When using PIL, we typically create **Image** objects that hold the data associated with the images that we want to process. On these objects, we operate by calling different methods that either return a new image object or modify the data in the image, and then end up saving the result in a different file.

For example, if we wanted to resize an image and save the new image with a new name, we could do it with:

1

2

3

4

5

from PIL import Image

im = Image.open("example.jpg")

new\_im = im.resize((640,480))

new\_im.save("example\_resized.jpg

In this case, we're using the resize method that returns a new image with the new size, and then we save it into a different file. Or, if we want to rotate an image, we can use code like this:

1

2

3

4

5

from PIL import Image

im = Image.open("example.jpg")

new\_im = im.rotate(90)

new\_im.save("example\_rotated.jpg")



This method also returns a new image that we can then use to create the new rotated file. Because the methods return a new object, we can even combine these operations into just one line that rotates, resizes, and saves:

# 

# Scale and convert images using PIL

Use the Python Imaging Library to do the following to a batch of images:

* Open an image
* Rotate an image
* Resize an image
* Save an image in a specific format in a separate directory
* p = {"search": "grey kitten",
* ...      "max\_results": 15}
* >>> response = requests.get("https://example.com/path/to/api", params=p)
* >>> response.request.url
* 'https://example.com/path/to/api?search=grey+kitten&max\_results=15'
* An alternative in that case is using the [***HTTP POST method***](https://tools.ietf.org/html/rfc7231#section-4.3.3). This method sends, or ***posts***, data to a web service. Whenever you fill a web form and press a button to submit, you're using the POST method to send that data back to the web server. This method tends to be used when there's a bunch of data to transmit.
* In our scripts, a POST request looks very similar to a GET request. Instead of setting the **params** attribute, which gets turned into a query string and appended to the URL, we use the **data** attribute, which contains the data that will be sent as part of the POST request.
* >>> p = {"description": "white kitten",
* ...      "name": "Snowball",
* ...      "age\_months": 6}
* >>> response = requests.post("https://example.com/path/to/api", data=p)

# Python Dictionaries and Upload to Running Web Service

## ****Introduction****

You’re working at a company that sells second-hand cars. Your company constantly collects feedback in the form of customer reviews. Your manager asks you to take those reviews (saved as .txt files) and display them on your company’s website. To do this, you’ll need to write a script to convert those .txt files and process them into Python dictionaries, then upload the data onto your company’s website (currently using Django).

* Use the Python OS module to process a directory of text files
* Manage information stored in Python dictionaries
* Use the Python requests module to upload content to a running Web service
* Understand basic operations for Python requests like GET and POST methods

# 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***](https://www.reportlab.com/opensource/). 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.

# Automatically Generate a PDF and send it by Email

## ****Introduction****

You work for a company that sells second hand cars. Management wants to get a summary of the amounts of vehicles that have been sold at the end of every month. The company already has a web service which serves sales data at the end of every month but management wants an email to be sent out with an attached PDF so that data is more easily readable.

## ****What you’ll do****

* Write a script that summarizes and processes sales data into different categories
* Generate a PDF using Python
* Automatically send a PDF by email

You'll have 90 minutes to complete this lab.===================

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?

1

2

3

4

5

6

>>> 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], ['cherries', 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.

1

2

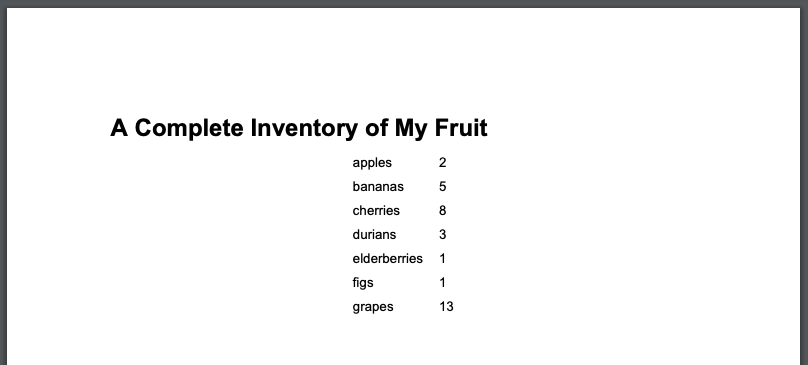
>>> report\_table = Table(data=table\_data)

>>> report.build([report\_title, report\_table])





And this is how the generated report looks now:



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.

1

2

3

4

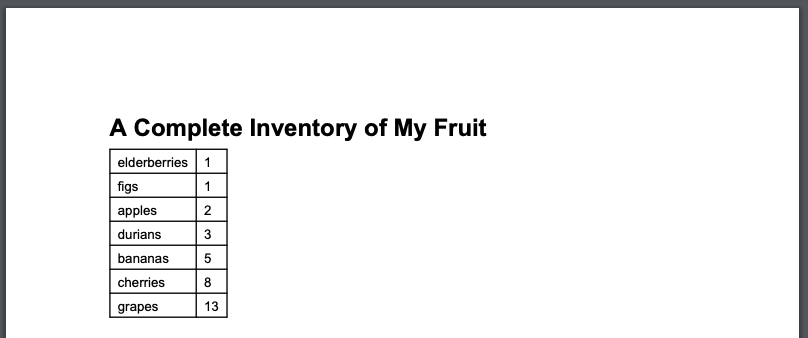
>>> from reportlab.lib import colors

>>> 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])





Much better! Up next, we'll look into making this more colorful by adding graphs to our reports. The **Pie** object isn’t Flowable, but it can be placed inside of a Flowable ***Drawing***.

1

2

>>> report\_chart = Drawing()

>>> report\_chart.add(report\_pie)





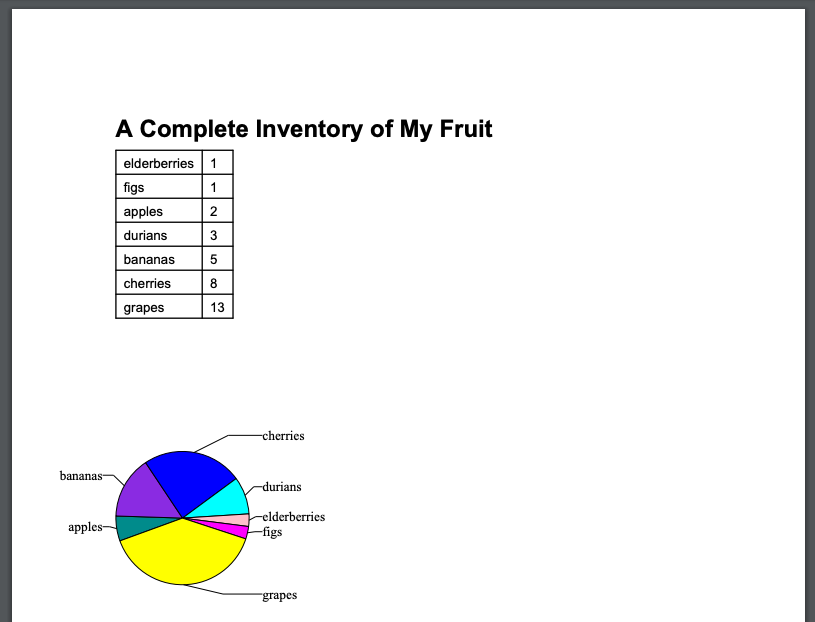
Now, we'll add the new Drawing to the report, and see what it looks like.

1

report.build([report\_title, report\_table, report\_chart])







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](https://www.reportlab.com/docs/reportlab-userguide.pdf) 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?

### Completed

# Automate updating catalog information

## ****Introduction****

You work for an online fruits store, and you need to develop a system that will update the catalog information with data provided by your suppliers. The suppliers send the data as large images with an associated description of the products in two files (.TIF for the image and .txt for the description). The images need to be converted to smaller jpeg images and the text needs to be turned into an HTML file that shows the image and the product description. The contents of the HTML file need to be uploaded to a web service that is already running using Django. You also need to gather the name and weight of all fruits from the .txt files and use a Python request to upload it to your Django server.

You will create a Python script that will process the images and descriptions and then update your company's online website to add the new products.

Once the task is complete, the supplier should be notified with an email that indicates the total weight of fruit (in lbs) that were uploaded. The email should have a PDF attached with the name of the fruit and its total weight (in lbs).

Finally, in parallel to the automation running, we want to check the health of the system and send an email if something goes wrong.

## ****What you’ll do****

* Write a script that summarizes and processes sales data into different categories
* Generate a PDF using Python
* Automatically send a PDF by email
* Write a script to check the health status of the system

You'll have 120 minutes to complete this lab.

**Passed** 80%