Python Web Scraping Tools:

A Survey Of The Landscape

Jon Reiter

April 20, 2018

Web Scraping: ?

- Wikipedia:
  - Web scraping is a technique in which a computer program extracts data from human-readable output coming from websites.

- Not discussing legality / morality / etc
  - Everything here comes from sites which explicitly allow/encourage automated access.

Python For Scraping

- Lots of packages
- Python handles complex text operations well
  - This is one of python's selling points
- Performance is (almost) irrelevant
  - Network and browser limited
  - Language, interpreter, etc do not matter

Web Content Types

What Content? How Written?	HTML	HTML / JavaScript / Complex Stuff
Well Formed	Text Parser	Browser Needed
Poorly Formed / Invalid / Real	Clever Text Parser	Browser With Forgiving Parser

What Is Complex?

- Not:
  - HTML, Text
  - Forms, Old-Style Links
- Complex:
  - JavaScript
  - CSS3 (some)
- Anything that doesn't work absent a "good" browser

Scraping Some Simple Text:

- Goal: Grab a field
- Example Page 1
- Code 1

XPath

- Data is in a tree
- "/" separators with "//" as a wildcard
- <img src="xyz">
  - //img/@src="xyz"
- some text
  - //a/text() = "some text"

Scraping A Link

- Basically the same
- XPath is now: //a[text()="link"]/@href
- This is a huge subject, separate talk
  - or just use stackoverflow
- Example Page 2
- Code 2

Invalid Content

- Goal: grab links from a real website
- Add beautifulsoup
- Example Page 3
- Code 3

Extacting Fields

- Goal: grab a field and do something with it
  - Use object "id"
  - Parse the text a bit
  - Build data structure

- Example Page 4

- Code 4

JavaScript

- Goal: click a JavaScript button
- New tools:
  - selenium
  - Chrome (or Firefox, ...)
  - Driver between python and browser
- Example Page 5
- Code 5

Downloads

- Goal: download a file via JS button
- This is not part of the browser
  - Native OS window, somehow
- Example Page 6
- Code 6

Browser: Key Points #1

- Browser is our parser
- Which browser matters
- Importance of which one:
  - proportional to complexity of page
  - inversely proportional to quality of page

Browser: Key Points #2

- Mild JavaScript, no errors: doesn't matter
- Funky JavaScript or errors: matters
- Semi-valid JavaScript, dynamic HTML, etc: hmm
  - Sometimes only 1 browser works
  - Or they both work, but differently

Headless Browsing

- Automated browser doesn't *need* a window
- Same API but don't draw anything
  - "not visible" errors but nothing visible
- Chrome and Firefox support this
- Let's try Firefox
  - With Head
  - Headless

Containers and Cloud

- You can run this stuff in containers
- Possible to run in the cloud
- This is bleeding edge
  - odd workarounds needed
  - no packages for some components

Use Case / Package Mapping

-

Browser: Review

- Key Point: if pages break Firefox or Chrome there is 0 chance you can write this yourself
  - Even if you could...maintenace nightmare
- Browser code $\approx$ basic parser code
  - Con: Environment heavier, testing harder
  - Pro: Better coverage

Practical Package Mapping

- Just always use selenium + Chrome / Firefox

Summary

- Casual users: learn selenium
  - ignore 99% of hype around scraping tools
  - they are useful...but just wrap a browser
- Learn XPath: needed for (almost) every solution
- Serious Users; learn selenium
  - keep Chrome and Firefox up to date
    * Release notes for the drivers are scary: "fixed: log all command line arguments" doesn't sound like a mature piece of software!

Using Just The Standard Library

- You can, sort of
- But some:
  - Error handling is your job
  - Retries are your job
  - Redirects are your job
- This gets tiring quickly
- requests, urllib, etc help a bit
- Recall: performance doesn't matter