2. Handling Raw Data in Different Formats

Lan Du

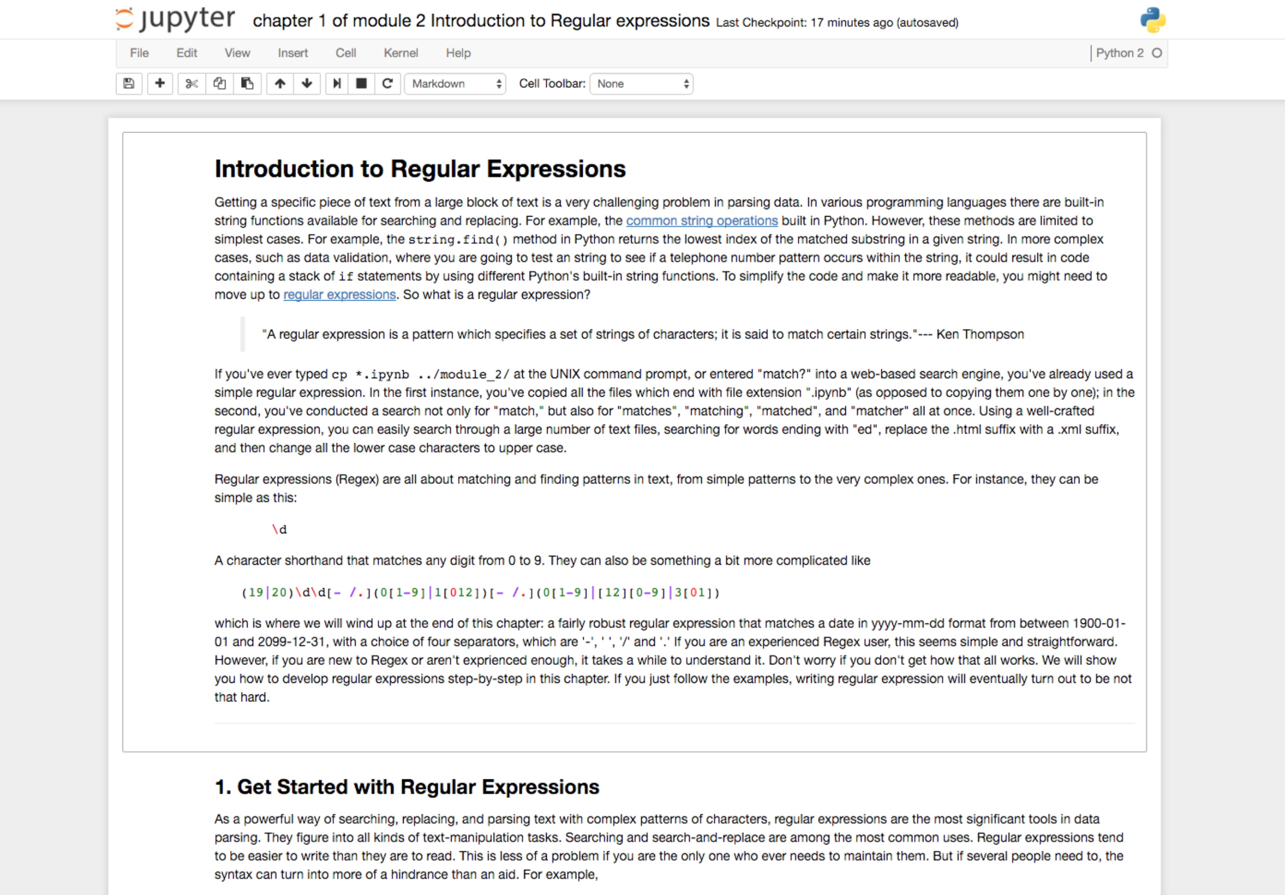
2. Handling Raw Data in Different Formats

Lan Du

Contents

1
Introduction to Regular Expressions

This chapter introduces the fundamentals of regular expressions under the umbrella of Python. You will learn by example how to develop regular expressions for dealing with street addresses, Roman numerals, phone numbers, and dates. Unlike Module 1, the text and Python code are all included in one JuPyteR Notebook file, as shown below.


jupyter
chapter 1 of module 2 Introduction to Regular expressions
Last Checkpoint: 17 minutes ago (autosaved)
File
Edit
View
Insert
Cell
Kernel
Help
Python 2 ○
Markdown
Cell Toolbar: None

Introduction to Regular Expressions

Getting a specific piece of text from a large block of text is a very challenging problem in parsing data. In various programming languages there are built-in string functions available for searching and replacing. For example, the common string operations built in Python. However, these methods are limited to simplest cases. For example, the `string.find()` method in Python returns the lowest index of the matched substring in a given string. In more complex cases, such as data validation, where you are going to test an string to see if a telephone number pattern occurs within the string, it could result in code containing a stack of `if` statements by using different Python's built-in string functions. To simplify the code and make it more readable, you might need to move up to regular expressions. So what is a regular expression?

> "A regular expression is a pattern which specifies a set of strings of characters; it is said to match certain strings."--- Ken Thompson

If you've ever typed `cp *.ipynb ../module_2/` at the UNIX command prompt, or entered "match?" into a web-based search engine, you've already used a simple regular expression. In the first instance, you've copied all the files which end with file extension ".ipynb" (as opposed to copying them one by one); in the second, you've conducted a search not only for "match," but also for "matches", "matching", "matched", and "matcher" all at once. Using a well-crafted regular expression, you can easily search through a large number of text files, searching for words ending with "ed", replace the .html suffix with a .xml suffix, and then change all the lower case characters to upper case.

Regular expressions (Regex) are all about matching and finding patterns in text, from simple patterns to the very complex ones. For instance, they can be simple as this:

```
\d
```

A character shorthand that matches any digit from 0 to 9. They can also be something a bit more complicated like

```
(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])
```

which is where we will wind up at the end of this chapter: a fairly robust regular expression that matches a date in yyyy-mm-dd format from between 1900-01-01 and 2099-12-31, with a choice of four separators, which are '-', ' ', '/' and '.' If you are an experienced Regex user, this seems simple and straightforward. However, if you are new to Regex or aren't exprienced enough, it takes a while to understand it. Don't worry if you don't get how that all works. We will show you how to develop regular expressions step-by-step in this chapter. If you just follow the examples, writing regular expression will eventually turn out to be not that hard.

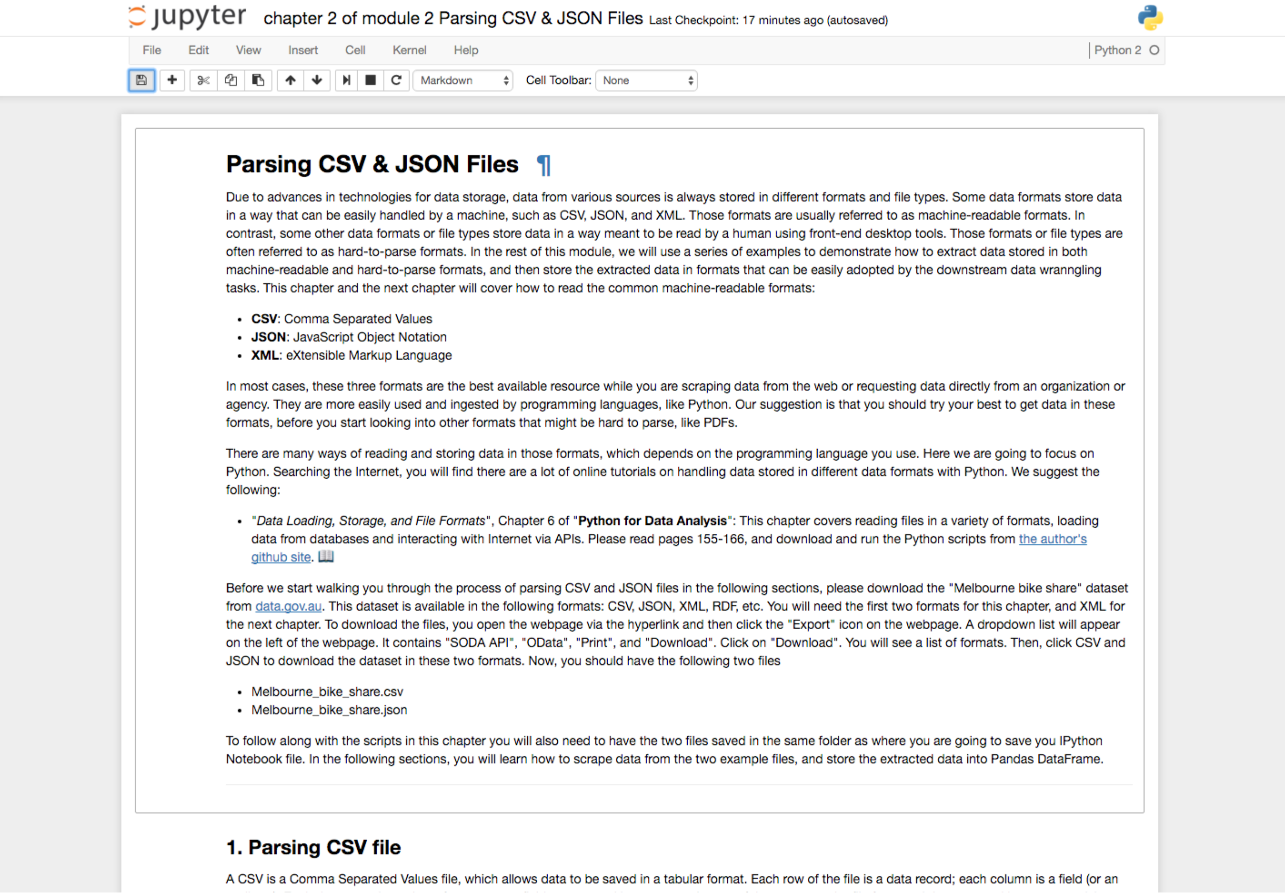
1. Get Started with Regular Expressions

As a powerful way of searching, replacing, and parsing text with complex patterns of characters, regular expressions are the most significant tools in data parsing. They figure into all kinds of text-manipulation tasks. Searching and search-and-replace are among the most common uses. Regular expressions tend to be easier to write than they are to read. This is less of a problem if you are the only one who ever needs to maintain them. But if several people need to, the syntax can turn into more of a hindrance than an aid. For example,

Please click here (https://www.alexandriarepository.org/wp-content/uploads/20151201030200/chapter-1-of-module-2.zip) to download the zip file that contains one notebook file. After unzipping the file, load the notebook with your Jupyter Notebook APP or with the command line. In order finish this chapter, you should read the text, run the code cell-by-cell and observe the output. You should also write Python code to finish the three exercises listed at the end of the notebook.

2
Parsing CSV & JSON Files

This chapter introduces how to scrape data stored in CSV (Comma Separated Values) and JSON (JavaScript Object Notation) formats. You will learn to use Pandas' built-in function to read CSV and JSON files, and transform the loaded data into a well tabulated form. Similar to Chapter 1, the text content and Python code are all included in an JuPyteR (iPython) Notebook file, as shown below.


jupyter
chapter 2 of module 2 Parsing CSV & JSON Files Last Checkpoint: 17 minutes ago (autosaved)
File
Edit
View
Insert
Cell
Kernel
Help
Python 2
Markdown
Cell Toolbar:
None

Parsing CSV & JSON Files ¶

Due to advances in technologies for data storage, data from various sources is always stored in different formats and file types. Some data formats store data in a way that can be easily handled by a machine, such as CSV, JSON, and XML. Those formats are usually referred to as machine-readable formats. In contrast, some other data formats or file types store data in a way meant to be read by a human using front-end desktop tools. Those formats or file types are often referred to as hard-to-parse formats. In the rest of this module, we will use a series of examples to demonstrate how to extract data stored in both machine-readable and hard-to-parse formats, and then store the extracted data in formats that can be easily adopted by the downstream data wranngling tasks. This chapter and the next chapter will cover how to read the common machine-readable formats:

- **CSV**: Comma Separated Values
- **JSON**: JavaScript Object Notation
- **XML**: eXtensible Markup Language

In most cases, these three formats are the best available resource while you are scraping data from the web or requesting data directly from an organization or agency. They are more easily used and ingested by programming languages, like Python. Our suggestion is that you should try your best to get data in these formats, before you start looking into other formats that might be hard to parse, like PDFs.

There are many ways of reading and storing data in those formats, which depends on the programming language you use. Here we are going to focus on Python. Searching the Internet, you will find there are a lot of online tutorials on handling data stored in different data formats with Python. We suggest the following:

- "*Data Loading, Storage, and File Formats*", Chapter 6 of "**Python for Data Analysis**": This chapter covers reading files in a variety of formats, loading data from databases and interacting with Internet via APIs. Please read pages 155-166, and download and run the Python scripts from the author's github site. 📖

Before we start walking you through the process of parsing CSV and JSON files in the following sections, please download the "Melbourne bike share" dataset from data.gov.au. This dataset is available in the following formats: CSV, JSON, XML, RDF, etc. You will need the first two formats for this chapter, and XML for the next chapter. To download the files, you open the webpage via the hyperlink and then click the "Export" icon on the webpage. A dropdown list will appear on the left of the webpage. It contains "SODA API", "OData", "Print", and "Download". Click on "Download". You will see a list of formats. Then, click CSV and JSON to download the dataset in these two formats. Now, you should have the following two files

- Melbourne_bike_share.csv
- Melbourne_bike_share.json

To follow along with the scripts in this chapter you will also need to have the two files saved in the same folder as where you are going to save you IPython Notebook file. In the following sections, you will learn how to scrape data from the two example files, and store the extracted data into Pandas DataFrame.

1. Parsing CSV file

A CSV is a Comma Separated Values file, which allows data to be saved in a tabular format. Each row of the file is a data record; each column is a field (or an

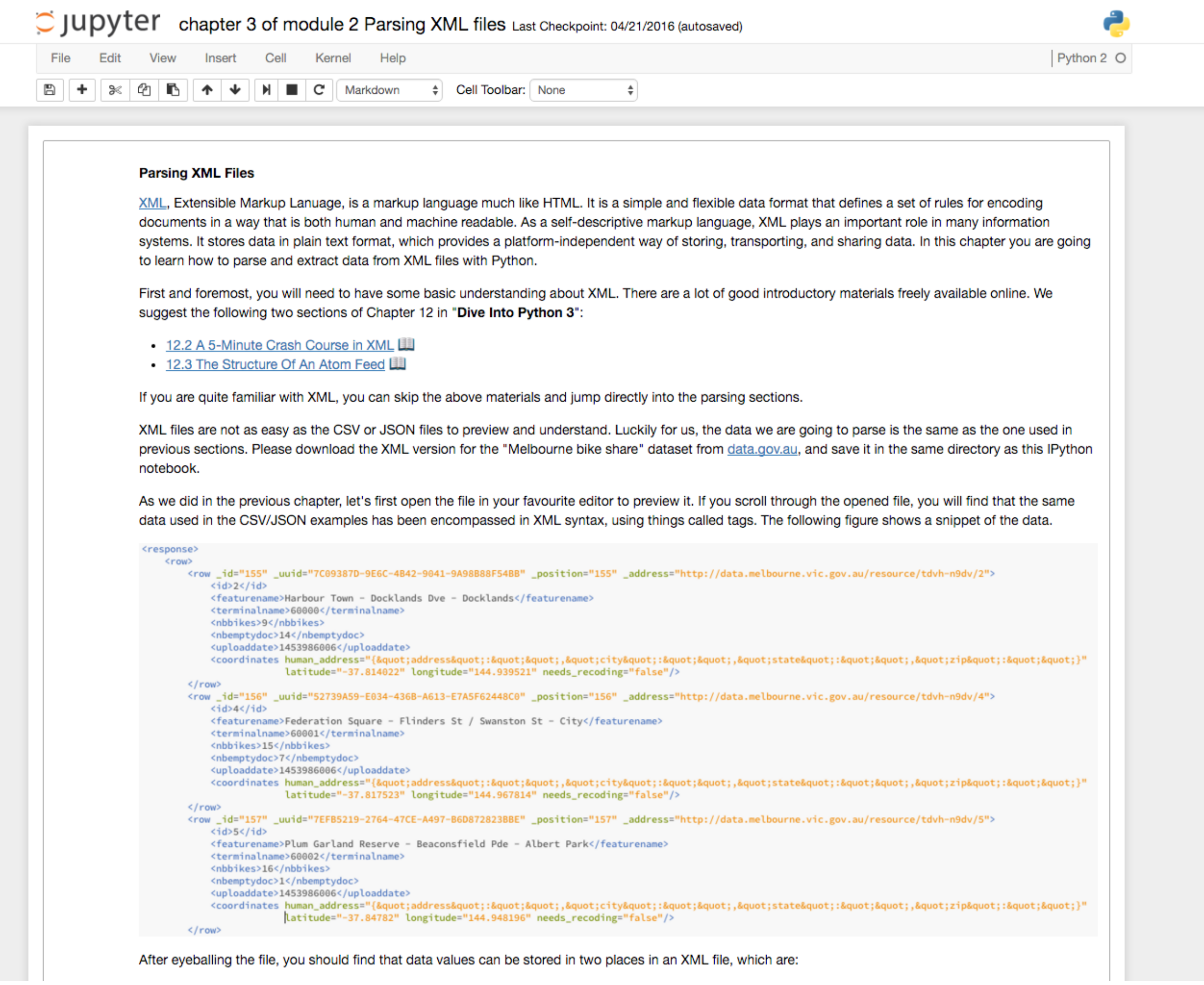
Please click here (https://www.alexandriarepository.org/wp-content/uploads/20151201042543/Chapter-2-of-module-2.zip) to download the zip file that contains the following list of files:

- chapter 2 of module 2 Parsing CSV & JSON Files.ipynb: the JuPyteR notebook containing all the Python code.
- csv1.png: an image showing what a CSV file looks like.
- elevations.json: a JSON dump used by the notebook
- json20.png: an image showing what a JSON file looks like.
- Melbourne_bike_share.csv: the CSV file to be parsed.
- Melbourne_bike_share.json: the JSON file to be parsed.

All the files listed above should be stored in the same folder so that you can run the notebook. Please read the text, run the code cell-by-cell and observe the output. You should also need to write Python code to finish the two exercises listed at the end of the notebook.

3 Parsing XML files

This chapter discusses a number of ways of scraping data from XML files. You will learn the fundamental structure of XML format, and the use of different Python libraries to load/explore XML files and extract data from them. A screenshot of the notebook is shown below:


jupyter
chapter 3 of module 2 Parsing XML files Last Checkpoint: 04/21/2016 (autosaved)
File
Edit
View
Insert
Cell
Kernel
Help
Python 2
Markdown
Cell Toolbar: None

Parsing XML Files

XML, Extensible Markup Lanuage, is a markup language much like HTML. It is a simple and flexible data format that defines a set of rules for encoding documents in a way that is both human and machine readable. As a self-descriptive markup language, XML plays an important role in many information systems. It stores data in plain text format, which provides a platform-independent way of storing, transporting, and sharing data. In this chapter you are going to learn how to parse and extract data from XML files with Python.

First and foremost, you will need to have some basic understanding about XML. There are a lot of good introductory materials freely available online. We suggest the following two sections of Chapter 12 in "**Dive Into Python 3**":

- 12.2 A 5-Minute Crash Course in XML 📖
- 12.3 The Structure Of An Atom Feed 📖

If you are quite familiar with XML, you can skip the above materials and jump directly into the parsing sections.

XML files are not as easy as the CSV or JSON files to preview and understand. Luckily for us, the data we are going to parse is the same as the one used in previous sections. Please download the XML version for the "Melbourne bike share" dataset from data.gov.au, and save it in the same directory as this IPython notebook.

As we did in the previous chapter, let's first open the file in your favourite editor to preview it. If you scroll through the opened file, you will find that the same data used in the CSV/JSON examples has been encompassed in XML syntax, using things called tags. The following figure shows a snippet of the data.

```
<response>
    <row>
        <row _id="155" _uuid="7C09387D-9E6C-4B42-9041-9A98B88F54BB" _position="155" _address="http://data.melbourne.vic.gov.au/resource/tdvh-n9dv/2">
            <id>2</id>
            <featurename>Harbour Town - Docklands Dve - Docklands</featurename>
            <terminalname>60000</terminalname>
            <nbbikes>9</nbbikes>
            <nbemptydoc>14</nbemptydoc>
            <uploaddate>1453986006</uploaddate>
            <coordinates human_address="{&quot;address&quot;:&quot;&quot;,&quot;city&quot;:&quot;&quot;,&quot;state&quot;:&quot;&quot;,&quot;zip&quot;:&quot;&quot;}"
                         latitude="-37.814022" longitude="144.939521" needs_recoding="false"/>
        </row>
        <row _id="156" _uuid="52739A59-E034-436B-A613-E7A5F62448C0" _position="156" _address="http://data.melbourne.vic.gov.au/resource/tdvh-n9dv/4">
            <id>4</id>
            <featurename>Federation Square - Flinders St / Swanston St - City</featurename>
            <terminalname>60001</terminalname>
            <nbbikes>15</nbbikes>
            <nbemptydoc>7</nbemptydoc>
            <uploaddate>1453986006</uploaddate>
            <coordinates human_address="{&quot;address&quot;:&quot;&quot;,&quot;city&quot;:&quot;&quot;,&quot;state&quot;:&quot;&quot;,&quot;zip&quot;:&quot;&quot;}"
                         latitude="-37.817523" longitude="144.967814" needs_recoding="false"/>
        </row>
        <row _id="157" _uuid="7EFB5219-2764-47CE-A497-B6D872823BBE" _position="157" _address="http://data.melbourne.vic.gov.au/resource/tdvh-n9dv/5">
            <id>5</id>
            <featurename>Plum Garland Reserve - Beaconsfield Pde - Albert Park</featurename>
            <terminalname>60002</terminalname>
            <nbbikes>16</nbbikes>
            <nbemptydoc>1</nbemptydoc>
            <uploaddate>1453986006</uploaddate>
            <coordinates human_address="{&quot;address&quot;:&quot;&quot;,&quot;city&quot;:&quot;&quot;,&quot;state&quot;:&quot;&quot;,&quot;zip&quot;:&quot;&quot;}"
                         latitude="-37.84782" longitude="144.948196" needs_recoding="false"/>
        </row>
```

After eyeballing the file, you should find that data values can be stored in two places in an XML file, which are:

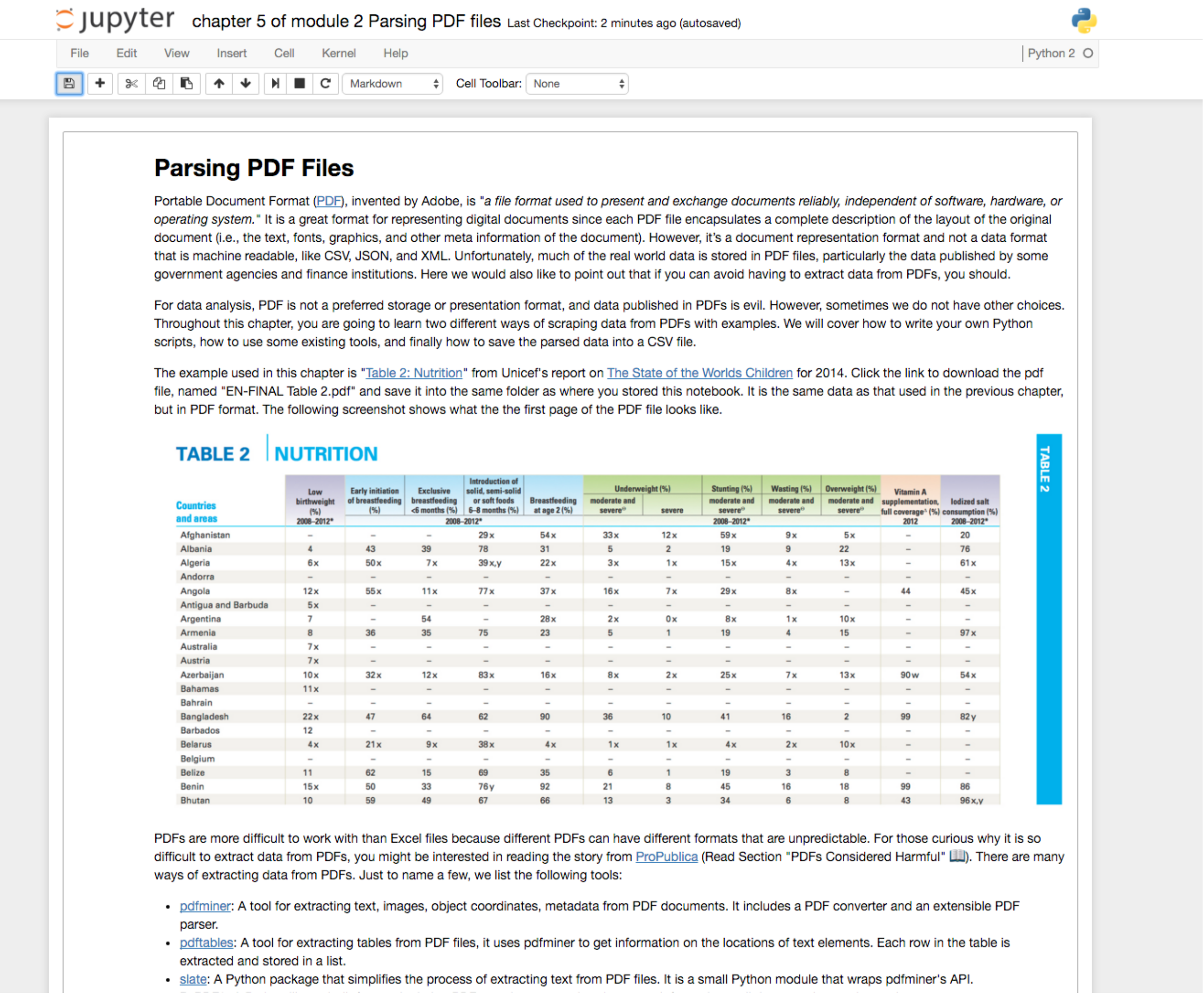
Please click here (https://www.alexandriarepository.org/wp-content/uploads/20151201030200/Chapter-3-of-module-2.zip) to download the zip file that contains the following list of files:

- chapter 3 of module 2 Parsing XML files.ipynb: an IPython notebook containing all the Python code.
- Melbourne_bike_share.xml: an XML file to be parsed.
- xml_example.png: an image showing what am XML file looks like.

All the files listed above should be stored in the same folder so that you can run the notebook. Please read the text, run the code cell-by-cell and observe the output. You should also need to write Python code to finish the two exercises listed at the end of the notebook.

4
Parsing PDF files

You might have found that extracting data from CSV, JSON, XML and even Excel files are not that hard. Then, how about PDF files? Can you extract tabular data stored in PDF files without scratching your head? PDF is a document representation format, rather than a proper data format. Unfortunately, scraping data from PDFs could not be avoidable since organisations, like government agencies and finance institutions prefer to use PDFs to release their data. This chapter uses an running example to show you the process of scraping simple tabular data from PDF pages. Here is a screenshot of the IPython notebook file:


jupyter
chapter 5 of module 2 Parsing PDF files Last Checkpoint: 2 minutes ago (autosaved)
File
Edit
View
Insert
Cell
Kernel
Help
Python 2
Markdown
Cell Toolbar:
None
TABLE 2

Parsing PDF Files

Portable Document Format (PDF), invented by Adobe, is "*a file format used to present and exchange documents reliably, independent of software, hardware, or operating system.*" It is a great format for representing digital documents since each PDF file encapsulates a complete description of the layout of the original document (i.e., the text, fonts, graphics, and other meta information of the document). However, it's a document representation format and not a data format that is machine readable, like CSV, JSON, and XML. Unfortunately, much of the real world data is stored in PDF files, particularly the data published by some government agencies and finance institutions. Here we would also like to point out that if you can avoid having to extract data from PDFs, you should.

For data analysis, PDF is not a preferred storage or presentation format, and data published in PDFs is evil. However, sometimes we do not have other choices. Throughout this chapter, you are going to learn two different ways of scraping data from PDFs with examples. We will cover how to write your own Python scripts, how to use some existing tools, and finally how to save the parsed data into a CSV file.

The example used in this chapter is "Table 2: Nutrition" from Unicef's report on The State of the Worlds Children for 2014. Click the link to download the pdf file, named "EN-FINAL Table 2.pdf" and save it into the same folder as where you stored this notebook. It is the same data as that used in the previous chapter, but in PDF format. The following screenshot shows what the the first page of the PDF file looks like.

TABLE 2 | NUTRITION

Countries and areas	Low birthweight (%)	Early initiation of breastfeeding (%)	Exclusive breastfeeding <6 months (%)	Introduction of solid, semi-solid or soft foods 6–8 months (%)	Breastfeeding at age 2 (%)	Underweight (%)		Stunting (%)	Wasting (%)	Overweight (%)	Vitamin A supplementation, full coverage^Δ (%)	Iodized salt consumption (%)
						moderate and severe^θ	severe	moderate and severe^θ	moderate and severe^θ	moderate and severe^θ		
	2008–2012*	2008–2012*				2008–2012*					2012	2008–2012*
Afghanistan	–	–	–	29 x	54 x	33 x	12 x	59 x	9 x	5 x	–	20
Albania	4	43	39	78	31	5	2	19	9	22	–	76
Algeria	6 x	50 x	7 x	39 x,y	22 x	3 x	1 x	15 x	4 x	13 x	–	61 x
Andorra	–	–	–	–	–	–	–	–	–	–	–	–
Angola	12 x	55 x	11 x	77 x	37 x	16 x	7 x	29 x	8 x	–	44	45 x
Antigua and Barbuda	5 x	–	–	–	–	–	–	–	–	–	–	–
Argentina	7	–	54	–	28 x	2 x	0 x	8 x	1 x	10 x	–	–
Armenia	8	36	35	75	23	5	1	19	4	15	–	97 x
Australia	7 x	–	–	–	–	–	–	–	–	–	–	–
Austria	7 x	–	–	–	–	–	–	–	–	–	–	–
Azerbaijan	10 x	32 x	12 x	83 x	16 x	8 x	2 x	25 x	7 x	13 x	90 w	54 x
Bahamas	11 x	–	–	–	–	–	–	–	–	–	–	–
Bahrain	–	–	–	–	–	–	–	–	–	–	–	–
Bangladesh	22 x	47	64	62	90	36	10	41	16	2	99	82 y
Barbados	12	–	–	–	–	–	–	–	–	–	–	–
Belarus	4 x	21 x	9 x	38 x	4 x	1 x	1 x	4 x	2 x	10 x	–	–
Belgium	–	–	–	–	–	–	–	–	–	–	–	–
Belize	11	62	15	69	35	6	1	19	3	8	–	–
Benin	15 x	50	33	76 y	92	21	8	45	16	18	99	86
Bhutan	10	59	49	67	66	13	3	34	6	8	43	96 x,y

PDFs are more difficult to work with than Excel files because different PDFs can have different formats that are unpredictable. For those curious why it is so difficult to extract data from PDFs, you might be interested in reading the story from ProPublica (Read Section "PDFs Considered Harmful" 📖). There are many ways of extracting data from PDFs. Just to name a few, we list the following tools:

- pdfminer: A tool for extracting text, images, object coordinates, metadata from PDF documents. It includes a PDF converter and an extensible PDF parser.
- pdftables: A tool for extracting tables from PDF files, it uses pdfminer to get information on the locations of text elements. Each row in the table is extracted and stored in a list.
- slate: A Python package that simplifies the process of extracting text from PDF files. It is a small Python module that wraps pdfminer's API.

Please click here (https://www.alexandriarepository.org/wp-content/uploads/20160502233906/Chapter-5-of-module-21.zip) to download the zip file that contains the following list of files:

- chapter 5 of module 2 Parsing PDF files.ipynb: an IPython notebook containing all the Python code.
- EN_FINAL_Table 2.pdf: a pdf file to be parsed.
- en_final_table_2.txt: a text file given by pdf2txt.py.
- en_final_table_2_1.csv: an output file.
- en_final_table_2_2.csv: am output file.
- EN_FINAL_Table_2_page_1.png: an image showing what a PDF page looks like. It is a screenshot of the first page of EN_FINAL_Table 2.pdf.

All the files listed above should be stored in the same folder so that you can run the notebook. Please read the text, run the code cell-by-cell and observe the output. You should also need to write Python code to finish the one exercise listed at the end of the notebook.