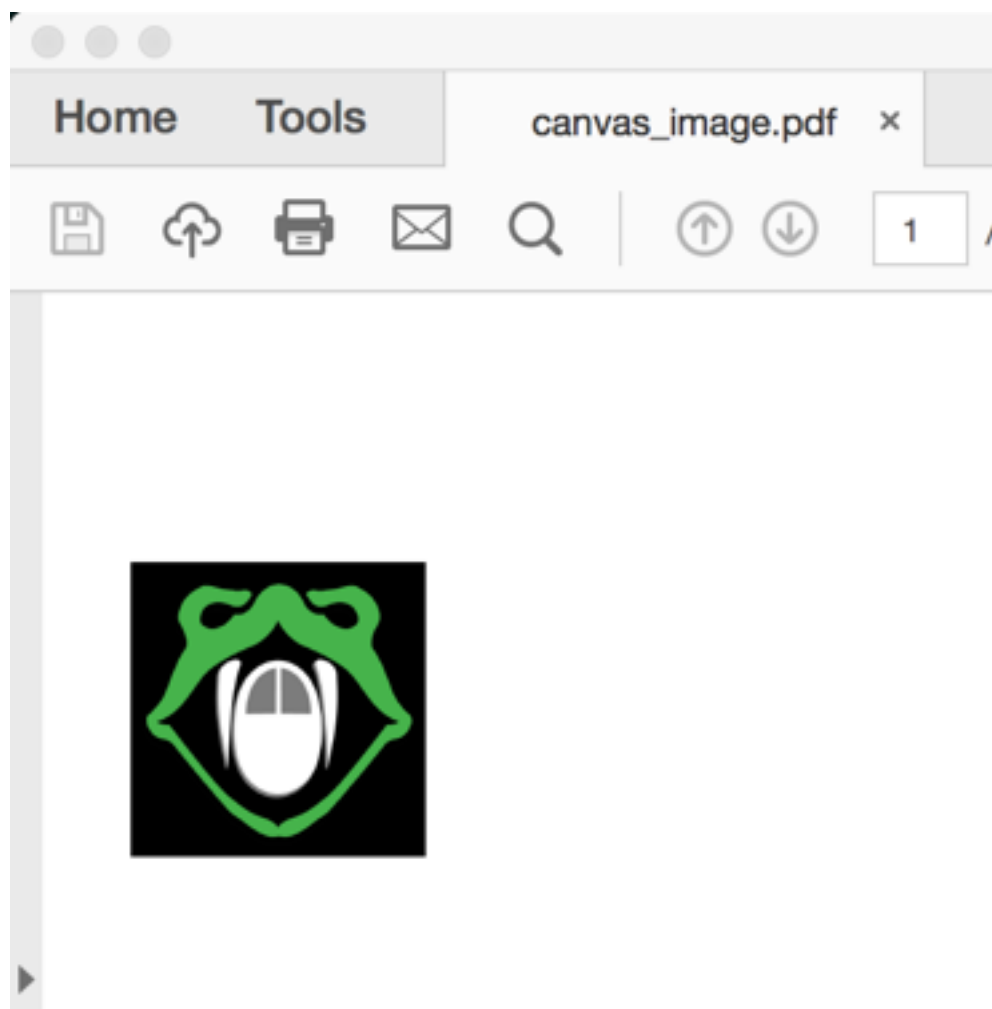If you are curious, here's what the PDF ended up looking:

Fig. 1-9: Adding an image

The textobject

For additional control of your text's presentation, you can also use a **textobject**. Frankly I have never had the need for one of these as ReportLab's **Paragraph** class gives you more than enough control over the presentation of your text. But to be thorough, I will show you how to create and use a textobject. One benefit to use the textobject is that it will make the PDF generation faster if you use it instead of making separate calls to **drawString**.

Let's see a quick little demo:

```python
# textobject_demo.py

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


def textobject_demo():
    my_canvas = canvas.Canvas("txt_obj.pdf",
                              pagesize=letter)
    # Create textobject
    textobject = my_canvas.beginText()

    # Set text location (x, y)
    textobject.setTextOrigin(10, 730)

    # Set font face and size
    textobject.setFont('Times-Roman', 12)

    # Write a line of text + carriage return
    textobject.textLine(text='Python rocks!')

    # Change text color
    textobject.setFillColor(colors.red)

    # Write red text
    textobject.textLine(text='Python rocks in red!')

    # Write text to the canvas
    my_canvas.drawText(textobject)

    my_canvas.save()
```

```python
if __name__ == '__main__':
    textobject_demo()
```

Here we learn that to create a **textobject**, we need to call the canvas's **beginText** method. If you happen to print out the textobject, you will find that it's technically an instance of **reportlab.pdfgen.textobject.PDFTextObject**. Anyway, now that we have a textobject, we can set its cursor position using a call to **setTextOrigin**. Then we set the font face and size as we saw before. The next new item is the call to **textLine**, which will allow you to write a string to the buffer plus what is basically a carriage return. The docstring for this method states that it makes the "text cursor moves down", but that amounts to a carriage return in my eyes. There is also a **textLines** method that allows you to write a multiline string out as well. If you want to control the location of the cursor, then you might want to use **textOut** as it won't add a carriage return to the end of the string.

The next thing we do is set the font color by calling **setFillColor**. In this example, we set the the next string of text to a red color. The last step is to call **drawText**, which will actually draw whatever you have in your textobject. If you skip calling **drawText**, then your text won't be written out and you may end up with an empty PDF document.
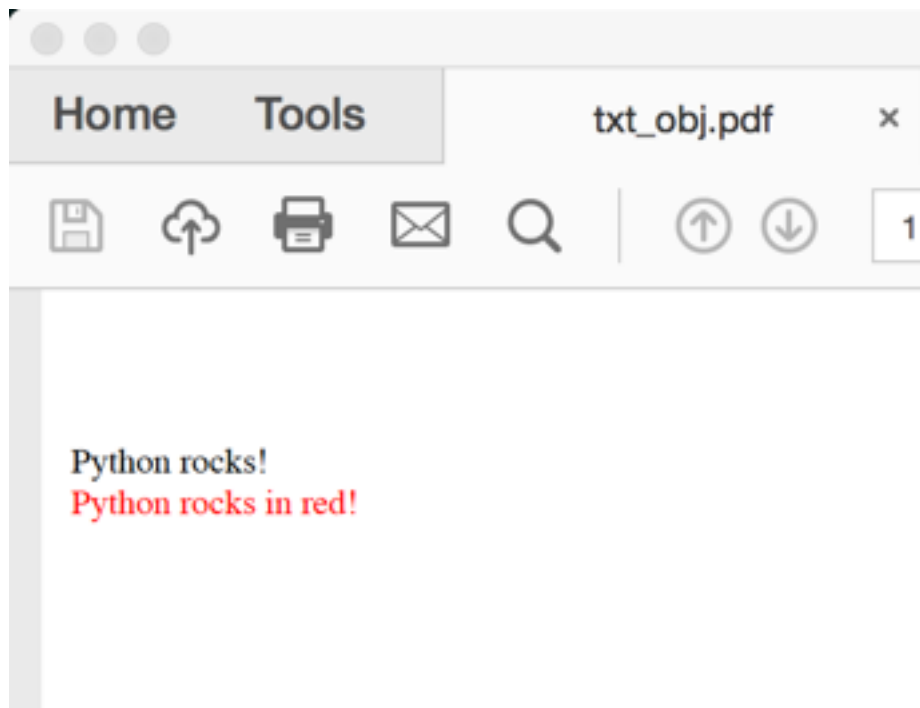
Here's the resulting PDF:

Fig. 1-10: Using a textobject

There are a lot of other methods you can call from your textobject. For example, if you want to move your cursor's position somewhere other than the very next line, you can call **moveCursor**. Let's take a look:

```python
# cursor_moving.py

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


def textobject_cursor():
    canvas_obj = canvas.Canvas("textobj_cursor.pdf", pagesize=letter)

    # Create textobject
    textobject = canvas_obj.beginText()

    # Set text location (x, y)
    textobject.setTextOrigin(10, 730)

    for indent in range(4):
        textobject.textLine('ReportLab cursor demo')
        textobject.moveCursor(15, 15)

    canvas_obj.drawText(textobject)
    canvas_obj.save()


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

Here we just set up a loop that will print out the same string four times, but at four different positions. You will note that we move the cursor 15 points to the right and 15 points down the page with each iteration of the loop. Yes, when using a textobject, a positive y number will move you down.

Now, let's say you would like to change the inter-character spacing; all you need to do is call **setCharSpace**. In fact, you can do a lot of interesting spacing tricks with textobject, such as changing the space between word using **setWordSpace** or the space between lines by calling **setLeading**. Let's take a look at how we might change the spacing of our text:

```python
# char_spacing_demo.py

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


def textobject_char_spacing():
    canvas_obj = canvas.Canvas("textobj_char_spacing.pdf",
                                pagesize=letter)
```

```python
    # Create textobject
    textobject = canvas_obj.beginText()

    # Set text location (x, y)
    textobject.setTextOrigin(10, 730)

    spacing = 0
    for indent in range(8):
        textobject.setCharSpace(spacing)
        line = '{} - ReportLab spacing demo'.format(spacing)
        textobject.textLine(line)
        spacing += 0.7

    canvas_obj.drawText(textobject)
    canvas_obj.save()


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

In this example, we increase the loop factor to 8 iterations and call **setCharSpace()** each time through the loop. We start with zero spacing and then add 0.7 in each iteration. You can see the result here:

```
0 - ReportLab spacing demo
0.7 - ReportLab spacing demo
1.4 - ReportLab spacing demo
2.099999999999996 - ReportLab spacing demo
2.8 - ReportLab spacing demo
3.5 - ReportLab spacing demo
4.2 - ReportLab spacing demo
4.9 - ReportLab spacing demo
```

Fig. 1-11: Character spacing with the textobject

Now let's see how applying word spacing effects our text:

```python
# wordspacing_demo.py

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


def wordspacer():
    canvas_obj = canvas.Canvas("textobj_word_spacing.pdf",
                               pagesize=letter)

    # Create textobject
    textobject = canvas_obj.beginText()

    # Set text location (x, y)
    textobject.setTextOrigin(10, 730)

    word_spacing = 0
    for indent in range(8):
        textobject.setWordSpace(word_spacing)
        line = '{} - ReportLab spacing demo'.format(word_spacing)
        textobject.textLine(line)
        word_spacing += 1.5

    canvas_obj.drawText(textobject)
    canvas_obj.save()


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

This example is pretty much the same as the previous one, but you will note that we are calling **setWordSpace()** instead of **setCharSpace()** and we are increasing the spacing by a factor of 1.5 in this example. The resulting text looks like this:

```
0 - ReportLab spacing demo
1.5 - ReportLab spacing demo
3.0 - ReportLab spacing demo
4.5 - ReportLab spacing demo
6.0 - ReportLab spacing demo
7.5 - ReportLab spacing demo
9.0 - ReportLab spacing demo
10.5 - ReportLab spacing demo
```

Fig. 1-12: Word spacing with the textobject

If you would like to create a superscript or subscript, then you would want to call **setRise** on your textobject. Let's create a demo that demonstrates how setting the rise works in ReportLab:

```python
# canvas_rising.py

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

def apply_scripting(textobject, text, rise):
    textobject.setFont("Helvetica-Oblique", 8)
    textobject.setRise(rise)
    textobject.textOut(text)
    textobject.setFont("Helvetica-Oblique", 12)
    textobject.setRise(0)


def main():
    canvas_obj = canvas.Canvas("textobj_rising.pdf",
                               pagesize=letter)

    # Create textobject
    textobject = canvas_obj.beginText()
    textobject.setFont("Helvetica-Oblique", 12)

    # Set text location (x, y)
    textobject.setTextOrigin(10, 730)

    textobject.textOut('ReportLab ')
```
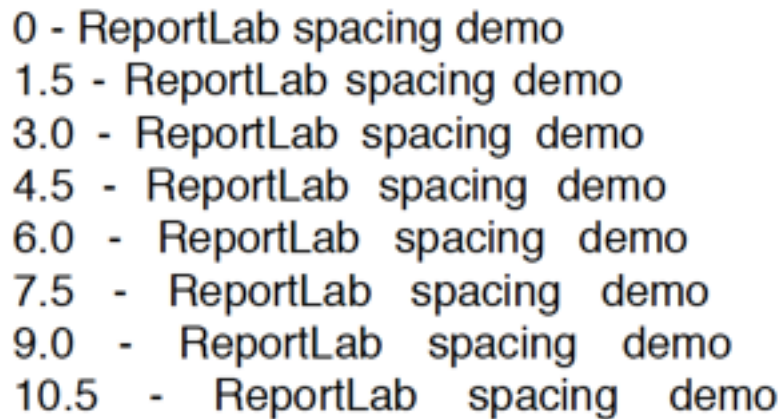
```
    apply_scripting(textobject, 'superscript ', 7)


    textobject.textOut('and ')

    apply_scripting(textobject, 'subscript ', -7)


    canvas_obj.drawText(textobject)
    canvas_obj.save()


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

Here we create a couple of functions, **apply_scripting** and **main**. The main function will create our canvas and all the other bits and pieces we need. Then we write out some normal text. The next few lines are where we apply superscripting (positive) and subscripting (negative). Note that we need to set the rise back to zero between the superscript and subscript to make the word, "and", appear in the right location. As soon as you apply a rising value, it will continue to apply from that point on. So you will want to reset it to zero to make sure the text stays in a normal location. You will also note that we set the font size for the super and subscripts to be smaller than the regular text. Here is the result of running this example:
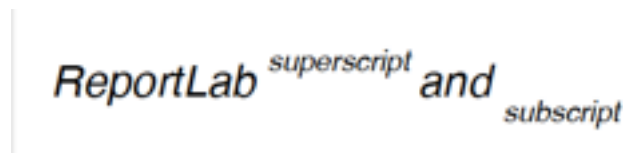
Fig. 1-13: Canvas rising with the textobject

Check out ReportLab's user guide for more interesting things you can do or check the source code itself.

Create a Page Break

One of the first things I wanted to know when I was creating PDFs with ReportLab was how to add a page break so I could have multipage PDF documents. The canvas object allows you to do this via the **showPage** method. Note however that for complex documents, you will almost certainly use ReportLab's **flowables**, which are special classes specifically for "flowing" your documents across multiple pages. Flowables are kind of mind bending in their own right, but they are also a lot nicer to use than trying to keep track of which page you are on and where your cursor position is at all times.

Canvas Orientation (Portrait vs. Landscape)

ReportLab defaults its page orientation to Portrait, which is what all word processors do as well. But sometimes you will want to use a page in **landscape** instead. There are at least two ways to tell Reportlab to use a landscape orientation. The first one is a convenience function called landscape that you can import from **reportlab.lib.pagesizes**. You would use it like this:

```python
from reportlab.lib.pagesizes import landscape, letter
from reportlab.pdfgen import canvas

c = canvas.Canvas('test.pdf', pagesize=letter)
c.setPageSize( landscape(letter) )
```

The other way to set landscape is just set the page size explicitly:

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

c = canvas.Canvas('test.pdf', pagesize=letter)
c.setPageSize( (11*inch, 8.5*inch) )
```

You could make this more generic by doing something like this though:

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

width, height = letter

c = canvas.Canvas('test.pdf', pagesize=letter)
c.setPageSize( (height, width) )
```

This might make more sense, especially if you wanted to use other popular page sizes, like A4.

Other methods

There are a bunch of additional methods that I'm not even going to cover in this chapter. For example, there are methods to set some metadata for your PDF, such as the author (**setAuthor**), title (**setTitle**) and subject (**setSubject**). The **bookmarkPage** method is actually useful if you want to create bookmarks in your PDF though. There are also methods for creating a named form and then interacting with it (**beginForm**, **endForm**, etc). Just go look through the ReportLab's user guide for a complete list or check out the canvas's source.

A Simple Sample Application

Sometimes it's nice to see how you can take what you've learned and see if applied. So let's take some of the methods we've learned about here and create a simple application that create a form:

```python
# sample_form_letter.py

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


def create_form(filename, date, amount, receiver):
    """
    @param date: The date to use
    @param amount: The amount owed
    @param receiver: The person who received the amount owed
    """
    my_canvas = canvas.Canvas(filename, pagesize=letter)
    my_canvas.setLineWidth(.3)
    my_canvas.setFont('Helvetica', 12)

    my_canvas.drawString(30, 750,'OFFICIAL COMMUNIQUE')
    my_canvas.drawString(30, 735,'OF ACME INDUSTRIES')

    my_canvas.drawString(500, 750, date)
    my_canvas.line(480, 747, 580, 747)

    my_canvas.drawString(275, 725,'AMOUNT OWED:')
    my_canvas.drawString(500, 725, amount)
    my_canvas.line(378,723, 580, 723)

    my_canvas.drawString(30, 703,'RECEIVED BY:')
    my_canvas.line(120, 700, 580, 700)
    my_canvas.drawString(120, 703, receiver)

    my_canvas.save()

if __name__ == '__main__':
    create_form('form.pdf', '01/23/2018',
                '$1,999', 'Mike')
```

Here we just create a simple function called **create_form** that accepts the filename, the date we want for our form, the amount owed and the person who receives the amount owed. Then we paint everything in the desired locations and save the file. When you run this, you will see the following:

OFFICIAL COMMUNIQUE   01/23/2018
OF ACME INDUSTRIES
                              AMOUNT OWED: $1,999

RECEIVED BY: Mike

Fig. 1-14: A Sample Form letter

That looks pretty professional for a short piece of code.

Wrapping Up

We covered a lot of information in this chapter. You should now know how to create a pretty basic PDF. I highly recommend trying out the examples in this chapter and then going back and editing them a bit to see what all you can accomplish on your own. Once you are done playing around the canvas methods mentioned here, prepare yourselves as the next chapter will be about how ReportLab handles fonts.

Chapter 2 - ReportLab and Fonts

We covered a little information about fonts in chapter 1, but I thought it was important to talk a little about ReportLab's font support. A few years ago, ReportLab added support for Asian languages. They also support TrueType fonts and Type-1 fonts. It's also worth talking about encodings in this chapter, which is what we will discuss next.

Unicode / UTF8 is the Default

Way back in 2006, ReportLab made it so that all text you provide to their APIs should be in UTF8 or as Python Unicode objects. This should be done with the **canvas**.**DrawString** methods as well as in the flowables that accept text (i.e. strings) as their argument. Fortunately Python 3's "strings" are Unicode by default, so you won't even have to think all that much about this topic if you just use the latest Python. However if you are using an older version of Python AND your string is not encoded as UTF8, then you will get a **UnicodeDecodeError** if you give it any character that is not ASCII.

The fix is to just encode your text as UTF8 or use a Unicode object. Just keep that in mind if you run into these sorts of issues.

The Standard Fonts

ReportLab comes with a set of fonts by default. They don't need to be stored/embedded in your PDF as Adobe's Acrobat Reader guarantees that they will be there. You can get a list of the fonts available by calling the **getAvailableFonts()** canvas method. This is the list I received:

- Courier
- Courier-Bold
- Courier-BoldOblique
- Courier-Oblique
- Helvetica
- Helvetica-Bold
- Helvetica-BoldOblique
- Helvetica-Oblique
- Symbol
- Times-Bold
- Times-BoldItalic
- Times-Italic
- Times-Roman

- ZapfDingbats

ReportLab supports limited automatic font substitution. This will happen only if the ReportLab engine detects a character that is not in your font of choice. In these cases, ReportLab's engine will attempt to switch to Symbol or ZapfDingbats to display said character. Here's a quick demo:

```python
# basic_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("basic_font_demo.pdf",
                              pagesize=letter)
    fonts = my_canvas.getAvailableFonts()
    font_demo(my_canvas, fonts)
    my_canvas.save()
```

You will note that all we need to do to get a list of fonts is to call the **getAvailableFonts** method. When you run this code, you will get the following in your PDF:

Fig. 2-1: The standard fonts

Now let's learn about embedding fonts in your PDF document.

Other Type-1 Fonts

If you need to embed a non-standard font, then you will need a couple of font description files. One needs to be in the Adobe AFM (Adobe Font Metrics) format and the other needs to be in PFB (Printer Font Binary) format. The Adobe AFM file is actually ASCII and tells ReportLab about the glyph's of the font. A font's glyph describes the height, width, bounding box information and other font metrics. The PFB describes the shapes of the font and is in binary format, so you won't be able to read it without a hex editor or similar. I have had to use these files for embedding a check font into a PDF before.

Fortunately, ReportLab actually includes an open source font called *DarkGardenMK* that they distribute with ReportLab in their fonts folder. Let's write a little demo that shows how to embed this font in our PDF:

```python
# type1_font_demo.py

import os
import reportlab

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


def embedded_font_demo():
    my_canvas = canvas.Canvas("type1_font_demo.pdf",
                              pagesize=letter)
    reportlab_folder = os.path.dirname(reportlab.__file__)
    fonts_folder = os.path.join(reportlab_folder, 'fonts')
    print('ReportLab font folder is located at {}'.format(
        fonts_folder))

    afm = os.path.join(fonts_folder, 'DarkGardenMK.afm')
    pfb = os.path.join(fonts_folder, 'DarkGardenMK.pfb')

    # Register the font so we can use it
    font_face = pdfmetrics.EmbeddedType1Face(afm, pfb)
    pdfmetrics.registerTypeFace(font_face)

    face_name = 'DarkGardenMK'
    font = pdfmetrics.Font('DarkGardenMK',
                           face_name,
                           'WinAnsiEncoding')
    pdfmetrics.registerFont(font)

    # Use the font!
    my_canvas.setFont('DarkGardenMK', 40)
    my_canvas.drawString(10, 730, 'The DarkGardenMK font')
    my_canvas.save()

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

This is a fairly complex process. First we have our imports. Note that we need **pdfmetrics** to register the font. Then we create our demo function and build the font folder by getting ReportLab's install location. I added a **print()** statement so that you could find out where this folder is located in case you would like to browse through it. Next we get the paths to the AFM and PFB files. Now we're finally

ready to register the font with ReportLab. That process begins by instantiating the pdfmetrics's **EmbeddedType1Face** class and passing it the AFM and PFB file paths. Next we register the font's face via the call to **registerTypeFace**. Funnily enough, I accidentally discovered that if I don't call that function, the code works just fine, so I am actually not sure why this is needed other than possibly a sanity check.

Anyway, the next step is to instantiate the **Font** class by passing it the name of the font, the face name and the encoding. Then you can just register the font by calling **registerFont**. Now we can actually use the font in our PDF. This is the result I got when I ran this code:

Fig. 2-2: Embedding a Type-1 Font

I don't know when you would want to use this font other than possibly for when a dragon speaks, but it looks kind of neat!

You can also edit the T1 font search path in **rl_settings.py**, which is located in your ReportLab installation location. The variable you will need to set is called **T1SearchPath**. On my system, the default looks like this:

```
>>> from reportlab import rl_settings
>>> rl_settings.T1SearchPath
('c:/Program Files/Adobe/Acrobat 9.0/Resource/Font',
 'c:/Program Files/Adobe/Acrobat 8.0/Resource/Font',
 'c:/Program Files/Adobe/Acrobat 7.0/Resource/Font',
 'c:/Program Files/Adobe/Acrobat 6.0/Resource/Font',
 'c:/Program Files/Adobe/Acrobat 5.0/Resource/Font',
 'c:/Program Files/Adobe/Acrobat 4.0/Resource/Font',
 '%(disk)s/Applications/Python %(sys_version)s/reportlab/fonts',
 '/usr/lib/Acrobat9/Resource/Font',
 '/usr/lib/Acrobat8/Resource/Font',
 '/usr/lib/Acrobat7/Resource/Font',
 '/usr/lib/Acrobat6/Resource/Font',
 '/usr/lib/Acrobat5/Resource/Font',
 '/usr/lib/Acrobat4/Resource/Font',
 '/usr/local/Acrobat9/Resource/Font',
 '/usr/local/Acrobat8/Resource/Font',
 '/usr/local/Acrobat7/Resource/Font',
 '/usr/local/Acrobat6/Resource/Font',
 '/usr/local/Acrobat5/Resource/Font',
 '/usr/local/Acrobat4/Resource/Font',
 '/usr/share/fonts/default/Type1',
```

```
    '%(REPORTLAB_DIR)s/fonts',
    '%(REPORTLAB_DIR)s/../fonts',
    '%(REPORTLAB_DIR)s/../../fonts',
    '%(CWD)s/fonts',
    '~/fonts',
    '~/.fonts',
    '%(XDG_DATA_HOME)s/fonts',
    '~/.local/share/fonts',
    '/usr/share/fonts/type1/gsfonts')
```

TrueType Fonts

Adding TrueType fonts in your PDF using ReportLab is a bit less complicated than embedding Type-1 Fonts. Let's take a look and see how TrueType font embedding differs from Type-1:

```python
# truetype_font_demo.py

import os
import reportlab

from reportlab.lib.pagesizes import letter
from reportlab.pdfbase import pdfmetrics
from reportlab.pdfbase.ttfonts import TTFont
from reportlab.pdfgen import canvas


def embedded_font_demo():
    my_canvas = canvas.Canvas("truetype_font_demo.pdf",
                              pagesize=letter)
    reportlab_folder = os.path.dirname(reportlab.__file__)
    fonts_folder = os.path.join(reportlab_folder, 'fonts')
    print('ReportLab font folder is located at {}'.format(
        fonts_folder))

    # Register the font so we can use it
    vera_font_path = os.path.join(fonts_folder, 'Vera.ttf')

    # Usage: TTFont(name,filename)
    vera_font = TTFont('Vera', vera_font_path)
    pdfmetrics.registerFont(vera_font)
```

```python
    # Use a generic font
    my_canvas.setFont('Helvetica', 40)
    my_canvas.drawString(10, 730, 'The Helvetica font')

    # Use the font!
    my_canvas.setFont('Vera', 40)
    my_canvas.drawString(10, 690, 'The Vera font')
    my_canvas.save()

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

One of the first changes is that we need to import **TTFont** from **reportlab.pdfbase.ttfonts**. Then we make an instance of that class by passing it the font's name and the font's file path. Then we call **registerFont** as we did with the Type-1 fonts in the previous section. The rest of the code is pretty much the same.

I would also like to point out that there is a **registerFontFamily** method that you should be aware of. This method will allow you to map the bold, italic and bolditalic versions of the font to the same name. Of course if you have all the different versions of the font, than you can use this function to register those name too. Here's the signature you would use for the **Vera** font:

```python
pdfmetrics.registerFontFamily('Vera', normal='Vera',bold='VeraBd',
                              italic='VeraIt',boldItalic='VeraBI')
```

As you can see, you just pass the name of the various flavors of the font to the appropriate parameter.

An alternative way to include TrueType fonts is to set the font search path in **rl_settings.py** in much the same way that you did the T1 search path. For TrueType fonts, you will want to set the **TTFSearchPath** variable. The defaults paths that ReportLab looks in are as follows:

```python
>>> from reportlab import rl_settings
>>> rl_settings.TTFSearchPath
('c:/winnt/fonts',
 'c:/windows/fonts',
 '/usr/lib/X11/fonts/TrueType/',
 '/usr/share/fonts/truetype',
 '/usr/share/fonts',
 '/usr/share/fonts/dejavu',
 '%(REPORTLAB_DIR)s/fonts',
 '%(REPORTLAB_DIR)s/../fonts',
 '%(REPORTLAB_DIR)s/../../fonts',
 '%(CWD)s/fonts',
 '~/fonts',
```

```
'~/.fonts',
'%(XDG_DATA_HOME)s/fonts',
'~/.local/share/fonts',
'~/Library/Fonts',
'/Library/Fonts',
'/Network/Library/Fonts',
'/System/Library/Fonts',
'/usr/share/fonts/truetype',
'/usr/share/fonts/truetype/kacst-one',
'/usr/share/fonts/truetype/freefont',
'/usr/share/fonts/truetype/nanum',
'/usr/share/fonts/truetype/ttf-khmeros-core',
'/usr/share/fonts/truetype/lohit-punjabi',
'/usr/share/fonts/truetype/takao-gothic',
'/usr/share/fonts/truetype/sinhala',
'/usr/share/fonts/truetype/ancient-scripts',
'/usr/share/fonts/truetype/tlwg',
'/usr/share/fonts/truetype/lyx',
'/usr/share/fonts/truetype/lao',
'/usr/share/fonts/truetype/fonts-guru-extra',
'/usr/share/fonts/truetype/abyssinica',
'/usr/share/fonts/truetype/dejavu',
'/usr/share/fonts/truetype/tibetan-machine',
'/usr/share/fonts/truetype/ubuntu-font-family',
'/usr/share/fonts/truetype/ttf-bitstream-vera',
'/usr/share/fonts/truetype/kacst',
'/usr/share/fonts/truetype/openoffice',
'/usr/share/fonts/truetype/liberation',
'/usr/share/fonts/truetype/noto',
'/usr/share/fonts/truetype/padauk')
```

If your font is in one of those paths, then you can simplify your code a bit because you won't need to include the entire path any longer:

```
vera_font = TTFont('Vera', 'Vera.ttf')
pdfmetrics.registerFont(vera_font)
```

Asian Fonts

A few years ago, ReportLab added support for Asian fonts. ReportLab currently supports, Japanese, Traditional Chinese (Taiwan / Hong Kong), Simplified Chinese (mainland China) and Korean. They do this by supporting the following fonts:

- chs = Chinese Simplified (mainland): 'STSong-Light'
- cht = Chinese Traditional (Taiwan): 'MSung-Light', 'MHei-Medium'
- kor = Korean: 'HYSMyeongJoStd-Medium','HYGothic-Medium'
- jpn = Japanese: 'HeiseiMin-W3', 'HeiseiKakuGo-W5'

If you use one of these fonts in your PDF, then you will likely discover that it isn't installed and Adobe's Reader may pop up a dialog like the following:
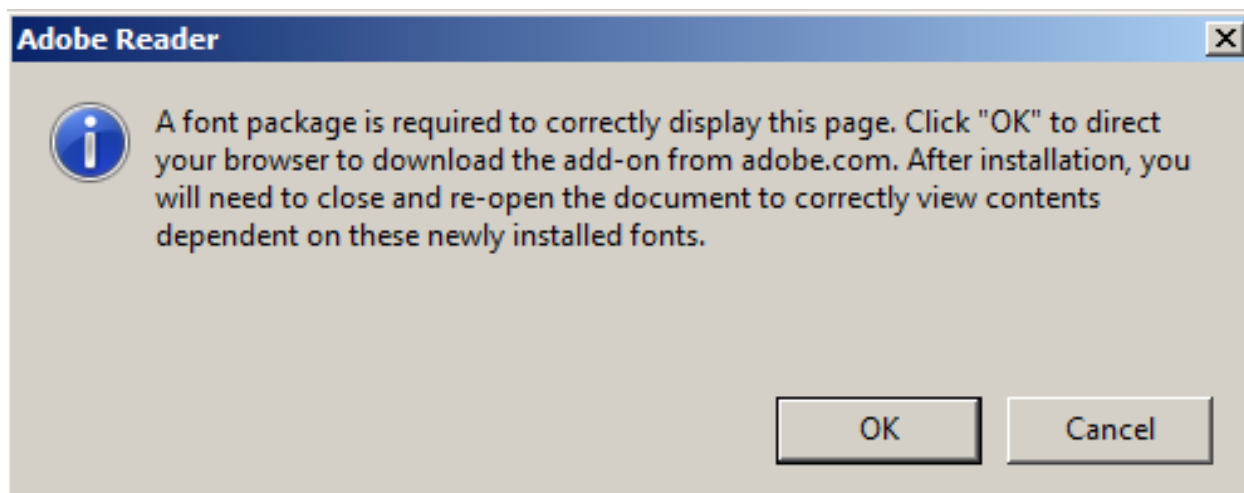

Fig. 2-3: **Font package warning**

Let's write some code to write the characters used for the word "Nippon", which means Japan. The characters we will use are Japanese kanji that were converted to Unicode. For this example, I just looked up a Unicode converter online and asked it to convert "Nippon" to Unicode. Here's the code:

```python
# asian_font_demo.py

# Works with Python 2 and 3
from reportlab.lib.pagesizes import letter
from reportlab.pdfbase import pdfmetrics
from reportlab.pdfbase.cidfonts import UnicodeCIDFont
from reportlab.pdfgen import canvas


def asian_font_demo():
    my_canvas = canvas.Canvas("asian_font_demo.pdf",
                              pagesize=letter)

    # Set a Japanese font
    pdfmetrics.registerFont(UnicodeCIDFont('HeiseiMin-W3'))
    my_canvas.setFont('HeiseiMin-W3', 16)
```

```
    # Find a word or phrase in Unicode to write out
    nippon = u'\u65e5\u672c'   # Nippon / Japan in Unicode

    my_canvas.drawString(25, 730, nippon)
    my_canvas.save()


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

When I ran this and tried to open the resulting PDF with Adobe Reader, I received the aforementioned dialog about needing to download a font package, **FontPack11009_XtdAlf_Lang.msi**, which ended up being a 52 MB download. If you don't install the font package, then the PDF will appear to be blank when opened. After installing the fonts, I got the following result:
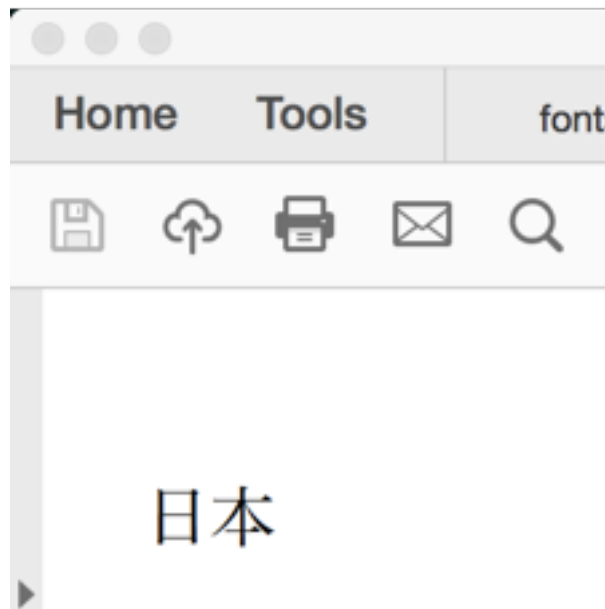

Fig. 2-4: **Embedding an Asian font**

Since Python 3 supports Unicode out of the box, you don't have to use Unicode characters directly like we did in the example above. You can just use the actual Kanji! Just for fun, I went ahead and updated the example to use some actual Kanji:

```python
# asian_font_demo2.py

from reportlab.lib.pagesizes import letter
from reportlab.pdfbase import pdfmetrics
from reportlab.pdfbase.cidfonts import UnicodeCIDFont
from reportlab.pdfgen import canvas


def asian_font_demo():
    my_canvas = canvas.Canvas("asian_font_demo2.pdf",
                              pagesize=letter)

    # Set a Japanese font
    pdfmetrics.registerFont(UnicodeCIDFont('HeiseiMin-W3'))
    my_canvas.setFont('HeiseiMin-W3', 16)

    # Find a word or phrase in Unicode to write out
    nippon = 'æ„›æf…'   # Love in Japanese

    my_canvas.drawString(25, 730, nippon)
    my_canvas.save()

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

You can also use TrueType fonts that have Asian characters. This is even easier to use than the **UnicodeCIDFont** that we had to use in the previous section. The ReportLab engineers did mention in their documentation that these kinds of fonts can take time to parse and that large subsets of the font will need to be embedded in your PDF for the characters to be displayed correctly. They also noted that ReportLab can also parse **.ttc** files, which is a variant of the TrueType extension, **.ttf**.

There are also the Noto and Source Han (Sans/Serif) fonts, which are both open source. Noto is from Google and aims to support all languages whereas Source Han Sans is a set of OpenType/CFF Pan-CJK fonts from Adobe.
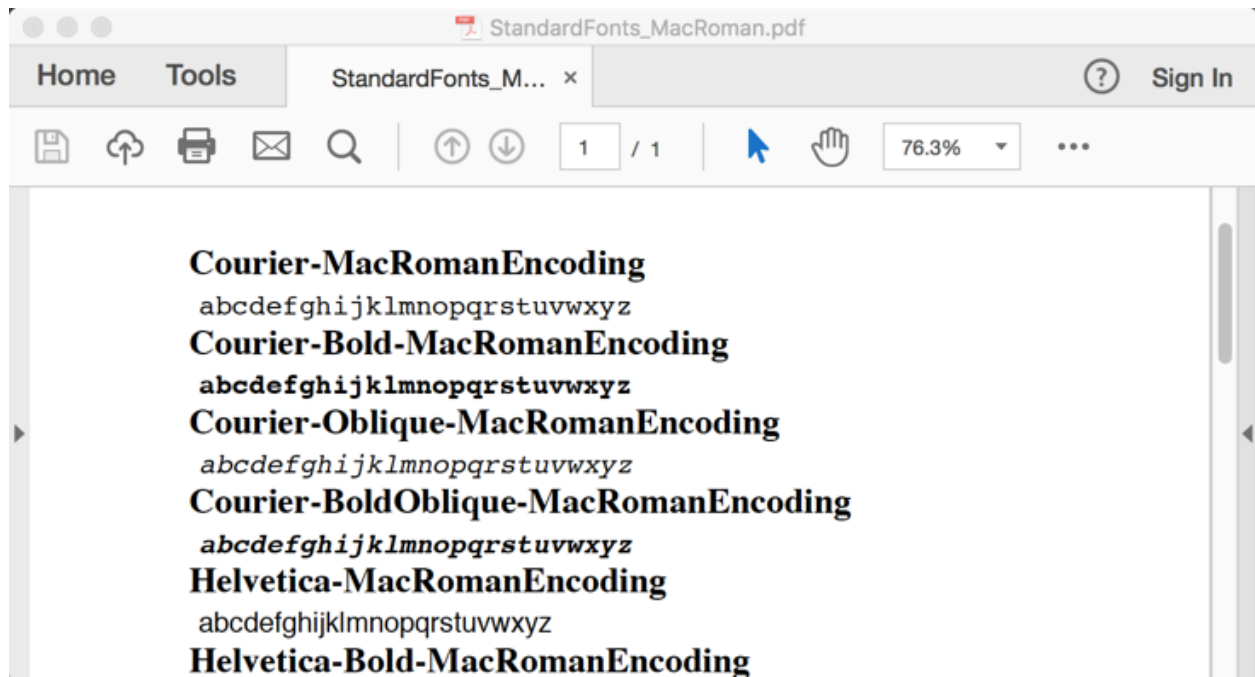
Switching Between Fonts

Fig. 2-5: Switching between fonts

We technically have already covered how to switch between fonts, but I didn't actually call it out in the previous examples. So I wanted to make sure my readers were quite clear on how to change fonts while generating their PDFs with ReportLab. Let's take a look at a simple demo:

```python
# font_switching.py

import string
import sys

from reportlab.pdfbase import pdfmetrics
from reportlab.pdfgen import canvas


def standardFonts():
    """
    Create a PDF with all the standard fonts
    """
    for enc in ['MacRoman', 'WinAnsi']:
        canv = canvas.Canvas(
                'StandardFonts_%s.pdf' % enc,
```

```python
            )
        canv.setPageCompression(0)

    x = 0
    y = 744
    for faceName in pdfmetrics.standardFonts:
        if faceName in ['Symbol', 'ZapfDingbats']:
            encLabel = faceName+'Encoding'
        else:
            encLabel = enc + 'Encoding'

        fontName = faceName + '-' + encLabel
        pdfmetrics.registerFont(pdfmetrics.Font(fontName,
                                    faceName,
                                    encLabel)
                )

        canv.setFont('Times-Bold', 18)
        canv.drawString(80, y, fontName)

        y -= 20

        alpha = "abcdefghijklmnopqrstuvwxyz"
        canv.setFont(fontName, 14)
        canv.drawString(x+85, y, alpha)

        y -= 20

    canv.save()

if __name__ == "__main__":
    standardFonts()
```

As mentioned earlier, Reportlab supports several fonts internally. You can think of them as standard or default fonts. The script above will create two PDFs: **StandardFonts_MacRoman.pdf** and **StandardFonts_WinAnsi.pdf**. As you can see, we just use a nested pair of for loops to extract the various fonts and register them with Reportlab. Then we called **setFont** with the selected font. From that point forward, ReportLab will use that font when it draws text. When you call **setFont** again, that will change your currently selected font to the one you specified and then that will be the font used. In other words, ReportLab always uses the last set font or the default until you explicitly set it to something else.

By the way, the difference between MacRoman and WinAnsi is that one was developed by Apple and

the other became a Microsoft proprietary character set. While they are identical for some character (32 - 126 of ASCII), they each have different distinct sets of control characters.

Wrapping Up

In this chapter we covered how to embed Type-1 and TrueType fonts in our PDFs using ReportLab. We also covered ReportLab's Asian font support. You should take some time and play around with these examples to make sure you fully understand how to embed fonts in your PDF. This is quite handy when you need to give your documents a unique look or you need to support your Asian customers.