
Scrape Documentation

Release 0.1a3

Yarko Tymciurak

October 29, 2015

1	[S]crape	Overview	3
1.1	Context		3
1.2	Modes of Operation		3
1.3	Knowledge You Should Have		4
1.4	[S]crape	Shell	4
2	[S]crape	Installation	7
2.1	Installing [S]crape		7
3	Tutorials		9
3.1	Introduction to [S]crape		9
3.2	Developing a Project		14
3.3	Introduction to [S]crape		22
3.4	Developing a Project		22
4	Copyright Notice		25
5	Overview		27
6	Installation		29
7	Tutorials		31
8	Alternatives		33
9	Copyright Notice		35
10	Work in Progress		37
11	Indices and tables		39

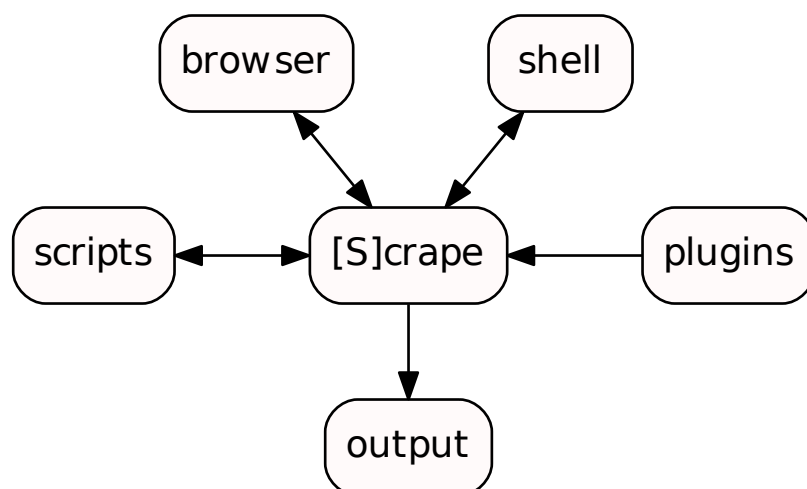
[S]^{crape} is a tool developed to help researchers extract selective data from web publications. It is particularly useful for serial web publications which have similar structure over many issues. You interactively develop a selection and extraction set of commands, and run them across a series of issues, generating output (JSON or CSV).

[S]crape Overview

1.1 Context

[S]crape

operates within the following environment:



1.2 Modes of Operation

You operate [S]^{crape} in one of two ways:

- interactively;
- batch;

[S]^{crape} starts and opens targets through a browser¹, and gets its data from that browser.

¹ [S]^{crape} uses a [Selenium Client Driver](#) to run your browser. At this time [S]^{crape} only supports Firefox.

You can interactively highlight sections in the browser and scrape will give you unambiguous code to select that data. As with a *Google* search, skill will help make the search term (xpath or css selector) more general, yet specific enough to return your desired result.

Inspect results as you work interactively until you are satisfied and then save a range of commands from your history to develop your script.

At anytime you can load and run scripts against an opened target. Thus you build up a complete script incrementally. In batch mode, you can automatically run it over a pattern or list of targets.

You can test your scripts interactively in headless mode (that is, without a browser). You can also run batch either with a browser, or headless.

1.3 Knowledge You Should Have

You should have a general understanding of *HTML* and *CSS* structure and form. You don't need to know much, but you should be able to understand and recognize what you are looking at when looking at small portions of web page source, and have an understanding of what type of thing you are trying to extract, i.e. path, attribute, or text.

You will need some basic understanding of *XPATH* syntax and *CSS Selectors* as you will be using these to describe what you are looking for. When manually highlighting something in your browser, [S]^{crape} will return an *XPATH*. Often a *CSS selector* is both shorter and more accurately selective. [S]^{crape} allows you to view context near your selection. This makes it easy to pick a different form of selector and test it before saving it to your script.

1.4 [S]^{crape} Shell

In interactive use, [S]^{crape} is similar to a typical command shell, such as *sh* or *bash*, or *cmd* on Windows. In command interpreters, there are typically built-in commands and a way to execute external commands. Shells also provide variables, and some sort of program control.

[S]^{crape} has a rich set of built-in commands, and allows calling external commands through your system's shell. You can also add built-in commands by writing extensions to [S]^{crape} in Python (*plugins*).

Since [S]^{crape} outputs tables ², variable names are like table column names. This means every variable in [S]^{crape} is a list (you can think of them as arrays), and every table an associative array of variables. In fact, you can save the result of your [S]^{crape} as either *csv*, *json* or *yaml*. There are other important kinds of variables in [S]^{crape}.

vars Output variables are the normal variables, and are used to specify output table column names.

local Local variables are similar to output variables, only they are omitted from tables. These are used for intermediate results. Local variables have scope per output table.

global These variables persist across output table changes.

[S]^{crape} is least like shells in that there is no familiar loop control. This simplifies traversing an *HTML* tree and extracting data. Instead of looping, you traverse to locations in the *XPATH* tree of the input file. We refer to selected (current) *XPATH* locations as *nodes*. Typical [S]^{crape} operation involves traversing a document's tree, extracting selected content from those nodes, and repeating. In place of program control, you control which nodes you search from. Multiple nodes can be active (for example all the list items of some part of the document), so scripts tend to be rather short. Some general control mechanisms [S]^{crape} provides are:

² [S]^{crape} was initially designed to output CSV, but this is a bit too restricting. For one thing, to change the view of the data (the order of way the data is populated into columns, the number and contents of tables) one would need to re-scrape the source. This is why you have a choice of saving variables as JSON or YAML also. Then, you could rebuild, re-shape your tables from your saved data source.

root Normally, navigation through the document is incremental. This sets the root of the tree to the starting `<html>` tag. When the root of the document tree is set, it's children are the active children, so in this case, normally `<head>` and `<body>` tags will be *current* starting nodes.

body This resets the root node to the `<body>` tag.

grab `[S]crape` opens a browser when it starts, and communicates with it. `grab` gets a highlighted region from your browser, giving you an `xpath` to it.

A majority of `[S]crape` commands involve selecting a node using an `xpath` selector, a `css-selector`, or a combination of path and text search. The remaining commands deal with interactive use (history, view variables, run scripts, save or load scripts), and outputting results (tables).

[s]crape

Installation

[s]crape

requires the following:

- a recent version of [python 2.7](#)
- a recent version of [Firefox](#)¹

Installing [S]^{crape} will install or upgrade the following python libraries:

- argparse
- lxml
- cssselect
- PyYAML
- selenium
- [S]^{crape}'s version of envoy
- a [S]^{crape} plugin library

Installation requires you have a compiler on your machine.

For Linux systems, this should already be the case.

For Macintosh OS/X systems, download Xcode free from the [Mac App Store](#) (also install the command line tools). Alternatively, you may be able to install just the command-line tools (see <https://github.com/kennethreitz/osx-gcc-installer> - we have not tried this).

For Windows platforms, this is not a straightforward process. See the MS Windows section at <http://lxml.de/installation.html>.

2.1 Installing [s]crape

¹ Firefox is the only browser officially supported for [S]^{crape}. As an alternative, you may try a current version of [Chrome](#), but note that you will need to download a [chrome-webdriver](#). For some combinations of versions of Chrome, chrome-webdriver and selenium, timeouts didn't properly work. For some medical journal sites with continuous stream advertising, Chrome would not respond (would never return when called from scrape).

Prerequisites:

- [python 2.7](#)
- a recent version of [Firefox](#)

Recommended:

- [virtualenv](#)
- [pip](#)

I suggest you use python's `virtualenv`, particularly your first time with `[S]crape` (see [virtualenv](#)).

This will ensure you have an isolated, clean python install of `[S]crape` to start. Once you have this working, you may consider installing this your system's python site-libraries.

To properly use `virtualenv`, you'll need `pip`. Ensure you have `pip` installed:

```
$ which pip
```

If you don't have `pip` installed, then install it:

```
$ easy_install pip
```

If you do have `pip`, be sure it's up-to-date:

```
$ pip install --upgrade pip
```

Todo

Have yet to debug the `scrape.gz` install file (installation does not mirror `setup.py`).

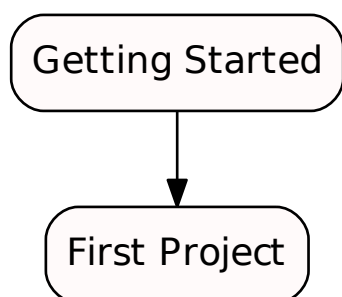
Now, install the current version of `[S]crape`. Currently, you must do this from sources. Clone a copy of `[S]crape` and run `setup.py`:

```
$ hg clone ssh://hg@bitbucket.org/yarko/scrape
$ cd scrape
$ python setup.py install
```

Tutorials

Select the tutorials which are appropriate for what you want to do.

To start, I recommend you follow the installation checkout and tutorial on this page, followed by the example PyCon project. These brief tutorials will introduce the concepts and strategies for using scrape, as well as give an overview of some of the most useful commands.



3.1 Introduction to [S]^{crape}

3.1.1 Getting Started

Overview

Introduction

We'll start by making sure you have [S]^{crape} installed.

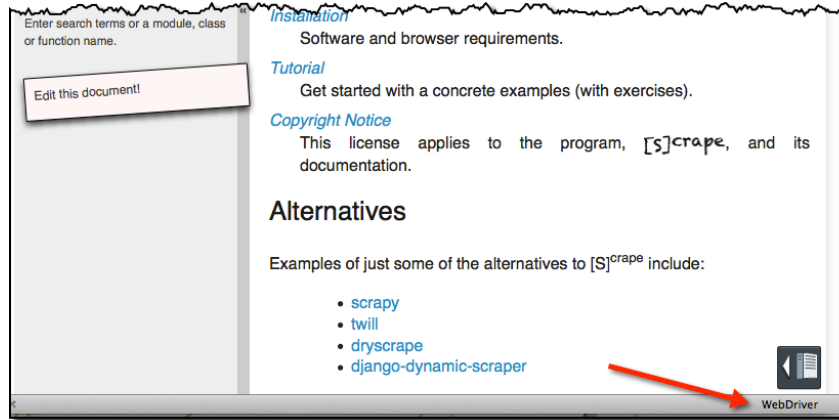
To start, I'll assume you have installed [S]^{crape} in a `virtualenv`.

Activate the virtual environment where you have installed [S]^{crape}, and run `scrape`:

```
$ scrape http://scrape.readthedocs.org
```

This should start [S]^{crape} and open its documentation in Firefox ¹.

¹ Be sure you've installed [S]^{crape} and a current Firefox browser.



- your Firefox should have “WebDriver” displayed in the lower-right;
 - this indicates that this Firefox is being controlled from [S]crape.
- you should see a log of the plugins registered (scrape comes distributed with one - affiliations);
 - if you don’t see `scrape: INFO - ...registering plugins:` in your [S]crape shell, then likely something is incomplete in your installation. You can continue with the exercises, but you will need to install plugins when you need them.

```
scrape: INFO - ...registering plugins:
scrape: INFO - affiliations
Scrape to [yaml, json or csv] through a thin layer over
the lxml library, python, and your shell. Type ^D to exit.
[S]crape >>>
```

At the [S]crape >>> prompt type the following:

```
[S]crape >>> help
```

```
[S]crape >>> help (nv builds)
Documented commands (type help <topic>):
EOF      find      getnext   history   populous  show
attrib   OS/X     find_by_text  getparent json      r         sparse
base     find_class  getpath   li        root      table
body     findall    getprevious list      run       tags
clear    findclass  global    load      save      tail
content  findtext   grab      local     scrape    text
cssselect get_element_by_id headless  nodes     search    text_content
current  getbyid    help      notheadless set        var
doc      getchildren hi         open      shell     yaml

Miscellaneous help topics:
complete
gin demo from Phil when available
Waiting
[S]crape >>>
```

[S]crape gives you access to an HTML file or web page. It does this by parsing your web page into a *tree* of HTML *nodes*. You then traverse the tree of nodes, scraping the information you want from a selected nodes.

[S]crape starts by setting the root node of your HTML page² to the `<html>` node.

Let’s show the contents of the current node:

² You can easily set to the root of the document at any time to either the entire document, or the body - see `help doc` and `help body`.

```
[S]crape >>> show node
```

You should see the source for the two subnodes (children) of the `<html>` tag, the `<head>` and `<body>` tags of the [S]^{crape} document page. This is the content of the document, rooted at `html` ³.

Just to confirm, lets count the current number of selected nodes:

```
[S]crape >>> nodes
```

When you will be looking at larger, more verbose selections, it can also be helpful to review just the tags of the selected nodes:

```
[S]crape >>> tags
```

The general starategy for using [S]^{crape} is:

- select a scrape target (a web page);
- declare a table name (which will hold a set of variables);
- navigate the web page's tree;
- declare a variable to collect information;
- capture the desired information;
- repeat as desired;
- save a table to a file;
- lastly, save your interactive commands to a script to run later.

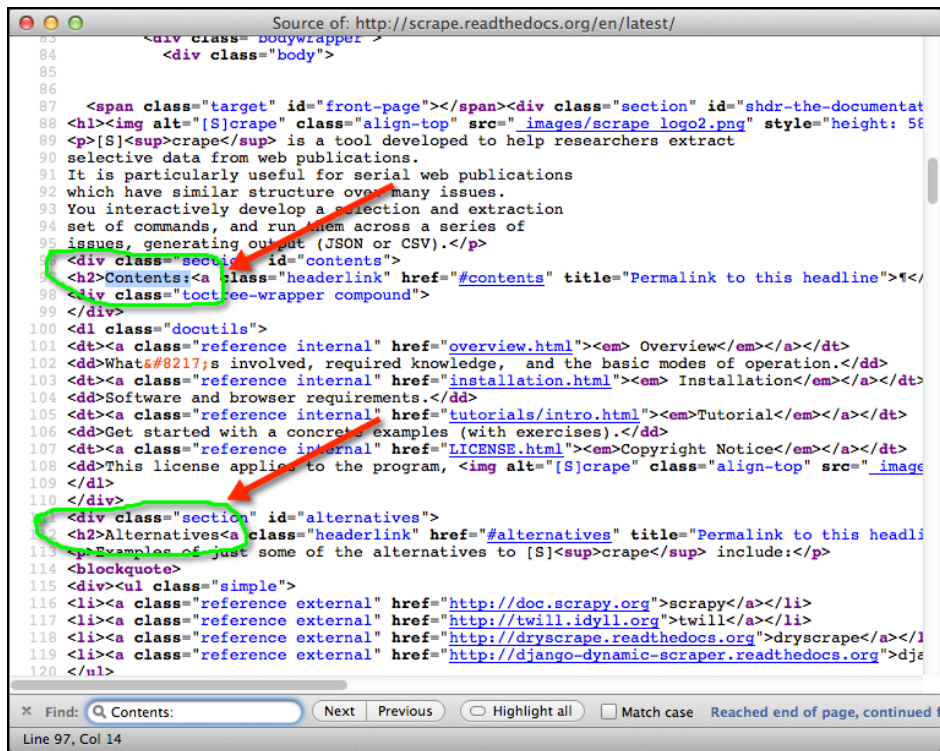
When looking a [S]^{crape} target, it's useful to also open the source view. When you are starting with a new page, you can easily search the various tags and attributes of the `html` elements.

Do that now - right-click on your webpage, and select `View Page Source`.

For this tutorial, we'll save the headers to develop an outline for this page.

The outline we'd like to make consists of the headers, `Contents:`, `Alternatives`, and so forth. In your source window, search for `Contents:`.

³ Note that when you look at an empty page (`about:blank`), scrape will create a minimal parse tree for you (`<html><head/><body/></head>`).



Contents: is in an `<h2>` tag, as is Alternatives; this looks like a reasonable target for our scrape.

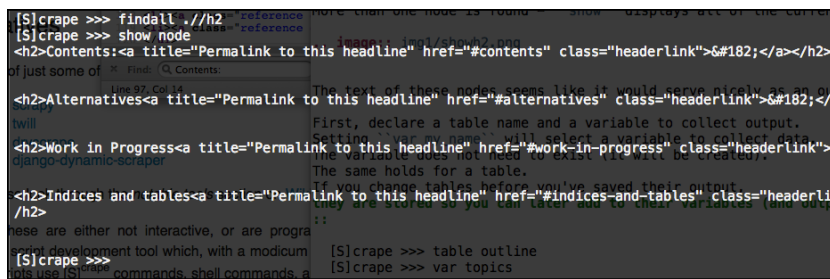
`[S]crape` provides a simplified interface to the `libxml2` library, so that most of the information you will find about `xpath` selectors and `cssselectors` will work as you expect. `[S]crape` also combines, extends and adds other commands for interactive use. For example, `find_by_text` will search nodes selected by an `xpath` expression for a string.

You might like view http://www.w3schools.com/xpath/xpath_syntax.asp for reference during this tutorial.

Let's find the subheadings on our target page to see if this will give us the page outline we'd like:

```
[S]crape >>> findall './h2'
[S]crape >>> show node
```

This should find all the `<H2>` nodes under the current node. More than one node is found - `show` displays all of the currently selected nodes.



There are four active nodes, as verified by:

```
[S]crape >>> nodes
```

The text of these nodes seems like it would serve nicely as an outline, so let's capture those.

First, declare a table name and a variable to collect output (if you don't declare a table name, the default is `scrape_table`). Setting `var my_name` will select a variable to collect data. The variable does not need to

exist (it will be created). If you change tables before you've saved their output, they are stored so you can later add to their variables (and output).

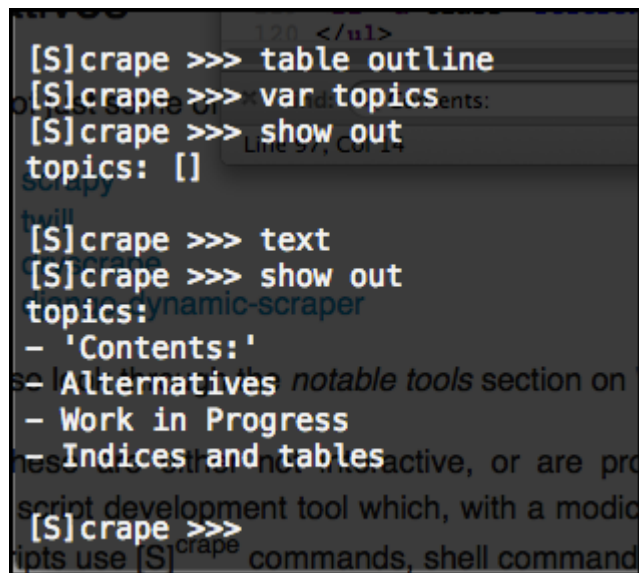
```
[S]crape >>> table outline
[S]crape >>> var topics
```

When you save the output from this table, it will be saved in a file `outline.csv`. You can also save the output as `json` or `yaml`. Once you save a table, its values are emptied. So far, this table has one column - one variable.

To see what the various variables of a table currently have, we issue the `show out` command to show pending output (the current table's contents). `[S]crape` variables are lists of values. Variable names are shown with a colon (`my_var:`), and their values are shown preceded by a `'-'`.

The `text` command will collect text contents of the currently selected HTML nodes into the current variable.

```
[S]crape >>> show out
[S]crape >>> text
[S]crape >>> show out
```



```
[S]crape >>> table outline
[S]crape >>> var topics
[S]crape >>> show out
topics: []

[S]crape >>> text
[S]crape >>> show out
topics:
- 'Contents:'
- Alternatives
- Work in Progress
- Indices and tables
[S]crape >>>
```

There was no output pending prior to the `text` command. If you wanted to save this now, the `table` command (with no argument) will output the current table to a `csv` file with the same name (if one already exists, it will not be overwritten; the name will be numerically extended).

If you want to save your script for later, look at your history. Only scrape commands which act on pages are saved in history. You can choose which parts of history you save to a script file.

```
[S]crape >>> history
[S]crape >>> help save
```

If you'd like, save your script now. You can edit saved `[S]crape` scripts with a text editor. You can add comments, which begin with `'#'` and extend to the end of the line.

There is an alternate form for selecting tables and variables, which may help the commands in your script (and what they apply to) stand out. If you'd like, in place of:

```
table outline
var topics
```

you can equivalently write:

```
[ outline ]  
< topics >
```

To exit [S]^{crape}, see `help EOF`.

After our brief interactive session with [S]^{crape}, here's what our script looks like:

```
##  
# [S]crape script to get outline of a page  
#  
# - gets the text of <h2> headings;  
#  
[ outline ]  
< topics >  
findall ./h2  
text  
table # save outline.csv
```

Summary

In this introductory tutorial, we've

- shown one way to select nodes;
- defined tables and variables;
- saved selected content;
- saved a [S]^{crape} script;

Please continue with the next tutorial.

Happy  ing!

3.2 Developing a Project

This is the second tutorial in a series.

From the introductory tutorial, we saw how to select a destination (file or URL). Initially, it's also beneficial to view the destination's source along with the browser window. You can either search for what interests you in the source window, or use `inspect element` to get to the item that interests you.

Once you've quickly found the item of interest, you can start trying various tree traversal commands to get to related items in [S]^{crape}, view the nodes found, and save some part of their content in [S]^{crape} variables for output. You can also save your script activity into a script, which you can edit and run later in [S]^{crape}.

In this tutorial we'll see how to backtrack and make corrections. We'll also see how the various [S]^{crape} commands behave when applied to multiple nodes.

Note: *A word about [S]^{crape} ing public sites:*

Be a *Good Citizen*!

- avoid repeatedly hitting a site, and loading its servers;

- always check for copyright, and observe fair use doctrines.

3.2.1 PyCon Volunteer Reporting

Here's our project: the US PyCon 2013 Conference is coming up. PyCon is a community conference and depends heavily on volunteers. We want to track how many volunteers we still need for session staff⁴.

The conference site lists the sessions and staff on <http://us.pycon.org/2013/schedule/sessions>. Since this will likely change dynamically, we'll use a snapshot version we saved, just as you would when first developing a script (in order to spare repeatedly hitting a site's servers). Having a static copy will also make it easier to follow along with the tutorial (also, after the conference, there will be no unfulfilled needs, so the web data won't be as interesting):

- download `tutorial2.zip`.

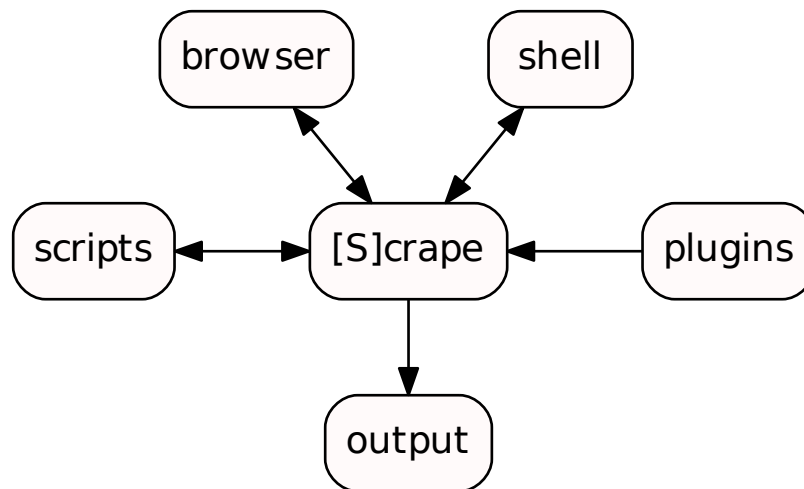
3.2.2 Getting Oriented with [S]^{crape} Commands

Let's review what we've learned so far.

When you open [S]^{crape} with a URL, [S]^{crape} opens the url in a browser and parses it into a tree of nodes held in scrape. These nodes are what you navigate. Using xpath and cssselect you select nodes and extract data.

The ability to inspect aspects during the process is useful, as well as being able to run scripts in batch.

In this tutorial we'll introduce some of the rhyme and reason behind [S]^{crape}. Since [S]^{crape} has over 60 commands, let's start by describing some structure around the commands (we will only introduce some of them in this tutorial).



[S]^{crape} commands affect each of these areas. Most of the action happens in the hub - in [S]^{crape} itself. The type of commands in [S]^{crape} are:

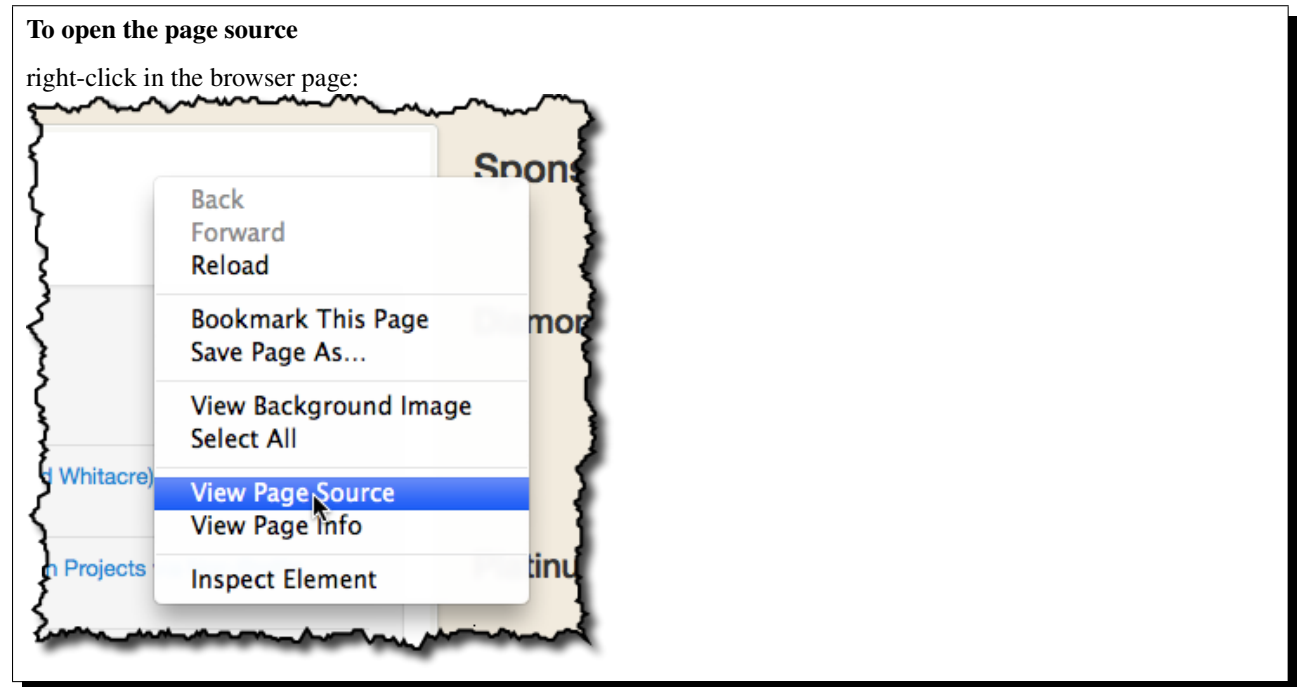
- navigation
- content extraction (capturing)
- interaction
- settings

⁴ Session staff consist of chairs and runners. Chairs introduce speakers, manage questions and keep track of time. Runners get speakers to their talk on time and ensure they have everything they need. A typical session consists of 3 talks. Sessions run simultaneously in multiple rooms.

- variables

3.2.3 A Starting Strategy

The first time you open a target URL it can be useful to open the page's source from the browser (I have them side-by-side at first).



For smaller pages, it can be useful to search in the source for what interested you in the browser. For larger pages, it can sometimes be easier to simply highlight what interests you in the web page, and use the `[S]crape grab` command to give you a small context. From there, it can be easier to search for the larger context in the source window, so you can get a good view of the context around your interest.

Let's do that now. Unzip the tutorial file (I've replaced the >1M in images with a single pixel gif to keep things manageable). You should have a file `sessions.html` and a directory `sessions_files`. Assuming you've unzipped in the current directory run `scrape`:

```
$ scrape sessions.html
```

To orient ourselves, use a few of the interaction commands from the *Introductory Tutorial*:

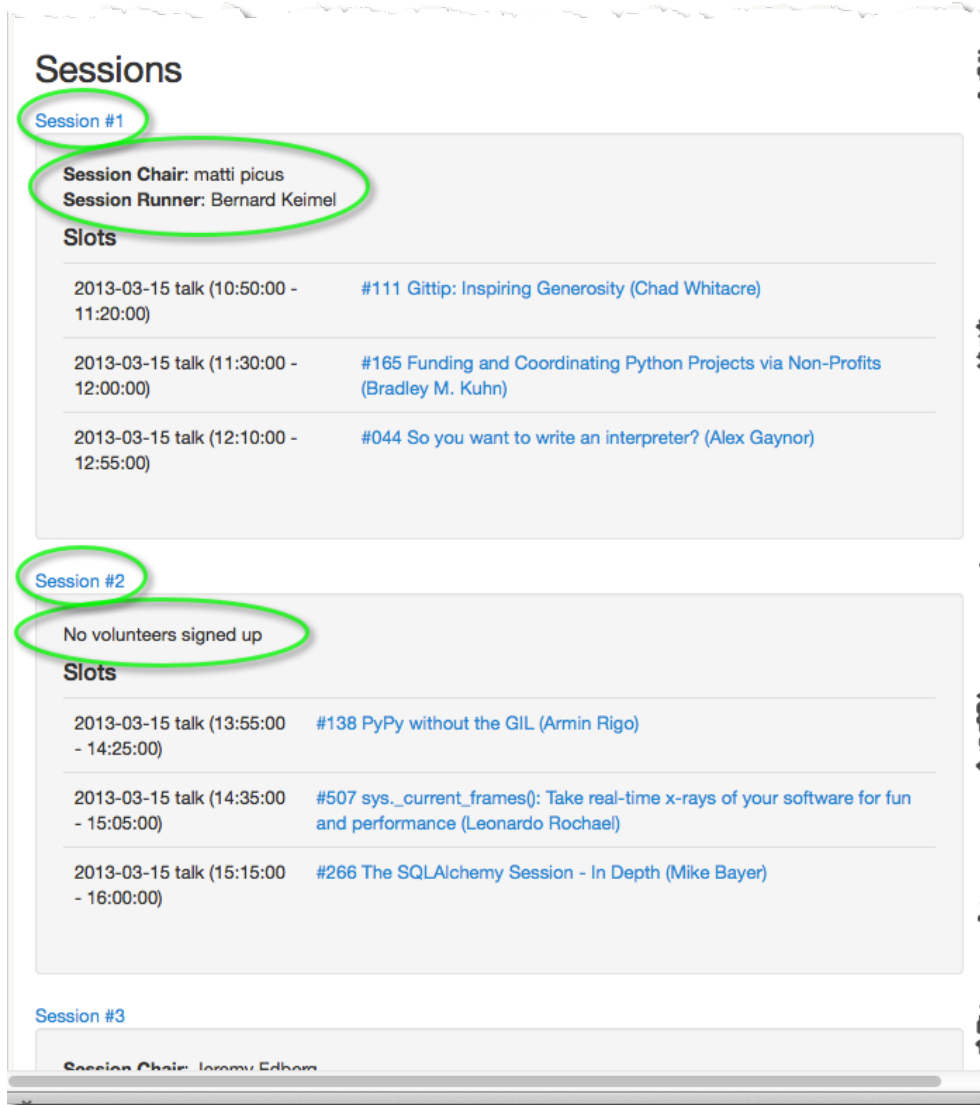
```
[S]crape >>> nodes
2
[S]crape >>> tags
['head', 'body']
[S]crape >>>
```

In this case, we are not concerned with any of the meta-data which might be in the `<head/>`:

```
[S]crape >>> body
[S]crape >>> nodes
7
[S]crape >>> tags
```

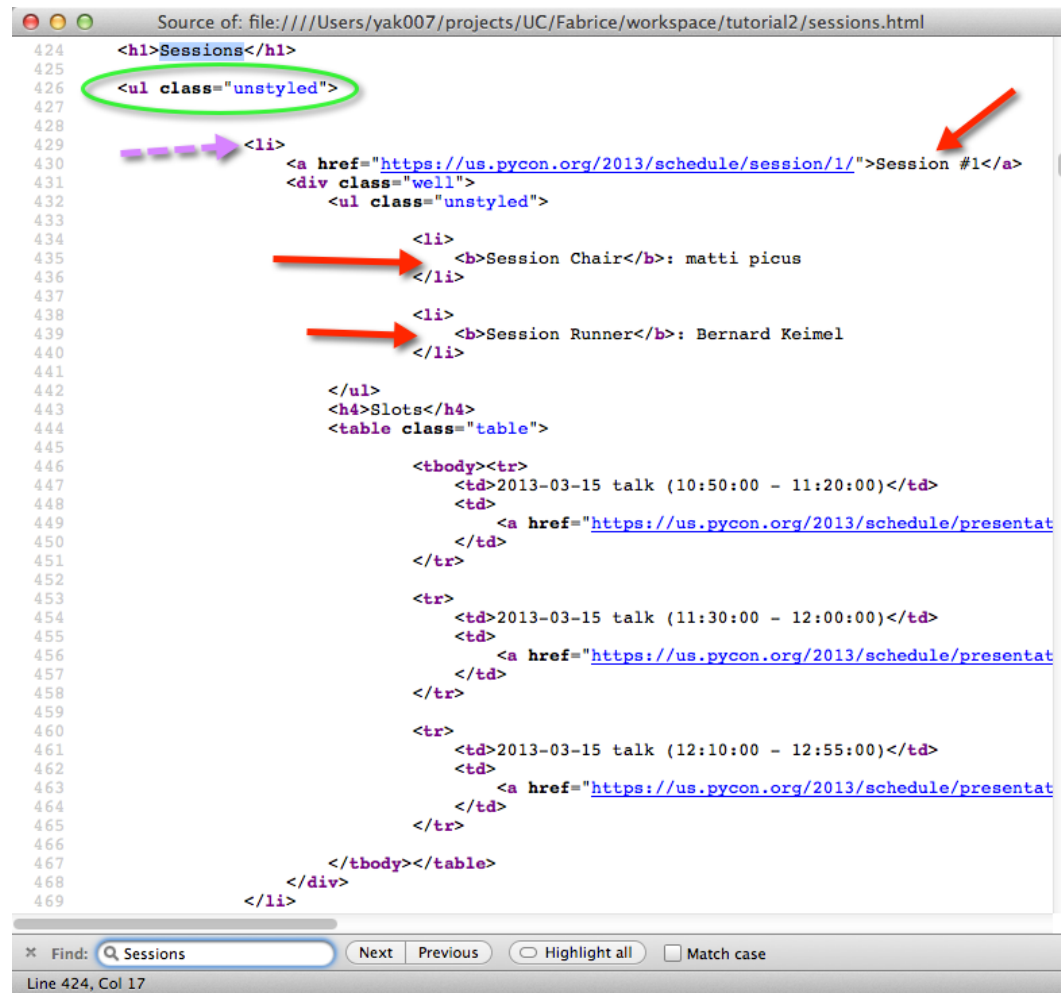
```
[ 'header', 'div', 'script', 'script', 'script', 'script', 'div' ]
[S]crape >>>
```

Looking at our browser window, the sessions are named and listed as visual blocks. Here are the parts interesting for our task:



Scrolling to the bottom of the browser page, we see there are 42 sessions. We can see that each session has a Session Chair and a Session Runner. If no one has signed up, the page shows: No volunteers signed up. We need a total of 84 volunteers. We'll need to gather information after the session name (e.g. Session #1).

Unfortunately, there's a lot of HTML code for headers, sponsors, and so forth - but let's go to our browser's source window and search for Sessions. It looks like our info is all contained in an HTML list.



Let's just start by seeing what happens when we try to get the list of sessions. If we try `findclass`:

```
[S]crape >>> findclass unstyled
[S]crape >>> nodes
43
[S]crape >>>
```

It looks like we might have gotten the 42 session (their content looks to be held in `<ul class="unstyled">` lists), and the outermost list holding them. You can look at what was selected with `show node`, but it's a little easier to digest at this point in the browser-source window. This is close to what we wanted, but not quite.

3.2.4 Adjusting Course

If you use further `[S]crape` navigation commands (such as `findclass`), they will act from each of the currently selected nodes. We're not where we want to be, so let's back up:

```
[S]crape >>> body
```

Some smaller ways you can back up in the tree:

- `doc`, or root (aliases)
- `getprevious`

- `getparent`

See the *help* for these, and experiment with them.

Now, let's try a couple of other commands to see if you can get to the 42 nodes of interest (look for hints in the browser-source view). Here are a few examples (I'll omit the output, so be sure to follow along at your computer):

```
[S]crape >>> flindclass unstyled
[S]crape >>> nodes
[S]crape >>> tags
[S]crape >>> body
[S]crape >>> help cssselect
[S]crape >>> cssselect div.box-content ul.unstyled
[S]crape >>> nodes
[S]crape >>> tags
```

There are a couple of ways to get to what we want (you may find others). Cssselectors are easy to write and powerful. Xpath expressions are explicit and functional (if you learn xpath expressions, you can take advantage of that knowledge for navigating XML documents also).

I find that either `cssselect h1+ul.unstyled` or `find .//ul[@class='unstyled']` work. The css expression says:

get all the elements ul which immediately follow an h1, and which have class unstyled.

The xpath expression says:

get the next (single) ul node with class unstyled.

The `./tag_name` form says look anywhere (any depth) under the current node.

I prefer the xpath expression - for this case, it seems more suitable, closer to what we intend.

```
[S]crape >>> body
[S]crape >>> find .//ul[@class="unstyled"]
[S]crape >>> nodes
[S]crape >>> show node
```

This looks like the spot we were interested in, in the browser-source.

3.2.5 Saving Output

So that we can have context, let's collect the session name. Let's also scrape the text of the first `ul` under that - the session volunteers. I want to have 42 names, and 42 pieces of volunteer information. Thus we can determine which sessions have needs. The first `ul` under each session name will do this for us. First, lets try to select the sessions. From the last `show` command, we can see *Session #42*.

```
[S]crape >>> findall ./li
[S]crape >>> nodes
[S]crape >>> show
```

Note that `findall` has a single `/'/'` - this will find only direct children of our current `ul` node.

Now lets get our session names:

```
[S]crape >>> find ./a
[S]crape >>> nodes
[S]crape >>> show
```

We use `find` (not `findall`) because we only want the first `a` tag under each of our 42 nodes.

This time, the *show* command is a joy to look at - it's clear that we have the session names, that our 42 nodes are indeed exactly what we want. The text of these nodes contain the session names we want. We're ready to setup some variables:

```
[S]crape >>> [sessions_table]
[S]crape >>> <session>
[S]crape >>> text
[S]crape >>> show out
```

We have our 42 session names waiting to be output. But still we need to add information about the volunteer status of each.

Thankfully, we have the browser-source window to refer to. We can see that after the `<a>` containing our session names we want the `ul` nodes which are the first children of the `div` tag following `li`.

The **getnext** command will get the next sibling node (the `div` we want). From there we will get the `ul` directly under:

```
[S]crape >>> getnext
[S]crape >>> nodes      # confirm
[S]crape >>> show
[S]crape >>> find ./ul
[S]crape >>> nodes      # still looks good
```

Where *text* will get the text inside the tag (up to the next child tag), *text_content* will get *all* the text inside a tag, even that inside other enclosed nodes. We're ready to save the status of the volunteers - we'll put this in a *volunteer* variable.

```
[S]crape >>> <volunteer>
[S]crape >>> text_content
[S]crape >>> show out
```

There is a good deal of *white space*, but we'll easily deal with that outside of scrape. I think the form of *show out* (yaml) would be easy to read into a python script which will do the counting.

```
[S]crape >>> yaml sessions.yaml
```

You could have also saved this as either *json* or *csv* (the latter using the *table* command). Either json or yaml is convenient for loading into python data structures. I chose yaml because it is easy on the eyes when viewing the scraped data file.

3.2.6 Running Another Day

We'll need to run this script quite often to keep the current volunteer needs up to date, so we'll need to save our script.

Have a look at your history:

```
[S]crape >>> history
```

Notice that history shows your navigation commands, but not your interactive inspection commands. Scripts are saved from this command history, so inspection commands are not stored there.

You could edit your script file (comments start with `'#'`), and eliminate any false starts, and test the edited result, or you could select which part of your history to save, and go from there. You decide:

```
[S]crape >>> help save
[S]crape >>> save sessions.scrape
```

Before you exit `[S]crape`, edit your file, and test it by running it against the current page (I use the *gvim* editor; you should use your favorite):


```
[S]crape >>> clear volunteers session # clear variables
[S]crape >>> show out # should now have no output pending
[S]crape >>> load sessions.scrape
[S]crape >>> show out
```

Loading a script runs it against the current document tree.

You can run your script in *headless* mode:

```
$ scrape -H -s sessions.scrape http://us.pycon.org/2013/schedule/sessions
```

I leave it to you to develop a script to count and report on volunteer needs, based on *sessions.yaml*. Mine was under 12 lines of python. Whatever you use for postprocessing, you can also run it from your *sessions.scrape* by adding something like this to the bottom of your script:

```
# after saving your yaml / json / csv file:
!python my_script.py sessions.yaml
```

3.2.7 Summary

After this exercise, your script should look similar:

```
## Count volunteer signups for PyCon Sessions
#
# open http://us.pycon.org/2013/schedule/sessions/
#
body
# I save to a different name than this table, which would be default;
[s_table]
find ./ul[@class='unstyled']
findall ./li
find ./a
<session> # column1: the session name
text
getnext
find ./ul
<txt> # column2: who's signed up to staff the session;
text_content
yaml sessions
!python session_volunteer_counter.py sessions.yaml
```

Let's look at which commands in [S]^{crape} we used:

- navigation:
 - body
 - cssselect
 - find
 - findall
 - findclass
 - getnext
- capturing:
 - text

- text_content
- interaction:
 - help
 - history
 - nodes
 - show
 - show out
 - tags
- settings:
 - headless (-H)
- variables:
 - clear
 - “[...]”, or table
 - “<...>”, or var
- output:
 - yaml
- scripts:
 - load
 - save
- shell:
 - “!”, or shell



Happy  ing!

3.3 Introduction to [S]crape

The “Getting Started” Tutorial will help you confirm your installation, and introduce the basic

 concepts.

3.4 Developing a Project

 In this [Intermediate Tutorial](#), we’ll look at using  to report on volunteers signed up for running talks at a national conference. In the process, will look at the different groups

[s]crape

of commands in scripts. and introduce some useful patterns for developing

Copyright Notice

Copyright (c) 2013, The University of Chicago. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of The University of Chicago nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Overview

What's involved, required knowledge, and the basic modes of operation.

Installation

Software and browser requirements.

Tutorials

Get started with a concrete examples.

Alternatives


Examples of just some of the alternatives to [S]^{crape} include:

- scrapy
- twill
- dryscrape
- django-dynamic-scraper

You can also look through the *notable tools* section on [Wikipedia](#).

Many of these are either not interactive, or are programmers libraries or toolkits. [S]^{crape} is an interactive script development tool which, with a modicum of knowledge, is both powerful and simple. [S]^{crape} scripts use [S]^{crape} commands, shell commands, and commands provided by extensions.

Copyright Notice

This license applies to the program, , and its documentation.

Work in Progress

This document is currently a work-in-progress. Here are a list of known items left to do:

Todo

Have yet to debug the scrape.gz install file (installation does not mirror setup.py).

(The original entry is located in `/home/docs/checkouts/readthedocs.org/user_builds/scrape/checkouts/latest/source/installation.rst`, line 92.)

Indices and tables

- `genindex`
- `modindex`
- `search`