Introduction to Python Email Library

Email messages look simple in an email client. But behind the scenes the client is doing a lot of work to make that happen! Email messages -- even messages with images and attachments -- are actually complicated text structures made entirely of readable strings!

The ***Simple Mail Transfer Protocol (SMTP)*** and ***Multipurpose Internet Mail Extensions (MIME)*** standards define how email messages are constructed. You *could* read the standards documentation and create email messages all on your own, but you don't need to go to all that trouble. The email built-in Python module lets us easily construct email messages.

We'll start by using the email.message.EmailMessage class to create an empty email message.

```
>>> from email.message import EmailMessage
>>> message = EmailMessage()
>>> print(message)
```

As usual, printing the message object gives us the string representation of that object. The email library has a function that converts the complex EmailMessage object into something that is fairly human-readable. Since this is an empty message, there isn't anything to see yet. Let's try adding the sender and recipient to the message and see how that looks.

We'll define a couple of variables so that we can reuse them later.

```
>>> sender = "me@example.com"
>>> recipient = "you@example.com"
```

And now, add them to the From and To fields of the message.

```
>>> message['From'] = sender
>>> message['To'] = recipient
>>> print(message)
From: me@example.com
To: you@example.com
```

Cool! That's starting to look a bit more like an email message now. How about a subject?

```
>>> message['Subject'] = 'Greetings from {} to {}!'.format(sender, recipient)
>>> print(message)
From: me@example.com
To: you@example.com
Subject: Greetings from me@example.com to you@example.com!
```

From, **To**, and **Subject** are examples of ***email header fields***. They're ***key-value pairs*** of labels and instructions used by email clients and servers to route and display the email. They're separate from the email's ***message body***, which is the main content of the message.

Let's go ahead and add a message body!

```
>>> body = """Hey there!
...
... I'm learning to send emails using Python!"""
>>> message.set_content(body)
```

Alright, now what does that look like?

```
>>> print(message)
From: me@example.com
To: you@example.com
```

```
Subject: Greetings from me@example.com to you@example.com!
MIME-Version: 1.0
Content-Type: text/plain; charset="utf-8"
Content-Transfer-Encoding: 7bit

Hey there!

I'm learning to send email using Python!
```

The message has a body! The **set_content()** method also automatically added a couple of headers that the email infrastructure will use when sending this message to another machine. Remember in an earlier course, when we talked about ***character encodings***? The ***Content-Type*** and ***Content-Transfer-Encoding*** headers tell email clients and servers how to interpret the bytes in this email message into a string. Now, what about this other header? What is MIME? We'll learn about that next!

Adding Attachments

Remember, email messages are made up completely of strings. When you add an attachment to an email, whatever type the attachment happens to be, it's encoded as some form of text. The ***Multipurpose Internet Mail Extensions (MIME)*** standard is used to encode all sorts of files as text strings that can be sent via email.

Let's dive in and break down how that works.

In order for the recipient of your message to understand what to do with an attachment, you need to label the attachment with a ***MIME type*** and ***subtype*** to tell them what sort of file you're sending. The ***Internet Assigned Numbers Authority (IANA)*** (iana.org) hosts a registry of valid MIME types. If you know the correct type and subtype of the files you'll be sending, you can use those values directly. If you don't know, you can use the Python **mimetypes** module to make a good guess!

```
>>> import os.path
>>> attachment_path = "/tmp/example.png"
>>> attachment_filename = os.path.basename(attachment_path)
>>> import mimetypes
>>> mime_type, _ = mimetypes.guess_type(attachment_path)
>>> print(mime_type)
image/png
```

Alright, that **mime_type** string contains the MIME type and subtype, separated by a slash. The **EmailMessage** type needs a MIME type and subtypes as separate strings, so let's split this up:

```
>>> mime_type, mime_subtype = mime_type.split('/', 1)
>>> print(mime_type)
image
>>> print(mime_subtype)
png
```

Now, finally! Let's add the attachment to our message and see what it looks like.

```
>>> with open(attachment_path, 'rb') as ap:
...     message.add_attachment(ap.read(),
...                            maintype=mime_type,
...                            subtype=mime_subtype,
...                            filename=os.path.basename(attachment_path))
...
>>> print(message)
Content-Type: multipart/mixed; boundary="===============5350123048127315795=="
```

```
--===============5350123048127315795==
Content-Type: text/plain; charset="utf-8"
Content-Transfer-Encoding: 7bit

Hey there!

I'm learning to send email using Python!

--===============5350123048127315795==
Content-Type: image/png
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="example.png"
MIME-Version: 1.0

iVBORw0KGgoAAAANSUhEUgAAASIAAABSCAYAAADw69nDAAAACXBIWXMAAAsTAAALEwEAmpwYAAAg
AElEQVR4nO2dd3wUZf7HP8/M9k2nKIJA4BCUNJKgNJWIBUUgEggCiSgeVhA8jzv05Gc5z4KHiqin
eBZIIBDKIXggKIeCRAhjQAqx4UiCARSt83uzDy/PzazTDZbwy4BnHde+9qZydNn97Pf5/uUIZRS
(...We deleted a bunch of lines here...)
wgAAAABJRU5ErkJggg==

--===============5350123048127315795==--
```

The entire message can still be serialized as a text string, including the image that we attached! The email message as a whole has the MIME type "multipart/mixed". Each ***part*** of the message has its own MIME type. The message body is still there as a "text/plain" part, and the image attachment is a "image/png" part. Cool!

Now, how do we *send* this email message? That's coming up!

Sending the Email Through an SMTP Server

As we called out, to send emails, our computers use the ***Simple Mail Transfer Protocol (SMTP)***. This protocol specifies how computers can deliver email to each other. There are certain steps that need to be followed to do this correctly. But, as usual, we won't do this manually; we'll send the message using the built-in smtplib Python module. Let's start by importing the module.

```
>>> import smtplib
```

With smtplib, we'll create an object that will represent our mail server, and handle sending messages to that server. If you're using a Linux computer, you might already have a configured SMTP server like postfix or sendmail. But maybe not. Let's create a smtplib.SMTP object and try to connect to the local machine.

```
>>> mail_server = smtplib.SMTP('localhost')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  (...We deleted a bunch of lines here...)
ConnectionRefusedError: [Errno 61] Connection refused
```

Oops! This error means that there's no local SMTP server configured. But don't panic! You can still connect to the SMTP server for your personal email address. Most personal email services have instructions for sending email through SMTP; just search for the name of your email service and "SMTP connection settings".

When setting this up, there are a couple of things that you'll probably need to do: Use a secure transport layer and authenticate to the service using a username and password. Let's see what this means in practice.

You can connect to a remote SMTP server using ***Transport Layer Security (TLS)***. An earlier version of the TLS protocol was called ***Secure Sockets Layer (SSL)***, and you'll sometimes see TLS and SSL used interchangeably. This SSL/TLS is the same protocol that's used to add a secure transmission layer to HTTP, making it HTTPS. Within the smtplib, there are two classes for making connections to an SMTP server: The ***SMTP class*** will make a direct SMTP connection, and the ***SMTP SSL class*** will make a SMTP connection over SSL/TLS. Like this:

```
>>> mail_server = smtplib.SMTP_SSL('smtp.example.com')
```

If you want to see the SMTP messages that are being sent back and forth by the smtplib module behind the scenes, you can set the debug level on the SMTP or SMTP_SSL object. The examples in this lesson won't show the debug output, but you might find it interesting!

```
mail_server.set_debuglevel(1)
```

Now that we've made a connection to the SMTP server, the next thing we need to do is authenticate to the SMTP server. Typically, email providers wants us to provide a username and password to connect. Let's put the password into a variable so it's not visible on the screen.

```
>>> import getpass
>>> mail_pass = getpass.getpass('Password? ')
Password?
>>>
```

The example above uses the getpass module so that passers-by won't see the password on the screen. Watch out, though; the **mail_pass** variable is still just an ordinary string!

```
>>> print(mail_pass)
It'sASecr3t!
```

Now that we have the email user and password configured, we can authenticate to the email server using the SMTP object's login method.

```
>>> mail_server.login(sender, mail_pass)
(235, b'2.7.0 Accepted')
```

If the login attempt succeeds, the login method will return a tuple of the SMTP status code and a message explaining the reason for the status. If the login attempt fails, the module will raise a SMTPAuthenticationError exception.

If you wrote a script to send an email message, how would you handle this exception?

Sending your message

Alright! We're connected and authenticated to the SMTP server. Now, how do we send the message?

```
>>> mail_server.send_message(message)
{}
```

Okay, well that last bit was pretty easy! We did the hard part first! The send_message method returns a dictionary of any recipients that weren't able to receive the message. Our message was delivered successfully, so send_message returned an empty dictionary. Finally, now that the email is sent, let's close the connection to the mail server.

```
>>> mail_server.quit()
```

And there you have it! We covered a lot in this lesson, so let's recap! First, we constructed an email message by using the built-in email module's EmailMessage class. Next, we added an attachment to our message with the help of the built-in mimetypes module. Finally, we connected to a SMTP server and sent the email using the smtplib module's 's SMTP_SSL class.

Did you have any idea all of this was happening behind a simple email message?

Introduction to Generating PDFs

Depending on what your automation does, you might want to generate a PDF report at the end, which lets you decide exactly how you want your information to look like.

There's a few tools in Python that let you generate PDFs with the content that you want. Here, we'll learn about one of them: ***ReportLab***. ReportLab has a **lot** of different features for creating PDF documents. We'll cover just the basics here, and give you pointers for more information at the end.

For our examples, we'll be mostly using the high-level classes and methods in the ***Page Layout and Typography Using Scripts (PLATYPUS)*** part of the ReportLab module.

Let's say that I have an awesome collection of fruit, and I want to create a PDF report of all the different kinds of fruit I have! I can easily represent the different kinds of fruit and how much of each I have with a Python dictionary. It might look something like this:

```
fruit = {
  "elderberries": 1,
  "figs": 1,
  "apples": 2,
  "durians": 3,
  "bananas": 5,
  "cherries": 8,
  "grapes": 13
}
```

Now let's take this information and turn it into a report that we can show off! We're going to use the **SimpleDocTemplate** class to build our PDF.

```
>>> from reportlab.platypus import SimpleDocTemplate
>>> report = SimpleDocTemplate("/tmp/report.pdf")
```

The **report** object that we just created will end up generating a PDF using the filename **/tmp/report.pdf**. Now, let's add some content to it! We'll create a title, some text in paragraphs, and some charts and images. For that, we're going to use what reportlab calls ***Flowables***. Flowables are sort of like chunks of a document that reportlab can arrange to make a complete report. Let's import some Flowable classes.

```
>>> from reportlab.platypus import Paragraph, Spacer, Table, Image
```

Each of these items (**Paragraph**, **Spacer**, **Table**, and **Image**) are classes that build individual elements in the final document. We have to tell reportlab what ***style*** we want each part of the document to have, so let's import some more things from the module to describe style.

```
>>> from reportlab.lib.styles import getSampleStyleSheet
>>> styles = getSampleStyleSheet()
```

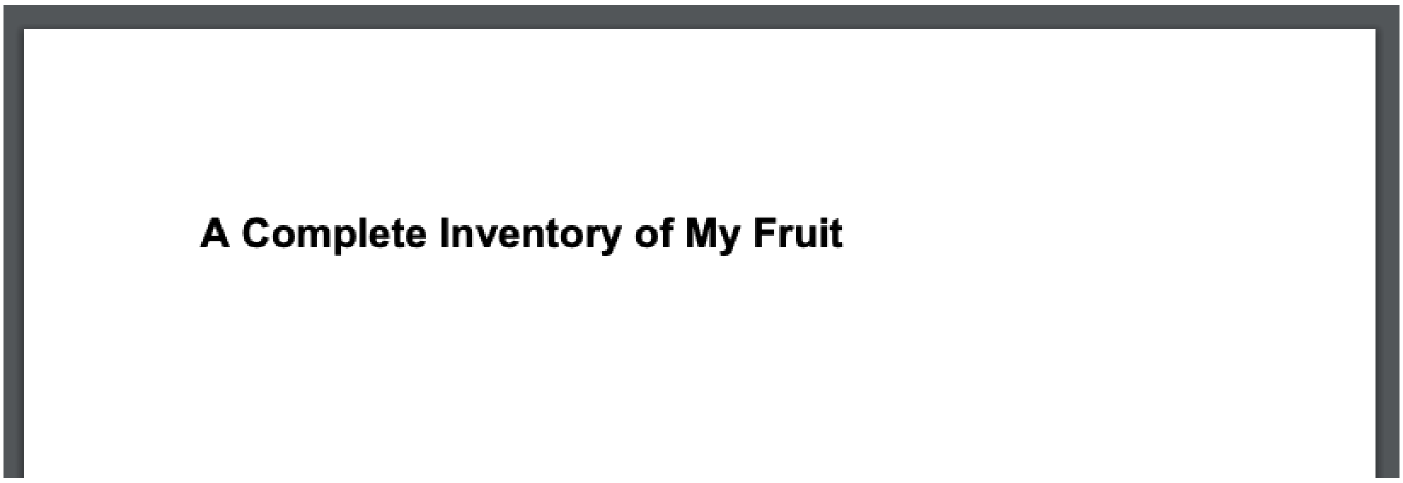
You can make a style all of your own, but we'll use the default provided by the module for these examples. The **styles** object now contains a default "sample" style. It's like a dictionary of different style settings. If you've ever written HTML, the style settings will look familiar. For example **h1** represents the style for the first level of headers. Alright, we're finally ready to give this report a title!

```
>>> report_title = Paragraph("A Complete Inventory of My Fruit", styles["h1"])
```

Let's take a look at what this will look like. We can build the PDF now by using the **build()** method of our report. It takes a list of Flowable elements, and generates a PDF with them.

```
>>> report.build([report_title])
```

Okay, now let's take a look at the PDF:



A Complete Inventory of My Fruit

It's not much, but it's a start!

Up next, we'll look into an interesting Flowable for our reports: Tables.

Adding Tables to our PDFs

Up to now, we've generated an extra simple PDF file, that just includes a title.

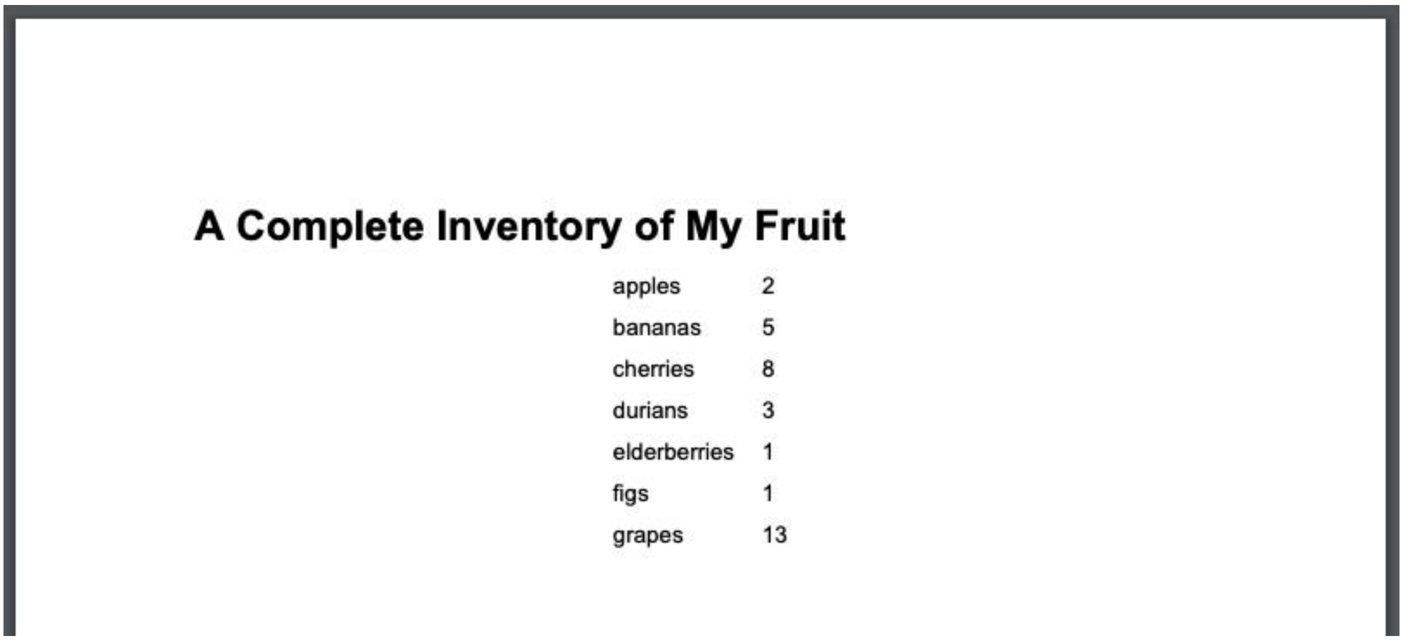
Let's spice this up by adding a ***Table***. To make a Table object, we need our data to be in a ***list-of-lists***, sometimes called a ***two-dimensional array***. We have our inventory of fruit in a dictionary. How can we convert a dictionary into a list-of-lists?

```
>>> table_data = []
>>> for k, v in fruit.items():
...   table_data.append([k, v])
...
>>> print(table_data)
[['elderberries', 1], ['figs', 1], ['apples', 2], ['durians', 3], ['bananas', 5], ['cherri
es', 8], ['grapes', 13]]
```

Great, we have the list of lists. We can now add it to our report and then generate the PDF file once again by calling the **build** method.

```
>>> report_table = Table(data=table_data)
>>> report.build([report_title, report_table])
```

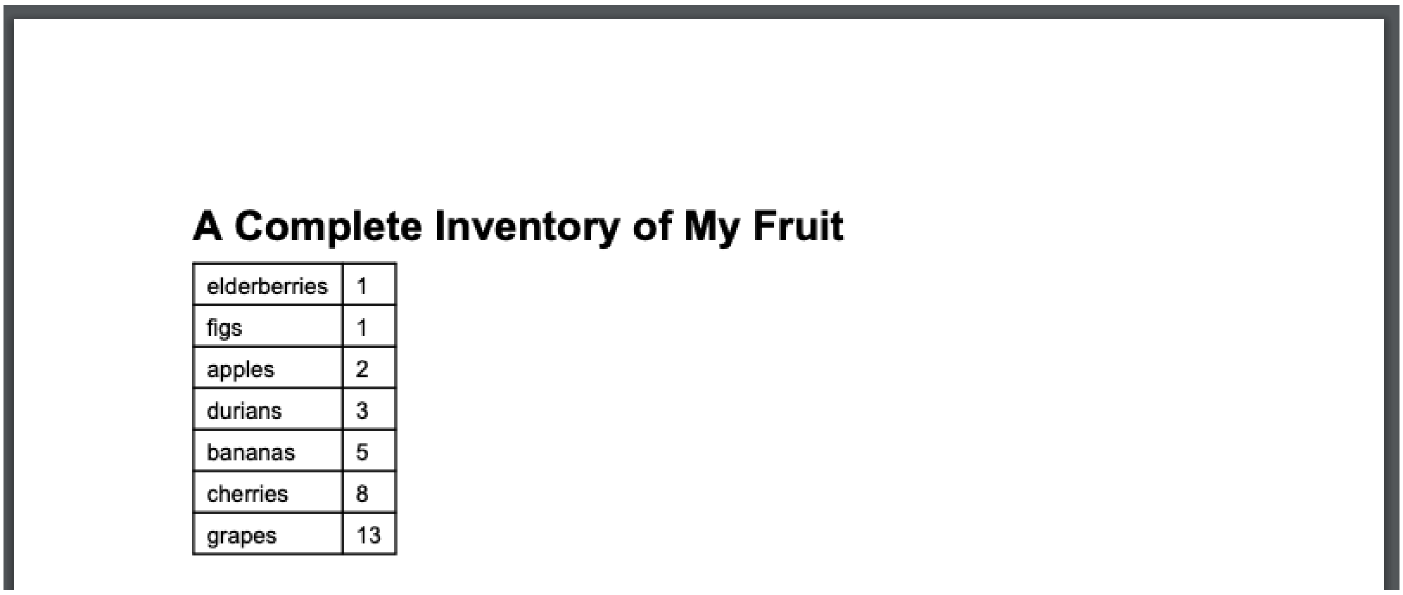
And this is how the generated report looks now:


A Complete Inventory of My Fruit

apples	2
bananas	5
cherries	8
durians	3
elderberries	1
figs	1
grapes	13

Okay, it worked! It's not very easy to read, though. Maybe we should add some style to **report_table**. For our example, we'll add a border around all of the cells in our table, and move the table over to the left. ***TableStyle*** definitions can get pretty complicated, so feel free to take a look at the documentation for a more complete idea of what's possible.

```
>>> from reportlab.lib import colors
>>> table_style = [('GRID', (0,0), (-1,-1), 1, colors.black)]
>>> report_table = Table(data=table_data, style=table_style, hAlign="LEFT")
>>> report.build([report_title, report_table])
```


A Complete Inventory of My Fruit

elderberries	1
figs	1
apples	2
durians	3
bananas	5
cherries	8
grapes	13

Much better! Up next, we'll look into making this more colorful by adding graphs to our reports.

Adding Graphics to our PDFs

Up to now, we've generated a report with a title and a table of data. Next let's add something a little more graphical. What could be better than a fruit pie (graph)?! We're going to need to use the ***Drawing*** Flowable class to create a ***Pie*** chart.

```
>>> from reportlab.graphics.shapes import Drawing
>>> from reportlab.graphics.charts.piecharts import Pie
>>> report_pie = Pie(width=3*inch, height=3*inch)
```

To add data to our **Pie** chart, we need two separate lists: One for data, and one for labels. Once more, we're going to have to transform our fruit dictionary into a different shape. For an added twist, let's sort the fruit in alphabetical order:

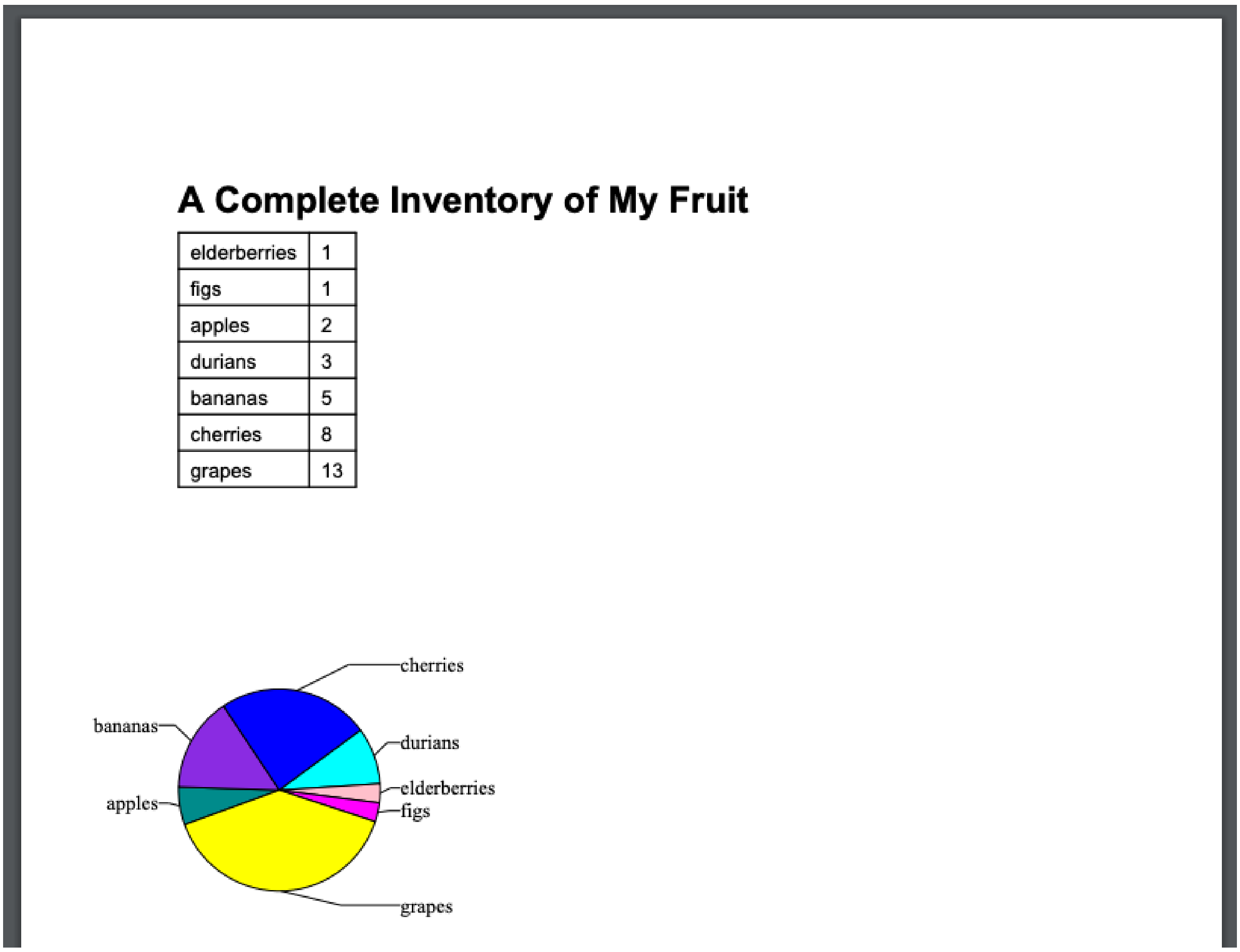
```
>>> report_pie.data = []
>>> report_pie.labels = []
>>> for fruit_name in sorted(fruit):
...   report_pie.data.append(fruit[fruit_name])
...   report_pie.labels.append(fruit_name)
...
>>> print(report_pie.data)
[2, 5, 8, 3, 1, 1, 13]
>>> print(report_pie.labels)
['apples', 'bananas', 'cherries', 'durians', 'elderberries', 'figs', 'grapes']
```

The **Pie** object isn't Flowable, but it can be placed inside of a Flowable ***Drawing***.

```
>>> report_chart = Drawing()
>>> report_chart.add(report_pie)
```

Now, we'll add the new Drawing to the report, and see what it looks like.

```
report.build([report_title, report_table, report_chart])
```



A Complete Inventory of My Fruit

elderberries	1
figs	1
apples	2
durians	3
bananas	5
cherries	8
grapes	13

Alright, and with that, you've seen a few examples of what we can do with the ReportLab library. There's a ton more things that can be done that we won't cover here. You'll want to refer to the ReportLab User Guide for more details on the features we've seen, and to see what else you can create with it.

By the way, the ReportLab User Guide is a PDF that is generated using reportlab! Cool, right?